Technical University of Koszalin

Natalia Maslennikowa

FAULT-TOLERANT IMPLEMENTATION OF THE LINEAR ALGEBRA ALGORITHMS WITH THE FIELD PROGRAMMABLE GATE ARRAYS (FPGA) BASED ON THE LOW SWITCHING NOISE CURRENT-MODE GATES

DISSERTATION

Koszalin 1999

FAULT-TOLERANT IMPLEMENTATION OF THE LINEAR ALGEBRA ALGORITHMS WITH THE FIELD PROGRAMMABLE GATE ARRAYS (FPGA) BASED ON THE LOW SWITCHING NOISE CURRENT-MODE GATES

CONTENTS

INTRODUCTION

The methods of linear algebra (LA) make a basis for mathematical models in various fields of science, engineering and technology [4] such as signal and image processing, system theory, statistical and numerical analysis, biomedical researches, physical experiments, etc. For example, modeling of many real life problems requires solving complex systems of difference equations, which using finite elements method are finally reduced to a linear systems, created for finite number of elements or nodes. Thus, when solving the majority of problems, the main computing procedures of these methods can be reduced to operations over large size matrices. Here are some of the specific problems to be solved by modern systems of real-time signal processing: matrix multiplications for covariance estimation, solving linear systems in adaptive processing, computing eigenvalues/eigenvectors for high-resolution array processing and adaptive beamforming [4]. However, most of LA methods are characterised by high computational complexity [1, 2] ($O(N^3)$ multiplication with addition operations, where N is the order of input matrix A). This implies the necessity of solving these problems on high performance consequential or parallel computers. However, the universality of these systems causes their high hardware overhead and few hardware utilisation. Therefore, the application-specific parallel systems (ASPS) destined to implementation of several applied algorithms and adapted to their properties are more suitable to real time processing. The application areas of these computers demand a large degree of reliability of output results. However, along with the increasing of computational complexity and complexity of computers, the probability of physical failures increases. Since a single temporary or permanent failure in a processor can break down an entire computing system, fault tolerance should be provided in these cases or on hardware, or(and) on software levels.

Recent advantages in VLSI technology have stimulated research in application-specific architectures which are tailored to particular application and intended for implementation as ASICs or FPGA-based devices. Among these architectures are, in particular, VLSI processor arrays [3,6-8,10,11,22,29,30,33]. Although advances in semiconductor manufacturing permit an unprecedented number of transistors on a single processor die, the several fundamental technological limits are existed, which cause technological as well as methodological requirements to designing of VLSI systems.

For example, while the realization of internal connections into VLSI chips is a actual problem, the fundamental technological requirement is the locality and regularity of interconnections between processor elements (PE) of the ASPS. Hence, maximum efficiency of the algorithm realization may be derived only when it is prepared (modified) in such a way, that data streams between PEs adequately correspond to the information dependencies in the implemented algorithm and the PEs loading is balanced [22,26-28]. Thus, fundamentally, the two ways in which a larger volume of resources (e.g., more transistors) improves performance are parallelism and locality [8].

Another technological requirements are regularity of the ASPS structure and minimal numbers of external I/O channels and different types of PEs. Moreover, in the case of location together (on a common chip surface) analog and digital parts of a of mixed analog-digital VLSI system, the important requirement is the minimization of the switching noise, generated by the digital part of a system on the it analog part [92,97].

From above mentioned technological requirements directly follow the methodological requirements and problems of VLSI ASPS systems: using „up-to-down" approach to the system designing, adapting algorithms to effective VLSI realisation (i.e., VLSI algorithm designing) and system fault tolerance. And if the most technological requirements are satisfied the FPGA chips implemented on the base of current-mode gates [88,90,93], then satisfying mentioned methodological requirements are actual problems mainly due to requirement of system fault tolerance which guaranties the confidence of computations. Thus, the elaboration of methods of fault tolerant VLSI ASPS designing is the actual problem.

Therefore, the manuscript purpose is the improving methods of the application-specific parallel processor design and their using for the synthesis of the fault-tolerant processor arrays destined to the implementation on the FPGA's with the low level of the switching noise current-mode gates, and the deriving of the fault-tolerant versions of the main linear algebra algorithms.

In according to this purpose, the following problems are solved in the manuscript:

1. requirements to algorithms and application-specific architectures for their effective realization in VLSI, and selection of the architectural platform for the realization of main LA tasks;

2. analysis of known fault tolerance methods and selecting of most suitable from them for the designing of fault-tolerant VLSI parallel systems;
3. requirements to methods of the application-specific architectures designing and analysis of the known mapping methods for searching of most suitable for the designing of fault-tolerant VLSI parallel processors;
4. modification of the weighted checksums (WCS) method and deriving of the fault tolerant versions of main LA algorithms;
5. designing of FPGA-based parallel system architecture for the fault tolerant realization of main linear algebra algorithms;
6. approaches to the minimization of current-mode logical functions and designing of binary current-mode digital circuits;
7. designing of the current-mode prototype of basic block of the FPGA XC4000 series cell.

METHODS OF RESEARCHES.

The methods of linear algebra, graph theory, linear operators, set theory, theory of system design.

SCIENTIFIC INNOVATIONS OF MANUSCRIPT:

1. The modification of the origin WCS method is carried out and sufficient conditions of it using for LA algorithms were formed. The proposed method operates with wider set of LA algorithms and allows to design the effective fault-tolerant versions of algorithms.
2. The fault tolerant versions of the Gauss elimination, LU-decomposition, Choleski and Jordan-Gauss algorithms were designed using modified WCS method. They enable to detect and to correct a single error in an arbitrary row or column of the input matrix at the each algorithm step. Hence, it is possible to correct up to $N^2/2$, $N^2/4$ and N^2 single errors during realization of the whole Gauss, Choleski and Jordan-Gauss algorithms respectively.
3. the new method for the construction of the lattice DGs of algorithms given by nested loops has been proposed. In a contrast with known analytical methods, the proposed method is more simple and feasible for the implementation in CAD systems, and

allows operating with a wider class of algorithms such as, for example, non-uniform recursive algorithms corresponding to non-perfect (or composite) loop nests.

4. the structure of the application-specific parallel system destined to the fault-tolerant implementation of proposed algorithms is designed. Thus, was proved the confirmation, that using FPGA-chips and libraries of files with configuration data for these chips, it is possible to construct the fast adapted (to the implemented algorithms) application-specific system with high performance and lowest cost/performance ratio.
5. The logical properties of the current-mode gates and logic and the several identities for the conversion of expressions from the Boolean algebra were proposed. They are a base of the proposed approach to the design of the digital current-mode circuits and allow to reduce the hardware overheads for circuits realization.
6. Using proposed approach, the functional schemes of the several current-mode digital circuits were derived. They are characterized by smaller hardware overhead in comparison with similar ones based on others gate types.

REALIZATION OF SCIENTIFIC INNOVATIONS:

The theoretical and practical results of manuscript are support by grant KBN 8T11B 049 10.

APPROBATION OF SCIENTIFIC INNOVATIONS OF THE MANUSCRIPT:

The main scientific results of the manuscript are discussed at:

- International Conference CAD DD'95, Minsk, Bielarus, 1995г.
- International Workshop „Parallel Numerics'95", Sorrento, Italy, 1995;
- „International Conference on Signal Processing Applications&Technology", ISPAT'95, Boston, USA, 1995;
- International Workshop „Parallel Numerics'96", Gozd Martujek, Slovenia, 1996;
- „9-th European Workshop on dependable computing", Gdansk, Poland, 1998
- XXI Conference KKTOiUE, Poznan, Poland, 1998
- XXII Conference KKTOiUE, Warszawa, Poland, 1999
- International Workshop „Parallel Numerics'99", Salzburg, Austria, 1999;
- 6-th International Conference MIXDES'99, Krakow, Poland, 1999

- 2-th Conference „RUC'99", Szczecin, Poland, 1999
- 3-th International Conference PPAM'99, Kazimierz Dolny, Poland, 1999

PUBLICATIONS. Theme of manuscript are connected with 17 scientific papers.

STRUCTURE OF MANUSCRIPT. Manuscript consists of introduction, four chapters, conclusions and references with 103 papers. Manuscript consist pages, including tables and figures.

In the first chapter, the requirements to algorithms and application-specific architectures for their effective realization in VLSI are determined, and the selection of the architectural platform for the realization of main LA tasks are performed. Moreover, the analysis of known fault tolerance methods and selecting of most suitable from them for the designing of fault-tolerant VLSI parallel systems are carried out.

In the second chapter, the modification of the origin WCS method is carried out and sufficient conditions of it using for LA algorithms were formed. Based on the proposed method the fault tolerant versions of the Gauss elimination, LU-decomposition, Choleski and Jordan-Gauss algorithms were designed. They enable to detect and to correct a single error in an arbitrary row or column of the input matrix at the each algorithm step. Hence, it is possible to correct up to $N^2/2$, $N^2/4$ and N^2 single errors during realization of the whole Gauss, Choleski and Jordan-Gauss algorithms respectively.

In the third chapter, the new method for the construction of the lattice DGs of algorithms given by nested loops has been proposed. In a contrast with known analytical methods, the proposed method is more simple and feasible for the implementation in CAD systems, and allows operating with a wider class of algorithms such as, for example, non-uniform recursive algorithms corresponding to non-perfect (or composite) loop nests. Then the processor array architectures performing fault-tolerant version of Jordan-Gauss algorithm with the partial pivoting, Cholesky, Gauss elimination and back substitution algorithms has been designed. Note, that in the order to deriving of the array architectures with desired features, some purposive transformations of the basic algorithm dependence graphs are employed. Based on these architectures, the structure of the

application-specific parallel system destined to the fault-tolerant implementation of proposed algorithms is designed.

In the fourth chapter, the logical properties of the current-mode gates and logic and the several identities for the conversion of expressions from the Boolean algebra is proposed. They are a base of the proposed approach to the design of the digital current-mode circuits and allow to reduce the hardware overheads for circuits realization. Then, using proposed approach, the functional schemes of the several current-mode digital circuits were derived. The proposed circuits are characterized by smaller hardware overhead in comparison with similar ones based on others gate types.

In the conclusions, the main scientific innovates of manuscript are formulated.

IT IS DEFENDED:

1. Algorithm-based fault tolerance method for main linear algebra tasks;
2. Fault-tolerant versions of the Gauss elimination, LU-decomposition, Choleski and Jordan-Gauss algorithms and fixed size processor array architectures for their realization.
3. Method of deriving dependence graphs of regular algorithms.
4. Main identities of the current-mode logic and approaches to the minimization of binary current-mode function.
5. current-mode prototypes of main standard digital circuits and basic block of FPGA cell.

CHAPTER 1. MAIN DESIGNING PROBLEMS OF APPLICATION-SPECIFIC PARALLEL PROCESSORS FOR LINEAR ALGEBRA (LA) TASKS

1.1. Requirements to algorithms and application-specific architectures for their effective realization in VLSI. Determination of the architectural platform for the realization of main LA tasks

Advances in semiconductor manufacturing permit an unprecedented number of transistors on a single processor die. But what architecture will make the best use of these riches? During an informal discussion about the future of microprocessors and billion-transistor architectures at „Computer" journal, several architectural platforms had a boisterous arguments over the direction that next-generation processor architectures will take. They are [9, 12, 13, 15, 16]:

- - reconfigurable parallel computing engines (FPGA-based and raw-computers);
- - specialized, very long instruction word (VLIW) machines;
- - wide, simultaneous multithreaded (SMT) uniprocessors;
- - single-chip multiprocessors (CMP);
- - memory-centric computing engines, such as intelligent RAM (IRAM);
- - very wide conventional superscalars; and
- - wide superspeculative processors.

The choice of a best from the mentioned architecture platforms mainly depends from the intention of a target computer system, i.e. from a set of tasks which should be realized by this system and needed performance. Therefore, the properties of main linear algebra algorithms should be firstly considered.

There are few numbers of main LA tasks. They are the solving linear systems and least squares problem, matrix multiplication and inversion, and matrix eigenvalues (or singular values) problem [19, 20, 21, 23]. However, many methods and algorithms for their solution are existed (see Fig.1). The selection of a method mainly depends from the input matrix type (symmetrical, band, squared, etc.) and as well as from the parameters of the SPCS (such as numbers, performance and instruction set of PEs, numbers and throughout of I/O channels, etc.) [22, 24, 25, 27].

The common properties of the most LA methods are a high computational complexity [19, 21, 24, 25, 26] ($O(N^3)$ multiply-addition operations, where N is the order of an input dense matrix $\boldsymbol{A}$) and regularity. The regularity means here that identical transformations are carried out with nearly all elements of input matrix at the each algorithm step. Moreover, the order of the computations is independent from the values of the input data. More details, the most of LA methods (algorithms) consists of the computation on the any i-th algorithm step (may be not one time) the elements of leading (i-th) row or/and column of matrix $\mathbf{A^i} = \{a_{ji}^{\mathbf{i}}\}$. Then the modification others matrix rows (columns) by means leading rows are performed. The example of corresponding algorithm fragment is represented by means construction (1.1), were values of the variables *K*, *K1*, *K2* and functions *f1, f2* are depended from the selected algorithm. In this example, the input matrix $\mathbf{A = A^1} = \{a_{ji}\}$ is recursively modified during *K* computation steps to the resulting matrix $\mathbf{A^{K+1}}$.

for *i:=1* **to** *K* **do**
begin
{Phase1:computation of the leading column elements a_{ji}^{i+1} *}*
for *j:=i+1* **to** *K1* **do**
$a_{ji}^{i+1} := f1(a_{ji}^{i}, a_{ii}^{i})$ (1.1)
{Phase2: computation of the elements of the matrix $\boldsymbol{A^{i+1}}$ *}*
for *j=i+1* **to** *K1* **do**
for *k:=i+1* **to** *K2* **do**
$a_{jk}^{i+1} := f2(a_{jk}^{i}, a_{ji}^{i+1}, a_{ik}^{i})$.
end

The analysis of the construction (1.1) shows that assignment statement of any loop body would be performed simultaneously for the all elements a_{ji}^{i+1} and a_{jk}^{i+1}. Therefore, the main LA algorithms may be effectively solved by different kinds parallel computers.

The real time LA applications is the object of this manuscript. Only specialized or reconfigurable parallel architectures (destined for the further realization as ASIC, FPGA-based or Raw-processors devices) may be effectively used for these applications. Therefore, the problems of designing of such specialized parallel architectures for main

LA algorithms is considered in this paper. The VLSI processor arrays [3, 7, 8, 22, 26, 27] are typical examples of such architectures.

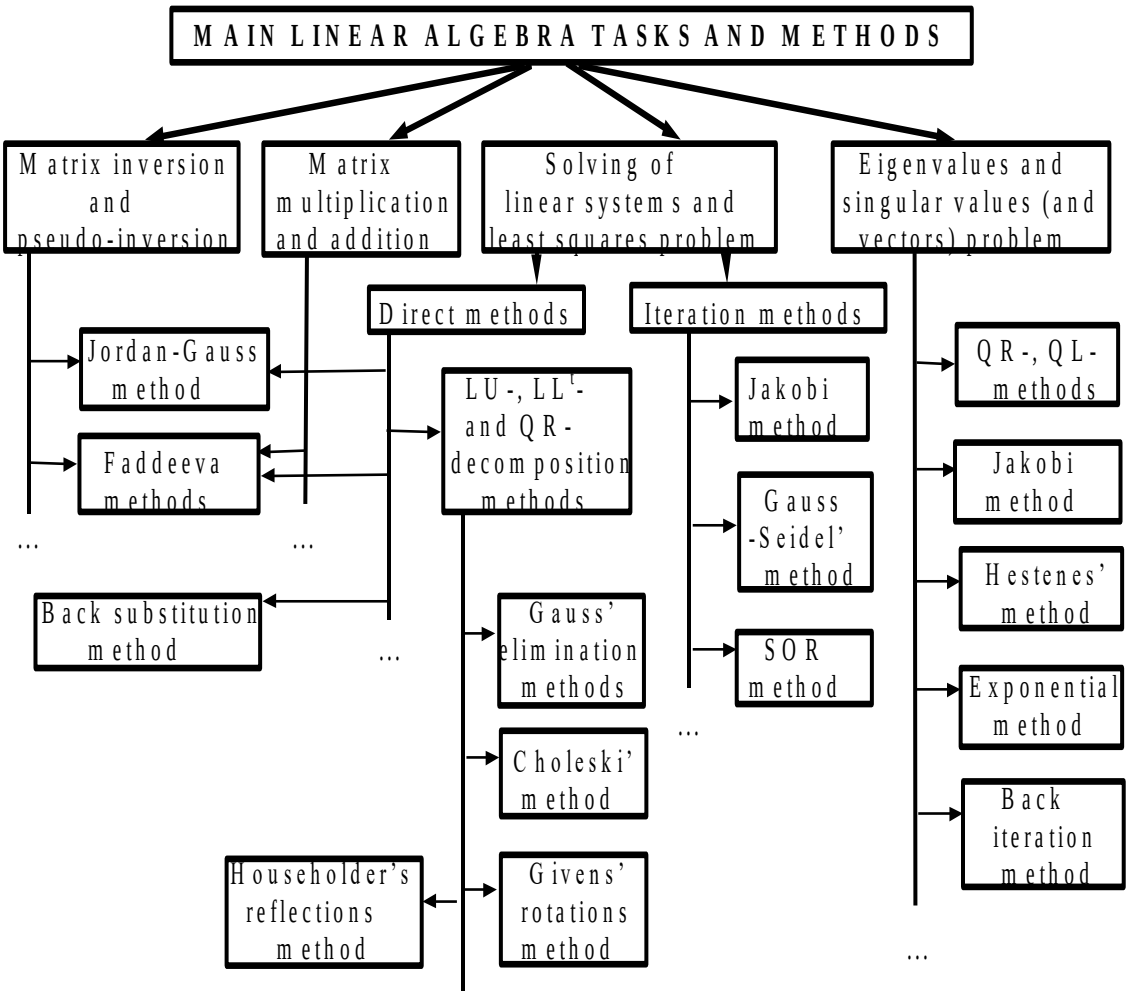


Fig.1.1. Main linear algebra tasks and methods

In an arbitrary case, devices will be realized in the VLSI technology. Therefore, the VLSI technology requirements also should be taken into account during target architecture and parallel algorithm design [1, 5].

VLSI technology needs from the designer the solving of such important problems, as high complexity of the designing, communication problems, effective utilization of a chip area, fault tolerance and low level of switching noise (for mixed analog-digital systems), problems of control and synchronization, etc. However, the fundamental factor is the very

higher cost of the data transmission in a comparison to the data transformation and storage [5]. Moreover, the critical factors are also I/O channel widths (due to limit of the chip pins) and their throughout. Therefore, in the order to obtaining of the better cost, performance, scalability, and control organization parameters, the implemented in VLSI systems should be application specific and parallel systems, consisting of mainly identical PEs with local interconnections [2]. Moreover, the maximal efficiency of the such system will be derived in the case when data streams between PEs adequately correspond to the information dependencies in the implemented algorithm and the PEs loading is balanced [2,3]. Thus, fundamentally, the two ways in which a larger volume of resources (e.g., more transistors) improves performance are parallelism and locality [8].

Therefore, the organization of the parallel computations needs the modification of the numerical algorithms base, which was designed to the sequential computers, because the criteria's of the VLSI algorithms designing and evaluating are differences. The reduction of the criteria of the algorithm computational complexity corresponds now the increasing of the criteria's of the regularity and locality of data dependencies in the algorithm [3, 22, 26, 27, 28, 29, 30]. It will be shown in the chapter 3, that such parameters for the selected algorithm can be very simple determined by means analysis of its dependence graph [3, 7, 31, 33, 73]. Following definitions is needed for this.

Definition 1.1. Assumes, that algorithm is determined, if are determined:

- - set of algorithm variables;
- - set of algorithm operations;
- - data dependencies in the algorithm.

Definition 1.2. Let each functional operator in the algorithm corresponds to one node I=(i1,...,in) belonging to the some integer lattice I^n named iteration space of an algorithm. Moreover, if the function operator corresponding to a node I2 depends on the functional operator corresponding to another node I1, this dependence is represented by the arc (vector) d=I2-I1 between these nodes. Obtained by a such way graph is named the dependence graph of an algorithm.

In further, the all algorithms will be considered at the function operation level.

Definition 1.3. Algorithm, for which dependence graph is independent from the input data values is named as unconditional algorithm.

Definition 1.4. Regular algorithm is the algorithm represented by the regular dependence graph. Such graphs are characterized by the almost identical set of arcs belonging to the each graph node.

Hence, if dependence graph of the applied algorithm is a regular graph with short arcs and few numbers of node types (i.e. numbers of different functional operators in the algorithm), then such algorithm can be effectively implemented in VLSI processor arrays.

Thus, VLSI algorithms should be characterized by following properties:

1. to posses the regular dependence graph with short arcs and few numbers of different node types;
2. to provide the load balance between different graph fragments;
3. to provide the some level of the fault tolerance (or confidence computing).

In the case when target VLSI system should be realized several different algorithms in the real time mode is it needed to provide the flexibility (reconfiguration) of the system structure and its fault tolerance. In the Ref.[67] is shown, that for VLSI processor arrays effective methods of fault tolerance are the reconfigurations methods [43-45] and algorithm-based fault tolerance (ABFT) methods [58-69]. However, the ABFT methods are needs in lower overhead for their realization [67, 74, 77, 80, 81].

Thus, VLSI parallel system architectures should be satisfied to following requirements:

1. Regularity and locality of data links between PE;
2. Locality of control links;
3. Correspondence of the system performance to the throughout of I/O channels;
4. Correspondence of the PEs performance to the throughout of data links between PEs;
5. Minimal numbers of the I/O channels;
6. Scalability and programmability (possibility of the system programmed reconfiguration to the implementation of the selected algorithm);
7. Fault tolerance and low level of switching noise (for mixed analog-digital systems)

It is follows from the above demands, that reconfigurable computers are most suitable for the VLSI parallel system architectures. At present time, researchers have proposed reconfigurable computers that employ large arrays of highly programmable building blocks. Typical examples are complex programmable logic devices (CPLD) and

field-programmable gate arrays (FPGA) [6, 10, 11, 17]. These devices rely on powerful soft ware tools to map application to the reconfigurable hardware. The intent is to construct powerful, specialized computing engines at a relatively low cost with very short turn-around time. FPGA-based machines achieve their speeds by exploiting fine-grained parallelism and fast static communication. An FPGA's software has access to its low-level details, allowing the software to optimize mapping of the user application. Users can be also bind commonly used instruction sequences into configurable logic. As a result, these special-purpose instruction sequences can execute in a single cycle. This approach's performance scalability has yet to be demonstrated beyond some specialized applications. Note, that a Raw-machine [16] also incorporates most of these features. However, Raw-computers are reconfigurable devices at a higher level - the level of the processor elements. Each PE has the fixed size data and instruction memory, and ALU with the fixed instruction set and data width. However, such fixed parameters are not always optimal for different algorithms. Therefore, the utilization of this devises typically is lower in comparison with FPGA-based devices.

However, for the minimization of the switching noise, generated by the digital part of mixed analog-digital VLSI system on the it analog part it is needed to apply the corresponding methods. In the chapter 4 will be shown, that the radical solution of this problem is based on the implementation of the digital part of the system (FPGA chip here) with the current-mode gates [88-98]. Moreover, using such gates allows to reduce the hardware overhead for the digital circuit realization (for example, adders, decoders, counters, etc.) [82—85,90,91].

Thus, the problem of the VLSI parallel system designing reduces to the designing:

1. parallel versions of the applied algorithms suitable to the VLSI realization;
2. reconfigurable system architecture which to provide the maximal efficiency and confidence of the computation and to satisfied to the VLSI technology demands.

It was be shown above, that the most LA algorithms are the unconditional and regular. Therefore, in the following manuscript chapters the theoretical results obtained into chapter 2 will be used for the designing of the VLSI parallel processor architectures for the solving of main linear algebra tasks.

1.2. Analysis of known fault tolerance methods and selecting of most suitable from them for the designing of fault-tolerant VLSI parallel systems

Modern computer systems is so more complex and so more susceptible to failure. In some instance failures are no more than annoyances; in others (for example, for real-time processing) they result in significant losses. Now it is needed to explore ways to deliver high-confidence computing to most users. The industry needs techniques that add reliability without adding significant cost [37].

The ideal system would be perfectly reliable and never fail. This, of course, is impossible to achieve in practice: System builders have finite resources to devote to reliability and consumers will only pay so much for this feature. Therefore, fault-tolerance is a best guarantee that high-confidence systems will not betray the intentions of their users by succumbing to physical, design, or human-machine interaction faults, or by allowing viruses and malicious acts to disrupt essential services [39, 40, 57].

A. Avizienis originally formulated the concept of fault-tolerance in 1967: „The system is fault-tolerant if its programs can be properly executed despite the occurrence of logic faults [34]. Systems fail for many reasons. Five classes of faults are relevant: physical, design, operator, environmental, and reconfiguration [38]. However, in either case, faults will cause errors. In this manuscript, we consider only physical (or hardware) and environmental faults that occur while the system is in use and which are independent from human actions. They are permanent and transient faults [36]. Note, that as we move to smaller and smaller VLSI implementations, transient hardware faults will predominate; today about 80 percent of all hardware faults are transient [38]. Transient faults are difficult to diagnose and, worse, can corrupt data. Therefore, the methods of transient faults detection, location and correction are considered in this manuscript.

The basic principle of fault-tolerant design is redundancy [36, 37, 39, 57], and there are three basic approaches to achieve it: spatial (redundant hardware) [45-49], informational (redundant data structures) [41, 42, 63] and temporal (redundant computation) [43, 44, 50-55].

Redundancy costs both money and time. The designers of fault-tolerant systems, therefore, must optimize the design by trading off the amount of redundancy used and the

desired level of fault tolerance. Temporal redundancy usually requires recomputation and typically results in a slower recovery from failure compared to spatial redundancy. On the other hand, spatial redundancy increases hardware costs, weight, and power requirements.

Many fault tolerance techniques can be implemented using only special hardware or software, and some techniques require a combination of these. Which method is used depends on the system requirements: hardware techniques tend to provide better performance at an increase hardware cost; software techniques increase software design costs. Software techniques, however, are more flexible because software can be changed after the system has been built.

Below the brief characteristics of the most widely used hardware and software techniques are represented [37].

Modular redundancy uses multiple, identical replicas of hardware modules and a voter mechanism. The voter compares the outputs from the replicas and determines the correct output using, for example, majority vote. Modular redundancy is a general technique - it can tolerant most hardware faults in a minority of the hardware modules.

N-version programming can tolerate both hardware and software faults. The basic idea is to write multiply versions of a software module. A voter receives outputs from these versions and determines the correct output. The different versions are written by different teams, with the hope that these versions will not contain the same bugs. N-version programming can therefore tolerate same software bugs that affect a minority of the replicas. However, it does not tolerate correlated faults, which may be catastrophic.

Replication is effective but expensive. For certain applications, such as RAM and buses, *error-correcting codes* can be used, and they require much less redundancy than replication. Hamming and reed-Solomon codes are among those commonly used.

A *checkpoint* is a copy of an application's state saved in some storage that is immune to the failures under consideration. A *rollback* restarts the execution from a previously saved checkpoint. When a failure occurs, the application's state is rolled back to the previous checkpoint and restarted from there. This technique can be used to recover from transient as well as permanent hardware failures and certain types of software failures. Both uniprocessor and distributed applications can uses rollback recovery.

The *recovery block* technique uses multiply alternates to perform the same function. One module is the primary module; the others are secondary modules. When the primary module completes executions, its outcome is checked by an acceptance test. If the output is not acceptable, a secondary module executes, and so on, until either acceptable output is obtained or the alternates are exhausted. Recovery blocks can tolerate software failures because the alternates are usually implemented with different approaches (algorithms).

Thus, the known methods which use hardware or time redundancy, essentially increase the cost or degrade the performance of computational systems. Therefore, they are few suitable for real-time computing systems and parallel processors. The algorithm-based fault tolerance (ABFT) methods [58-68] are more suitable for such systems. ABFT is an error detection, location and correction scheme which uses redundant computations within the algorithms to detect and correct errors caused by transient failures in the hardware , concurrently with normal operation [58]. In ABFT, the input data are encoded in the form of error detecting or (and) correcting codes. The algorithm is modified to operate on encoded data and produce encoded outputs, from which useful information can be recovered very easily. The modified algorithm will be more complexity (approximately on 1-10%) and therefore, will take more time to operate on the encoded data when compared to the original algorithm. This time overhead must not be excessive. Thus, ABFT methods establish the rules of the original (applied) algorithms and input data arrays modification. From this, it is clear that this methods are not a general mechanism as other methods (e.g. the triple modular redundancy TMR), because they are varied from algorithm to algorithm. However, when the modified algorithm is actually executed on a target architecture, the overheads are required to be minimum in comparison to TMR.

Module-level faults are assumed [15] in the algorithm-based fault tolerance. A module (processor or PE for parallel computers here) is allowed to produce arbitrary logical errors under physical failure mechanism. This assumption is quite general since it does not assume any technology-dependent fault model. Without loss of generality, a single module error is assumed in this manuscript. Also, communication links are supposed to be fault-free.

The most known ABFT method called weighted checksum (WCS) one, which is specially tailored for matrix algorithms and array architectures, has been proposed by Abraham et al. [58,59]. In their scheme, redundancy is encoded at the matrix level by augmenting the original matrix with weighted checksums. Since the checksum property is preserved for various matrix operations, these checksums are able to detect and correct errors in the resultant matrix. Furthermore, the complexity of correction process is much smaller than that of the original computation. For example, the computational complexity of the modified fault-tolerant version of the matrix multiplication algorithm A(N,N)*B(N,N)=C(N,N) increases on the $2N^2$ operations and is equal to $(N^3+ 2N^2)$ multiply-add operations. However, this version allows to detect and corrects the single error among elements of each column of an input matrix A(M,N) occurred during algorithm implementation. Consequently, it enables to correct up to N single errors during solving the whole matrix multiplication task. Note, that the detail description of this algorithm is represented in the section 2.1.

However (it will be proved bellow), the original WCS method don't suitable for most LA algorithms (for example, such as Gauss elimination, Choleski algorithm, Householder reflections and Givens rotations algorithms, etc.), since a single transient fault in processor or processor elements (PE) of an array during computation might cause multiple output errors, which can not be located.

Therefore, in this manuscript, we improve Abraham's WCS method and to extend it for the LU- LL^T - and QR- decomposition, linear systems solution and matrix inversion, i.e. to design ABFT modifications (versions) of above mention algorithms.

1.3. Requirements to methods of the application-specific architectures designing. Analysis of the known mapping methods for searching of most suitable for the designing of fault-tolerant VLSI parallel processors

At present time, there exists the hierarchical procedure of the digital systems design which consists of the following more or less formalized stage: structural (system), logical, technological and software design [2,12]. Several stage are yet realized on the base of corresponding CAD systems. The structural design stage is the least investigated one. The input data of this stage are the set of functions f_i which system must perform, and structural, timing and others parameters. The result of this stage is the architecture of the target system, i.e. system structure (in the functional blocks level) and computational algorithms for realization of the each function f_i . In according to the functional-structural designing approach, this stage corresponds to the transformation (mapping) of the mathematical description of applied algorithm into the architecture of the target system. Therefore, the structural designing methods based on this approach are named as the mapping methods.

The mapping methods should be satisfied to following requirements:

1. The applied algorithm should be represented by the mathematical description (for example, by equations, recurrence expressions, or nested loops);
2. Simple and correctional computer form of algorithms and structures representation;
3. Guaranty of deriving the correctional architecture designs, i.e. both system structure and execution algorithms with the synchronization function of computations.
4. Possibility of deriving the fault-tolerant architectures.

At present time, the several mapping methods there exist [26, 27, 28, 31-33]. They differ by level of formalization and feasible for the CAD implementation, the classes of suitable algorithms, and :

- forms of applied algorithm representation;
- approaches to determination of the data dependencies of the algorithm;
- searching of mapping functions;
- approaches to verification of the derived solutions.

The known mapping methods, as a rule, are based on the representation of applied regular algorithm expressed by systems of recursive equations or nested loops by its dependence graph (may be indirectly). Each node of such a DG corresponds to a certain operator (or iteration) of the original algorithm, and is associated with the integer vector $K = (k_1,...,k_n)$. All its nodes are located in the vertices K of a lattice $K^n \subset Z^n$, where K^n is called the index space. If the iteration corresponding to a node K_2 depends on the iteration corresponding to another node K_1, this dependence is represented by the dependence vector $d = K_2 - K_1$.

In the course of mapping, a given algorithm AL with the dependence graph G is transformed into a set of structural schemes $C = <S,T,\Phi>$ of arrays architectures implementing this algorithm, where S is a directed graph called the array structure, T is the synchronization function specifying the computation time of nodes in the DG, and Φ is the set of operation algorithms of PEs.

The analysis of the known mapping methods showed that:

1. The dependence graph DG of the algorithm is most widely used form of the algorithm representation. However, this form is suitable only for regular algorithms. Moreover, for deriving of the algorithm DG it is needed to use the special methods which are complicated, few feasible for the CAD implementation and can operate with only uniform recursive algorithms. They are not suitable , for example, to the non-uniform recursive algorithms corresponding to non-perfect (or composite) loop nests. Note, that most of LA algorithms are non-uniform recursive ones. Therefore, the creating the method of DGs construction for algorithms given by both perfect and composite nested loops with regular and quasi-regular data dependencies is the actual problem.
2. All known mapping methods operate only with unconditional regular algorithms which can be represented by regular or quasi-regular directed graphs (or can be reduced to such algorithms).
3. All known methods are formalized and not whole automatic ones, and therefore, needs to user intervention. Therefore, the CAD implementation of mapping method must be interactive one.

4. Using the existing mapping methods, for example, [3, 28, 31-33], the set of array architectures for implementing algorithms with regular data dependencies can be derived. These architectures is directly determined by input algorithm DG properties, and not always effective ones. Therefore, in a order to deriving the array architecture with desired features, the purposive transformations of the basic algorithm DG may be performed before space-time mapping of graph into array architecture. These transformations can be isomorphic and/or geomorphic ones and correspond to reducing the original algorithm to the VLSI algorithm (i.e. suitable to the parallel implementation in VLSI). The examples of several such DGs transformations are represented in the chapter 3.
5. All known mapping methods are not allow to obtain the fault-tolerant solutions. However, in the case when ABFT methods are used, the increasing of fault tolerance of target system is derived automatically, due to mapping of the fault tolerant version of the origin algorithm. Hence, arbitrary mapping method can be used for deriving of the tolerant to transient faults architecture without using any others fault tolerance methods. Therefore, in following chapters the mapping method [31] will be used.

Modern application specific system (for example, signal processing systems) composes both digital and analog parts, where first part is the specialized parallel processor while last part is the interface unit between digital parts and external world. Advances of the modern VLSI technology permit to implement such mixed systems on a single die. Single-chip integration offers many advantages such as size and cost reduction, greater reliability and yields enhancement coupled with an improvement in high-frequency performance thanks to the reduced package interconnection parasitics. Moreover, the programmable (reconfigurable) analog circuits there exist (for example field programmable analog array - FPAA). This enables to design and to implement on a common semiconductor substrate the whole programmable specialized mixed analog-digital system. However, the problem of influence of digital part to the analog part of the mixed system must be solved during system designing. The main problem is the substrate interferences. Switching transients (noise) of the digital part can perturb its analog part of a system by means of coupling through the substrate [97].

There are several known solutions for substrate interference reduction - the use of physical separation of analog and digital circuits , guard rings, and a low inductance substrate bias [92-98]. Each of these methods has own advantages and drawbacks. For example, remedial approaches used to diminishing of substrate cross-talk are signal frequency dependent, e.g. SOI-based process provides high isolation from cross-talk at low operating frequencies, while guard rings appear as the technique which is better suited for preventing cross-talk at high operating frequencies. Another alternative approach for minimizing substrate cross-talk, is a design of interference-resistant analog circuits together with low-level interference-generating digital circuits [88-94]. One of methods for the last approach realization is based on the implementation of the mixed system digital part with the current mode gates [88-91]. Due to the nearly constant value of power supply current at the different gate states (for example, logical"0" and „1"), the level of its noise is essentially lower in comparison with the classical voltage type gates. Besides, based on the current-mode gates, digital circuits with lower hardware overheads may be designed (see, for example, [82-85]). However, the logical properties of the current-mode logic, as well as the rules of current-mode digital circuit design and analysis are not derived. Therefore in this manuscript, the problem of digital circuits design with the current-mode gates will be investigated.

1.4. Conclusions of the chapter 1.

1. The common properties of the most LA methods are a high computational complexity and regularity. Moreover, the order of the computations is independent from the values of the input data. Therefore, most LA algorithms may be effectively solved by different kinds parallel computers.

2. Based on the determined requirements to algorithms and application-specific architectures for their realization in VLSI, the properties of the VLSI algorithms were derived, and was be shown, that the regularity and locality of data dependencies are more important criteria's of the VLSI algorithm, instead its computational complexity. Moreover, it was be shown, that reconfigurable computers are most suitable for the VLSI

parallel system architectures destined for the further realization as FPGA-based or Raw-processors devices.

3. The known fault tolerance methods which use hardware or time redundancy, essentially increase the cost or degrade the performance of computational systems. Therefore, they are few suitable for real-time computing systems and parallel processors. The algorithm-based fault tolerance (ABFT) methods are more suitable for such systems.

4. The original WCS method don't suitable for most LA algorithms (for example, such as Gaussian elimination, Choleski algorithm, Householder reflections and Givens rotations algorithms, etc.), since a single transient fault in processor or processor elements (PE) of an array during computation might cause multiple output errors, which can not be located. Therefore, the important problem is improving Abraham's WCS method for designing of the fault tolerant versions of main LA algorithms.

5. The analysis of the known mapping methods showed that:

- The dependence graph DG of the algorithm is most widely used form of the algorithm representation. However, for deriving of the algorithm DG it is needed to use the special methods which are complicated, few feasible for the CAD implementation and can operate with only uniform recursive algorithms. They are not suitable , for example, to the non-uniform recursive algorithms corresponding to non-perfect (or composite) loop nests. Note, that most of LA algorithms are non-uniform recursive ones. Therefore, the creating the method of DGs construction for algorithms given by both perfect and composite nested loops with regular and quasi-regular data dependencies is the actual problem.
- Using the existing mapping methods, for example, [3, 28, 31-33], the set of array architectures for implementing algorithms with regular data dependencies can be derived. These architectures is directly determined by input algorithm DG properties, and not always effective ones. Therefore, in a order to deriving the array architecture with desired features, the purposive transformations of the basic algorithm DG may be performed before space-time mapping of graph into array architecture. These transformations can be isomorphic and/or geomorphic ones and correspond to reducing the original algorithm to the VLSI algorithm (i.e. suitable to the parallel implementation in VLSI).

- All known mapping methods are not allow to obtain the fault-tolerant solutions. However, in the case when ABFT methods are used, the increasing of fault tolerance of target system is derived automatically, due to mapping of the fault tolerant version of the origin algorithm. Hence, arbitrary mapping method can be used for deriving of the tolerant to transient faults architecture without using any others fault tolerance methods.

6. Advances of the modern VLSI technology permit to implement on a common semiconductor substrate the whole programmable specialized mixed analog-digital system. However, the problem of influence of digital part to the analog part of the mixed system must be solved during system designing. The main problem is the reducing of switching noise of the digital part which can perturb analog part of a system by means of coupling through the substrate.

7. The best approach for minimizing of switching noise is based on the implementation of the mixed system digital part with the current mode gates. Due to the nearly constant value of power supply current at the different gate states (for example, logical”0” and „1”), the level of its noise is essentially lower in comparison with the classical voltage type gates. Besides, based on the current-mode gates, digital circuits with lower hardware overheads may be designed. However, the logical properties of the current-mode logic, as well as the rules of current-mode digital circuit design and analysis are not derived. Therefore, the problem of digital circuits design with the current-mode gates should be investigated.

CHAPTER 2. MODIFICATION OF THE WEIGHTED CHECKSUM METHOD AND DERIVING OF THE FAULT TOLERANT VERSIONS OF MAIN LA ALGORITHMS

2.1. Weighted checksums (WCS) method and its modification for deriving of the fault tolerant versions of main LA algorithms

The WCS code has been adopted by Jou and Abraham [59]in matrix arithmetic operations for algorithm-based fault tolerance. The idea is to compress the information contained in the row/column elements of matrix into a single element which named a check element. Information is compressed in such a way that it is preserved during algorithm implementation. In their scheme, redundancy is encoded at the matrix level by augmenting the original matrix with weighted checksums. Since the checksum property is preserved for various matrix operations, these checksums are able to detect and correct errors in the resultant matrix. Furthermore, the complexity of correction process is much smaller than that of the original computation. For example, a WCS encoded data vector **v**(N) with Hamming distance equal three which can correct a single error (SEC) can be expressed as

$$\mathbf{V}^{\mathbf{T}} = \left[\mathbf{v}_1 \quad \mathbf{v}_2 \quad \ldots \quad \mathbf{v}_N \quad \mathbf{PCS} \quad \mathbf{QCS}\right], \tag{2.1}$$

where $\mathbf{v}_i$ is a element of a data vector **v**(N),

$$\begin{aligned} \mathbf{PCS} &= \mathbf{p}^{\mathbf{T}} \times \left[\mathbf{v}_1 \quad \mathbf{v}_2 \quad \ldots \quad \mathbf{v}_N\right] \\ \mathbf{QCS} &= \mathbf{q}^{\mathbf{T}} \times \left[\mathbf{v}_1 \quad \mathbf{v}_2 \quad \ldots \quad \mathbf{v}_N\right], \end{aligned} \tag{2.2}$$

and **p**(N), **q**(N) - are encoder vectors.

Possible choices for vector pares **p** and **q** are, for example, [59]

$$\mathbf{p}^{\mathbf{T}} = \left[\mathbf{1} \quad \mathbf{1} \quad \ldots \quad \mathbf{1}\right] \quad \text{and} \quad \mathbf{q}^{\mathbf{T}} = \left[2^0 \quad \mathbf{2}^1 \quad \ldots \quad 2^{N-1}\right] \tag{2.3}$$

(were **q** is named the exponential weighted encoder vector), or [62]

$$\mathbf{p}^{\mathbf{T}} = [\mathbf{1} \quad \mathbf{1} \quad \ldots \quad \mathbf{1}] \quad \text{and} \quad \mathbf{q}^{\mathbf{T}} = [\mathbf{1} \quad \mathbf{2} \quad \ldots \quad \mathbf{N}] \tag{2.4}$$

(were **q** is named the linear weighted encoder vector).

The difficulty with the first choice is loss of the numerical accuracy due to large weights, while the second choice leads to larger extra computations necessary to correct an error.

Based on the, for example, linear encoding vector (2.4), a matrix **A**(M,N) can be encoded as either a row encoded matrix $\mathbf{A_R}$ given by

$$\mathbf{A_R} = [\mathbf{A} \quad \mathbf{A} \times \mathbf{p}(N) \quad \mathbf{A} \times \mathbf{q}(N)] = [\mathbf{A} \quad \mathbf{PRS} \quad \mathbf{QRS}], \tag{2.5}$$

where

$$\begin{aligned} \mathbf{PRS_i} &= \mathbf{a_{i1}} + \mathbf{a_{i2}} + \ldots + \mathbf{a_{iN}} \\ \mathbf{QRS_i} &= \mathbf{1} \times \mathbf{a_{i1}} + \mathbf{2} \times \mathbf{a_{i2}} + \ldots + \mathbf{N} \times \mathbf{a_{iN}} \end{aligned},$$

a column encoded matrix $\mathbf{A_C}$

$$\mathbf{A}_C = \begin{bmatrix} \mathbf{A} \\ \mathbf{p}^{\mathbf{T}}(M) \times \mathbf{A} \\ \mathbf{q}^{\mathbf{T}}(M) \times \mathbf{A} \end{bmatrix} = \begin{bmatrix} \mathbf{A} \\ \mathbf{PCS} \\ \mathbf{QCS} \end{bmatrix}, \tag{2.6}$$

where

$$\begin{aligned} \mathbf{PCS_j} &= \mathbf{a_{1j}} + \mathbf{a_{2j}} + \ldots + \mathbf{a}_{Mj} \\ \mathbf{QCS_j} &= \mathbf{1} \times \mathbf{a_{1j}} + \mathbf{2} \times \mathbf{a_{2j}} + \ldots + \mathbf{M} \times \mathbf{a}_{Mj} \end{aligned},$$

or a full encoded matrix $\mathbf{A_{RC}}$ [60, 62] given by

$$\mathbf{A_{RC}} = \begin{bmatrix} \mathbf{A} & \mathbf{PRS} & \mathbf{QRS} \\ \mathbf{PCS} & \mathbf{1} & \mathbf{0} \\ \mathbf{QCS} & \mathbf{0} & \mathbf{1} \end{bmatrix}. \tag{2.7}$$

For example, for matrix multiplication **A**(M,N) * **B**(N,K) = **C**(M,K) , the column encoded matrix $\mathbf{A}_C$ of form (2.6) is exploited [62]. Then , the following expression is computed:

$$\boldsymbol{A}_C \times \boldsymbol{B} = \boldsymbol{C}_C . \tag{2.8}$$

To verify the computation, syndromes S_1 and S_2 for the j-th column of matrix **C** should be calculated *(j = 1, ... , K)* :

$$\mathbf{S_1} = \sum_{\mathbf{i=1}}^{M} \mathbf{c_{ij}} - \mathbf{PCS_j} \qquad (2.9)$$

and

$$\mathbf{S_2} = \sum_{\mathbf{i=1}}^{M} \mathbf{i} \times \mathbf{c_{ij}} - \mathbf{QCS_j} \qquad (2.10)$$

In order to correct a single error, the following procedure is used:

if $\mathbf{S_1 = S_2 = 0}$ then no error has been detected;

if $\mathbf{S_1 \neq 0}$ and $\mathbf{S_2 = 0}$ then $\mathbf{PCS_j}$ is inconsistent;

if $\mathbf{S_2 \neq 0}$ and $\mathbf{S_1 = 0}$ then $\mathbf{QCS_j}$ is inconsistent;

if $\mathbf{S_1 \neq 0}$ and $\mathbf{S_2 \neq 0}$ then $\mathbf{S_2 / S_1 = i}$ and element $\boldsymbol{c_{ij}}$ is erroneous,

and the correction procedure is:

$$\boldsymbol{c_{ij} = c_{ij} - S_1} . \qquad (2.11)$$

Thus, the computational complexity of the modified version of the matrix multiplication algorithm increases only on the $2N^2$ operations and is equal to $(N^3 + 2N^2)$ multiply-add operations. This version allows to detect and to correct the single error among elements of each column of an input matrix A(M,N) occurred during algorithm implementation. Consequently, it is possible to correct up to N single errors during solving the whole task.

However, the original WCS method is not suitable for most LA algorithms (for example, such as Gaussian elimination, Choleski algorithm, Householder reflections and Givens rotations algorithms, etc.), since a single transient fault in processor or processor elements (PE) of an array during computation might cause multiple output errors, which can not be located. In fact, the common property of all above mentioned algorithms is the computation on the any i-th algorithm step (may be not one times) the elements of leading (i-th) row or/and column of matrix $\mathbf{A^i} = \{a_{ji}^{\mathbf{i}}\}$ and then modification others matrix rows (columns) by means leading ones. The example of corresponding fragment of such algorithms with leading column computations is represented by means construction (2.12), were values of variables *K, K1, K2* and functions *f1*, *f2* are depended from

selected algorithm. Note, that in this example the input matrix $\mathbf{A} = \mathbf{A^1} = \{a_{ji}\}$ is recursively modified during K computation steps to obtain the resulting matrix $\mathbf{A^{K+1}}$.

for *i=1* **to** *K* **do**

begin

{Phase1:computation of the leading column elements a_{ji}^{i+1} *}*

for *j=i+1* **to** *K1* **do**

$a_{ji}^{i+1} = f1(a_{ji}^{i}, a_{ii}^{i});$ (2.12)

{Phase2: computation of the elements of the matrix $\mathbf{A^{i+1}}$ *}*

for *j=i+1* **to** *K1* **do**

for *k=i+1* **to** *K2* **do**

$a_{jk}^{i+1} = f2(a_{jk}^{i}, a_{ji}^{i+1}, a_{ik}^{i});$

end

As shown from the (2.12), if at the i-th algorithm step the element a_{ji}^{i+1} of leading (i-th) column is wrongly calculated, then errors will appear in all elements a_{jk}^{i+1} of j-th row of $\mathbf{A}^{i+1}$. Analogously, if any element a_{ik}^{i} of the leading (i-th) row was wrongly calculated, then errors appear in the all elements of *j*-th column of $\mathbf{A}^{i+1}$. In both cases, these errors can not be corrected by WCS method. If the correction of elements a_{jk} is performed during calculations, then the computational complexity of the original algorithm increases more than twice.

For removing of this defects by means a modification of the original WCS method, the following confirmations were proved for each above mentioned algorithm (see sections 2.2, 2.3 of this manuscript):

- - If during i-th step of computations the element a_{jk}^{i+1} is wrongly calculated, then errors will not appear among others elements of matrix $\mathbf{A}^{i+1}$, while j-th row will not become the leading one (i.e. while i=j).
- - If the element a_{jk}^{i} (j=i,i+1,...,N) was wrongly calculated several times q (q<i) before performing of the i-th step of algorithm (2.12), then it is possible to correct its using the WCS method for the row encoded matrix A_R (2.5) at the beginning of the i-th step of the algorithm.

- - If an element a_{jk}^{i} (j=i,i+1,...,N) was wrongly calculated during executing of the first phase at the i -th step of algorithm (2.11), then it is possible to correct its using the WCS method for the column encoded matrix A_C (2.6) after executing of this phase.

The main consequence of these confirmations is the possibility to perform the detection and correction procedures during each i-th algorithm step among only elements of the leading (i-th) row and leading (i-th) column of matrix $\mathbf{A}^i$. Based on these confirmations, the modifica-tion of the origin WCS method was performed. The main idea of the proposed unified WCS method (scheme) destined for main linear algebra algorithms is the performing of check procedures concurrently with algorithm computations or more exactly, the performing of the detection and correction procedures:

- - at each i-th algorithm step;
- - among only elements of the leading (i-th) row and leading (i-th) column of the matrix $\mathbf{A}^i$.

Note, that proposed modified method may be used for designing the fault-tolerant version of an arbitrary matrix algorithm for which above mentioned confirmations are corrected. Therefore, these confirmation may be considered as sufficient conditions for using of the modified WCS method.

As a result, the proposed checksums scheme increases the computational complexity of original algorithm 2.12 on $O(N^2)$ operations (such as multiply-add operations). Consequently, proposed modification of WCS-method do not increase its computational complexity. However, using of the proposed uniform scheme enables to correct one single error among elements of an arbitrary column (or row) of an input matrix A(M,N) on any from K steps of algorithm implementation. Consequently, it is possible to correct up to K (where K=(N-1) for case M=N) single errors during solving the whole task.

Therefore, in next sections of this manuscript, we will try to use proposed modified WCS method to the designing of the fault tolerant versions of LU-, LL^T - decomposition, linear systems solution and some other linear algebra algorithms.

2.2. Designing of the fault tolerant versions of the Gauss elimination and LU-decomposition algorithms

Matrix triangular decomposition is one of the most important problems in linear algebra. By triangularisation many problems are reduced to simpler problems with triangular matrices. Due to own computational complexity (equal to $N^3/3$ multiply-add operations), Gauss elimination algorithm

$$\mathbf{M}_{(N,N)} \cdot \mathbf{A}_{(N,N)} = \mathbf{U}_{(N,N)} \qquad (2.15)$$

(or LU - decomposition $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$) with partial pivoting is most suitable for the triangularisation of non-symmetric matrix $\mathbf{A}_{(N,N)}$.

At first, the design of fault tolerant version of Gaussian elimination algorithm without pivoting will be represented. This algorithm may be written by construction (2.16). Note that the input matrix $\mathbf{A} = \mathbf{A^1} = \{a_{ji}\}$ is recursively modified during *(N-1)* computation steps to obtain the upper triangular matrix $\mathbf{U} = \mathbf{A^N}$. Note, that at the i-th algorithm step (i=1,...,N-1) the i-th row of matrix $\mathbf{A^i}$ is the pivoting (or leading) row.

$a_{jk}^{1} = a_{jk}$, $m_{kk}=1$, $j=1,2,...,N$, $k=1,2,...,N$.
for $i=1$ **to** $N-1$ **do**
begin
for $j=i+1$ **to** N **do**
if $a_{ii}^{i} <> 0$ **then** $m_{ji} = a_{ji}^{i} / a_{ii}^{i}$
else $m_{ji} = 0;$ (2.16)
for $j=i+1$ **to** N **do**
for $k=i+1$ **to** N **do**
$a_{jk}^{i+1} = a_{jk}^{i} - m_{ji} \cdot a_{ik}^{i}$;
end

where m_{ji} is a data element of the lower triangular matrix **M**.

It is followed from the construction (3.16), that if during computation the element m_{ji} (or l_{ji} for LU-decomposition $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$) is wrongly calculated, then errors will appear

in all elements a_{jk}^{i+1} of j-th row of $\mathbf{A}^{i+1}$. Moreover, if any element a_{ik}^{i} of the leading row is wrongly calculated, then errors appear in all elements of *k*-th column of $\mathbf{A}^{i+1}$. In both cases, these errors can not be corrected by the original WCS-method. Therefore, in a order to deriving of a fault tolerant version of this algorithm, the proposed modified WCS method must be used. However, the truth of the represented in the section 2.1 confirmation must be proved.

For the algorithm (2.16) these confirmations are transformed in the theorems 2.1, 2.2 and 2.3 respectively.

THEOREM 2.1. If during the i-th step of computations the element a_{jk}^{i+1} is wrongly calculated, then errors will not appear among others elements of matrix $\mathbf{A}^{i+1}$, until the j-th row becomes the pivoting one.

The proof of this theorem directly follows from the algorithm (2.16), where each element a_{jk}^{i} takes part in calculations only elements a_{jk}^{i+1}, a_{jk}^{i+2},..., a_{jk}^{i+p}, where $(i+p) \le j$ and $(i+p) \le k$.

□

THEOREM 2.2. Let the element a_{jk}^{i} (j=i, i+1,..., N) was wrongly calculated q times (q<i) before executing the i-th step of algorithm (2.16). Then it is possible to correct its value only once, using WCS method for row encoded matrix $\mathbf{A}_r$, at the beginning of the i-th step of the algorithm (2.16).

PROOF. Without the loss of a generality, we assume that i<j, i<k and q=2 for element a_{jk}^{i}. Let the element a_{jk}^{i} was wrongly calculated at the (i-1)-th step of algorithm (16). Then its value will be equal to

$$a_{jk}^{iz} = a_{jk}^{i} + z_{jk}^{i},$$

where z_{jk}^{i} - is the calculation error. Then, in accordance with (2.16), after the next step we obtain:

$$a_{jk}^{i+1} = a_{jk}^{i} - m_{ji} \cdot a_{ik}^{i} = a_{jk}^{iz} - z_{jk}^{i} - m_{ji} \cdot a_{ik}^{i} \qquad (2.17)$$

We assume now that expression (2.17) was also wrongly calculated. In similar way, we obtain that the true value of the element a_{jk}^{i+1} will be equal to

$$a_{jk}^{i+1} = a_{jk}^{(i+1)z} - z_{jk}^{i+1} = a_{jk}^{iz} - z_{jk}^{i} - z_{jk}^{i+1} - m_{ji} \cdot a_{ik}^{i} = a_{jk}^{iz} - z_{jk} - m_{ji} \cdot a_{ik}^{i}, \qquad (2.18)$$

where $z_{jk} = z_{jk}^{i} + z_{jk}^{i+1}$.

Thus, the computation errors of the element a_{jk} are accumulated in the variable z_{jk}. Consequently, the wrongly calculated element a_{jk} may be corrected only at the j-th step of the algorithm (2.16), i.e. when the j-th row will become the leading row (j=i).

□

CONSEQUENCE. It is possible during algorithm computation to correct only elements of the leading (pivoting) row of the matrix A using WCS method for row encoded matrix $\mathbf{A}_R$ (2.5).

Now, we should derive the method for detection and correction of erroneous elements m_{ji} of the matrix M. In order to this, we prove the following theorem.

THEOREM 2.3. The values of checksum CS_i and weighted checksum WCS_i of the i-th column of the matrix M are respectively equal to the values of checksum $PCS_i^{(i+1)}$ and weighted checksum $QCS_i^{(i+1)}$ of the i-th column of matrix $A^{(i+1)}$ (i.e. the values of checksum and weighted checksum of i-th column of matrix A_C after performing of the i-th step of the algorithm (2.16)).

PROOF. At the beginning of the i-th step of algorithm (2.16) the values PCS_i^i and QCS_i^i of the matrix A_C in accordance with (2.6) are equal to the following expressions:

$$PCS_i^i = a_{ii}^i + a_{(i+1)i}^i + \ldots + a_{Ni}^i$$

$$\text{and} \qquad QCS_i^i = i \cdot a_{ii}^i + (i+1) \cdot a_{(i+1)i}^i + \ldots + N \cdot a_{Ni}^i \qquad (2.20)$$

respectively.

After performing of the i-th step of the algorithm 2.16 with the column encoded matrix A_C , these values will be equal to

$$PCS_i^{(i+1)} = PCS_i^i / a_{ii}^i \qquad \text{and} \qquad QCS_i^{(i+1)} = QCS_i^i / (i \cdot a_{ii}^i) . \quad (2.21)$$

In other side, the values of checksum CS_i and weighted checksum WCS_i of the i-th column of the matrix M in accordance to the expression (2.6) and algorithm (2.16) are equal to the following expressions:

$$CS_i = 1 + m_{(i+1)i} + m_{(i+2)i} + ... + m_{Ni} = 1 + a_{(i+1)i}^{i} / a_{ii}^{i} + ... + a_{Nk}^{i} / a_{ii}^{i} = PCS_i^i / a_{ii}^i$$

and $$WCS_i = 1 + (i+1)\, m_{(i+1)i} + (i+2) \cdot m_{(i+2)i} + ... + N \cdot m_{Ni} =$$

$$= 1 + (i+1) \cdot a_{(i+1)i}^{i} / (i \cdot a_{ii}^{i}) + ... + N \cdot a_{Nk}^{i} / /(i \cdot a_{ii}^{i}) = WCS_i^i / (i \cdot a_{ii}^i). \quad (2.22)$$

□

Thus, the truth of the represented in the section 2.1 confirmations (which were transformed in the theorems 2.1, 2.2 and 2.3) is proved. Therefore, in a order to deriving of a fault tolerant version of this algorithm, the proposed modified WCS method checksum scheme may be used.

However, we should be certain that the elements of i-th column of matrix A^i were calculated correctly at the (i-1) step of the algorithm (2.16). It is proved below, that it may be verified using WCS-method for i-th column of matrix A^i_C analogously to (2.9) - (2.11).

In accordance with (2.6) and (2.16), at the (i-1)-th algorithm step the value of checksum $PCS_i^{(i-1)}$ of the i-th column of the matrix A is equal to the following expression:

$$PCS_i^{(i-1)} = a_{(i-1)i}^{(i-1)} + a_{ii}^{(i-1)} + ... + a_{Ni}^{(i-1)}. \quad (2.23)$$

In other side, the values PCS_i^i calculated in accordance to the algorithm (2.16) will be equal to the following expression:

$$PCS_i^i = PCS_i^{(i-1)} - a_{(i-1)i}^{(i-1)} \cdot PCS_{(i-1)}^{(i-1)} / a_{(i-1)(i-1)}^{(i-1)} = (a_{(i-1)i}^{(i-1)} + a_{ii}^{(i-1)} + ... + a_{Ni}^{(i-1)}) -$$

$$- a_{(i-1)i}^{(i-1)} / a_{(i-1)(i-1)}^{(i-1)} \cdot (a_{(i-1)(i-1)}^{(i-1)} + a_{i(i-1)}^{(i-1)} + ... + a_{N(i-1)i}^{(i-1)}) = (a_{ii}^{(i-1)} + a_{(i+1)i}^{(i-1)} +$$

$$+ ... + a_{Ni}^{(i-1)}) - a_{(i-1)i}^{(i-1)} / a_{(i-1)(i-1)}^{(i-1)} \cdot (a_{i(i-1)}^{(i-1)} + a_{(i+1)(i-1)}^{(i-1)} + ... + a_{N(i-1)}^{(i-1)}).$$

At the same time, the value PCS_i^i should be equal to the following expression:

$$PCS_i^i = a_{ii}^i + a_{(i+1)i}^i + ... + a_{Ni}^i = (a_{ii}^{(i-1)} - a_{(i-1)i}^{(i-1)} \cdot a_{i(i-1)}^{(i-1)} / a_{(i-1)(i-1)}^{(i-1)}) + (a_{(i+1)i}^{(i-1)} -$$

$$- a_{(i-1)i}^{(i-1)} \cdot a_{(i+1)(i-1)}^{(i-1)} / a_{(i-1)(i-1)}^{(i-1)}) + ... + (a_{Ni}^{(i-1)} - a_{(i-1)i}^{(i-1)} \cdot a_{N(i-1)}^{(i-1)} / a_{(i-1)(i-1)}^{(i-1)}) =$$

$$= (a_{ii}^{(i-1)} + a_{(i+1)i}^{(i-1)} + ... + a_{Ni}^{(i-1)}) - a_{(i-1)i}^{(i-1)} / a_{(i-1)(i-1)}^{(i-1)} \cdot (a_{i(i-1)}^{(i-1)} + a_{(i+1)(i-1)}^{(i-1)} + ... + a_{N(i-1)}^{(i-1)}).$$

Note, that the truth of this confirmation for the variable QCS_i^i is proved analogously. Consequently, it is possible to correct only elements of the i-th column of matrix A^i at the beginning of the i-th step of the algorithm (2.16) using the WCS method checking procedures for the column encoded matrix $\mathbf{A_C}$.

Finally, the fault tolerant version of Gauss elimination algorithm without pivoting will consist of performing the following stages:

1. The original matrix **A** is represented in the form of the full encoded matrix $\mathbf{A_{RC}}$ (see expression 2.7).

$$\mathbf{A} = \begin{bmatrix} \mathbf{a_{11}} & \mathbf{a_{12}} & \cdots & \mathbf{a_{1N}} & \mathbf{PRS_1} & \mathbf{QRS_1} \\ \mathbf{a_{21}} & \mathbf{a_{22}} & \cdots & \mathbf{a_{2N}} & \mathbf{PRS_2} & \mathbf{QRS_2} \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ \mathbf{a_{N1}} & \mathbf{a_{N2}} & \cdots & \mathbf{a_{NN}} & \mathbf{PRS_N} & \mathbf{QRS_N} \\ \mathbf{PCS_1} & \mathbf{PCS_2} & \cdots & \mathbf{PCS_N} & \mathbf{1} & \mathbf{0} \\ \mathbf{QCS_1} & \mathbf{QCS_2} & \cdots & \mathbf{QCS_N} & \mathbf{0} & \mathbf{1} \end{bmatrix},$$

where, in the case of using the linear encoded vector (2.4), the values of the checksums and weighted checksums are represented by following expressions:

$$\begin{aligned} \mathbf{a[i, N+1]} &= \mathbf{PRS_i} = \mathbf{a_{i1} + a_{i2} + \ldots + a_{iN}}, \\ \mathbf{a[i, N+2]} &= \mathbf{QCS_i} = \mathbf{1 \times a_{i1} + 2 \times a_{i2} + \ldots + N \times a_{iN}}, \\ \mathbf{a[N+1, j]} &= \mathbf{PCS_j} = \mathbf{a_{1j} + a_{2j} + \ldots + a_{Nj}} \\ \mathbf{a[N+2, j]} &= \mathbf{QCS_j} = \mathbf{1 \times a_{1j} + 2 \times a_{2j} + \ldots + N \times a_{Nj}} \end{aligned} \qquad (2.24)$$

2. For *i=1, 2, ..., N-1* , stages 3-6 are repeated.

3. At the beginning of the *i*-th step of the algorithm the detection and correction within elements belonging to the *i*-th row and the *i*-th column of $\mathbf{A^i_{RC}}$ matrix are performing in according to the expressions (2.20) and (2.9)-(2.11).

This stage needs to perform 2(N-i) multiply-add operations and 2(N-i) additions.

4. The elements m_{ji} are calculated.

5. The detection and the correction of the elements m_{ji} are performed in according to the following expressions (2.22) and (2.9)-(2.11).

This stage requires (N-i) multiplications with additions and (N-i) additions.

6. The elements of matrix $A_{RC}^{(i+1)}$ are calculated.

In the Pascal-like form the fault tolerant version of Gauss elimination algorithm may be represented by the following construction, were δ is the small machine depended constant (roundoff value):

for *i := 1* **to** *N-1* **do**

begin

{ Errors detection and correction within elements of the i-th column of A^i }

$PCS_i := 0;\ QCS_i := 0;$

for $j := i$ **to** N **do begin** $PCS_i := PCS_i + a_{ji}\ ;\ QCS_i := QCS_i + j^* a_{ji}$; **end;**

$S1 := PCS_i - a_{N+1i}\ ;\ \ S2 := QCS_i - a_{N+2i}\ ;$

if abs $(S1) > \delta$ **and abs** $(S2) < \delta$ **then** $a_{N+1i} := PCS_i\ ;$

if abs $(S2) > \delta$ **and abs** $(S1) < \delta$ **then** $a_{N+2i} := QCS_i\ ;$

if abs $(S1) > \delta$ **and abs** $(S2) > \delta$ **then begin** $j := S2/S1\ ;\ a_{ji} := a_{ji} - S1;$ **end;**

{ Errors detection and correction within elements of the i-th row of A^i }

$PRS_i := 0;\ QRS_i := 0;$

for $k := i$ **to** N **do begin** $PRS_i := PRS_i + a_{ji}\ ;\ QRS_i := QRS_i + k^* a_{ji}$; **end;**

$S1 := PRS_i - a_{i,N+1}\ ;\ \ S2 := QRS_i - a_{i,N+2}\ ;$

if abs $(S1) > \delta$ **and abs** $(S2) < \delta$ **then** $a_{i,N+1} := PRS_i\ ;$

if abs $(S2) > \delta$ **and abs** $(S1) < \delta$ **then** $a_{i.N+2} := QRS_i\ ;$

if abs $(S1) > \delta$ **and abs** $(S2) > \delta$ **then begin** $k := S2/S1\ ;\ a_{ik} := a_{ik} - S1;$ **end;**

{Computation of the elements of the matrix M}

for $j := i + 1$ **to** $N+2$ **do**

$m_{ji} := a_{ji} / a_{ii}\ ;$

{ Errors detection and correction within elements of the i-th column of matrix M }

$CS_i := 1;\ WCS_i := j;$

for $j := i+1$ **to** N **do begin** $CS_i := CS_i + m_{ji}\ ;\ WCS_i := WCS_i + j^* m_{ji}$; **end;** (2.25)

S1 := CS$_i$ - a$_{N+1,i}$; S2 := WCS$_i$ - a$_{N+2,i}$;

if abs *(S1) > δ* **and abs** *(S2) < δ* **then** *a$_{N+1i}$:= CS$_i$;*

if abs *(S2) > δ* **and abs** *(S1) < δ* **then** *a$_{N+2i}$:= WCS$_i$;*

if abs *(S1) > δ* **and abs** *(S2) > δ* **then begin** *j:= S2 /S1 ; a$_{ji}$:= a$_{ji}$ - S1;* **end;**

{Elimination of the elements of matrix A}

for *j := i + 1* **to** *N+2* **do**

for *k := i + 1* **to** *N+2* **do**

*a$_{jk}$:= a$_{jk}$ - m$_{ji}$ * a$_{ik}$;*

end;

Note, that the linear weighted encoder vector (2.4) is used here for coding of the QRS and OCS values.

REMARK 2.1. Theorems 2.1-2.3 and all stages of the fault tolerant version of the Gauss elimination algorithm without pivoting are also truth for the Gauss algorithm with partial pivoting. However, for partial pivoting algorithm version, at stage 3, the error detection and correction within the elements of i-th column of the matrix A^i_{RC} is performed first. Then the pivoting row is determined, and the detection and the correction procedure for this row is carried out.

REMARK 2.2. Theorems 2.1 - 2.3 and all their consequences are also true for the LU-decomposition algorithm, if the elements m_{ji} will be replaced with corresponding elements l_{ji} . Consequently, the all stages of the fault tolerant version of the LU-decomposition algorithm will be the same ones, as in the Gauss elimination algorithm.

Thus, using modified version of the WCS-method, the fault-tolerance versions of Gauss elimination and LU-decomposition algorithms were designed.

Comparison of the constructions (2.16) and (2.25) shows that inserting the error detection and correction procedures increases the computational complexity of original algorithm (2.16) on the each i-th step on the 3(N-i) operations of multiplication with addition and 3(N-i) additions. This means, that the computational complexity of the whole algorithm increases on $1{,}5 \cdot N^2$ multiply-add operations and $1{,}5 \cdot N^2$ additions. Besides, due to the increasing of input matrix sizes, the computational complexity of the proposed algorithm (2.25) also is increased on $2 \cdot N^2$ multiply-add operations. As a result,

the computational complexity of the whole fault-tolerant algorithm is increased approximately on $3.5 \cdot N^2 + O(N)$ multiply -add operations in comparison with the original algorithm (2.16). However, new algorithms enable to detect and to correct a single error in an arbitrary row or column of the input matrix A at the each algorithm step. Hence, it is possible to correct up to $N^2/2$ single errors during solving the whole decomposition task (2.15).

2.3. Designing of the fault tolerant version of the Choleski algorithm

The Choleski algorithm is used for the triangular decomposition of the positively defined symmetric matrix **A**(N,N) ($\boldsymbol{detA > 0}$) in such a way, that

$$\mathbf{A}_{(N,N)} = \mathbf{L}_{(N,N)} \cdot \mathbf{L}^{\mathbf{T}}_{(N,N)} , \qquad (2.26)$$

where L(N,N) is the lower triangular matrix. It is represented below by the construction (2.27)

```
for i = 1 to N do
 begin
   a_ii^(i+1) = sqrt(a_ii^i) ;
   for j = i+1 to N do
        a_ji^(i+1) = a_ji^i / a_ii^(i+1);
   for j = i+1 to N do                                   (2.27)
    begin
     for k = i+1 to j do
        a_jk^(i+1) = a_jk^i - a_ji^(i+1) × a_ki^(i+1);
   end
 end
```

Note that the input matrix $\mathbf{A} = \mathbf{A^1} = \{a_{ij}\}$ (where $i=1,...,N;\ j=1,...,j$) is recursively modified during N computation steps to obtaining the lower triangular matrix $\mathbf{L} = \mathbf{A^N}$.

Moreover, at the i-th algorithm step ($i=1,...,N$) the i-th row and i-th column of the matrix $\mathbf{A}^{\mathbf{i}}$ are the pivoting (or leading) ones.

It is followed from the construction (2.27), that if during computation step the element a_{ji}^{i+1} is wrongly calculated, then errors will appear in all elements of the j-th row and the j-th column of the matrix $\mathbf{A}^{i+1}$. Note, that these errors can not be corrected by the original WCS-method. Therefore, in a order to deriving of a fault tolerant version of this algorithm, the proposed modified WCS method must be used. However, the truth of the below represented theorems 2.4 - 2.6 (which correspond to the represented in the section 2.1 confirmations) must be proved.

THEOREM 2.4. If during the i-th step of algorithm implementation the element a_{jk}^{i+1} is wrongly calculated, where $j=i+1,..., N$ and $k=i+1,..., j$, then errors will not appear among others elements of matrix $\mathbf{A}^{i+1}$, until the k-th column becomes the pivoting one.

The proof of this theorem directly follows from the algorithm (2.27), in which each element a_{jk}^{i}

takes part in calculations only elements a_{jk}^{i+1}, a_{jk}^{i+2},..., a_{jk}^{i+p}, where $(i+p)\leq k$.

THEOREM 2.5. Let the element a_{jk}^{i} was wrongly calculated q times (q<i) before executing the i-th step of algorithm (2.27). Then it is possible to correct its value only once, using WCS method for column encoded matrix $\mathbf{A}_C$, at the beginning of the k-th step of the algorithm (2.27).

PROOF. The proof of this theorem is similar to that of theorem 2.2.

CONSEQUENCE. It is possible, at the beginning of the i-th step of computations, to correct only elements of the i-th (pivoting) column of the matrix $\mathbf{A}^{i}$ using WCS method for the column encoded matrix $\mathbf{A}^{i}_{C}$ (2.6).

Now, we should derive the scheme for the detection and the correction of erroneous elements belonging to the leading column of the matrix **A**. In order to this, we prove the following theorem.

THEOREM 2.6. The calculated values of the elements belonging to the i-th (pivoting) column of the matrix $\mathbf{A}^{i+1}$ can be verified using the original WCS -method checking procedures for the i-th column of the column encoded matrix $A_C^{(i+1)}$ at the i-th step of the algorithm (2.27)

PROOF. At the beginning of the i-th step of algorithm (2.27) the values of checksums PCS_i^i and QCS_i^i of the i-th column of the matrix A_C in accordance with (2.6) are equal to the following expressions:

$$PCS_i^i = a_{ii}^i + a_{(i+1)i}^i + \ldots + a_{Ni}^i$$
$$\text{and} \qquad QCS_i^i = i \cdot a_{ii}^i + (i+1) \cdot a_{(i+1)i}^i + \ldots + N \cdot a_{Ni}^i \qquad (2.28)$$

respectively.

According to the algorithm (2.27), after performing of the i-th step, with the column encoded matrix A_C , these values will be equal to

$$PCS_i^{(i+1)} = PCS_i^i / a_{ii}^{i+1} = PCS_i^i / \sqrt{a_{ii}^i} \qquad (2.29)$$
$$\text{and} \qquad QCS_i^{(i+1)} = QCS_i^i / (i \cdot \sqrt{a_{ii}^i}) .$$

In other side, the values of checksum PCS_i^{i+1} of the i-th column after performing i-th algorithm step will equal to the following expressions:

$$PCS_i^{i+1} = a_{ii}^{i+1} + a_{(i+1)i}^{i+1} + \ldots + a_{Ni}^{i+1} = \sqrt{a_{ii}^i} + a_{(i+1)i}^i / \sqrt{a_{ii}^i} + \ldots + a_{Ni}^i / \sqrt{a_{ii}^i} =$$
$$= (a_{ii}^i + a_{(i+1)i}^i + \ldots + a_{Ni}^i) / \sqrt{a_{ii}^i} = PCS_i^i / \sqrt{a_{ii}^i}$$

Note, that the truth of this theorem for the values of weighted checksum QCS_i^i is performed in a similar way.

The theorem 2.6 is proved. □

Thus, the truth of the represented in the section 2.1 confirmations (which were transformed in the theorems 2.4, 2.5 and 2.6) is proved. Therefore, in a order to deriving of a fault tolerant version of this algorithm, the proposed modified WCS method checksum scheme may be used.

However, as in the case of the Gauss elimination algorithm, we should be certain that the elements a_{ji}^i of i-th column of matrix A^i were calculated correctly at the previous (i-1)-th step of the algorithm (2.27) (when this column was not a pivoting one). It is proved below, that it may be verified using WCS-method checking procedures for the k-th column of the matrix A^i_C analogously to (2.9) - (2.11).

In accordance with (2.6) and (2.27), after performing of the (i-1)-th algorithm step the value of checksum $PCS_k^{(i-1)}$ of the k-th column of the matrix A^i_C is equal to the following expression:

$$PCS_i^i = a_{(i+1)k}^i + a_{(i+2)k}^i + \ldots + a_{Nk}^i = (a_{(i+1)k}^{(i-1)} - a_{(i+1)i}^i \cdot a_{ki}^i) + (a_{(i+2)k}^{(i-1)} - a_{(i+2)i}^i \cdot a_{ki}^i) +$$
$$+ \ldots + (a_{Nk}^{(i-1)} - a_{Ni}^i \cdot a_{ki}^i) = (a_{(i+1)k}^{(i-1)} + a_{(i+2)k}^{(i-1)} + \ldots + a_{Nk}^{(i-1)}) -$$
$$- (a_{(i+1)i}^{(i-1)} + a_{(i+2)i}^{(i-1)} + \ldots + a_{Ni}^{(i-1)}) \cdot a_{ki}^{(i-1)} / a_{ii}^{(i-1)}.$$

In other side, the values PCS_i^i calculated in accordance to the algorithm (2.20) will be equal to the following expression:

$$PCS_k^i = PCS_k^{(i-1)} - a_{ik}^i \cdot PCS_i^i = PCS_k^{(i-1)} - a_{ik}^{(i-1)} \cdot PCS_i^{(i-1)} / a_{ii}^{(i-1)} = (a_{ik}^{(i-1)} + a_{(i+1)k}^{(i-1)} +$$
$$+ \ldots + a_{Nk}^{(i-1)}) - (a_{ii}^{(i-1)} + a_{(i+1)i}^{(i-1)} + \ldots + a_{Ni}^{(i-1)}) \cdot a_{ki}^{(i-1)} / a_{ii}^{(i-1)} = (a_{(i+1)k}^{(i-1)} + a_{(i+2)k}^{(i-1)} + \ldots +$$
$$+ a_{Nk}^{(i-1)}) - (a_{(i+1)i}^{(i-1)} + a_{(i+2)i}^{(i-1)} + \ldots + a_{Ni}^{(i-1)}) \cdot a_{ki}^{(i-1)} / a_{ii}^{(i-1)}.$$

Note, that the truth of this confirmation for the variable QCS_i^i is proved analogously. Consequently, it is possible to correct only elements of the i-th column of matrix A^i after performing of the i-th step of the algorithm (2.27) using the WCS method checking procedures for the column encoded matrix A_C.

Finally, the fault tolerant version of the Choleski algorithm will consist of the performing of following stages:

1. The original matrix **A** is represented in the form of the column encoded matrix $\mathbf{A_C}$ (2.6).

$$\boldsymbol{A_C} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \\ CS_1 & CS_2 & \cdots & CS_N \\ WCS_1 & WCS_2 & \cdots & WCS_N \end{bmatrix},$$

where $\boldsymbol{a[n+1, j] = CS_j = a_{1j} + a_{2j} + \ldots + a_{Nj}}$

and $a[n+2, j] = WCS_j = 1 \times a_{1j} + 2 \times a_{2j} + \ldots + N \times a_{Nj}$.

2. For i=1, 2, ..., N-1 , stages 3-6 are repeated.

3. At the beginning of the i-th step of the algorithm, the detection and correction procedures within elements belonging to the i-th column of A^i_C matrix are performing in according to the expressions (2.28) and (2.9)-(2.11).

This stage needs to perform (N-i) multiply-add operations and (N-i) additions.

4. The new elements of the i-th (pivoting) column are calculated.

5. The detection and the correction procedures for the computed elements belonging to the i-th column are performed in according to the expressions (2.28) and (2.9)-(2.11).

This stage requires (N-i) multiplications with additions and (N-i) additions.

6. The elements of matrix $A_C^{(i+1)}$ are calculated.

In the Pascal-like form the fault tolerant version of Choleski algorithm may be represented by the following construction, were δ is the small machine depended constant (roundoff value):

```
for i = 1 to N do
  begin
{ Errors detection and correction within  elements of the i-th column of A^i }
   PCS_i :=0;  QCS_i :=0;
   for j := i  to N do begin PCS_i := PCS_i + a_ji ;  QCS_i := QCS_i + j* a_ji; end;
   S1 := PCS_i  - a_N+1i ;   S2 := QCS_i  -  a_N+2i ;
   if  abs (S1) > δ and abs (S2) < δ  then a_N+1i  := PCS_i ;
   if  abs (S2) > δ and abs (S1) < δ  then a_N+2i  := QCS_i ;
   if  abs (S1) > δ and abs (S2) > δ  then begin  j:= S2 /S1 ;  a_ji := a_ji - S1; end;
   a_ii =  √a_ii  ;
   for j = i+1 to N do
       a_ji = a_ji / a_ii;
{ Errors detection and correction within new elements of the i-th column of matrix A^i }
   CS_i :=0;  WCS_i :=0;
     for  j := i to N do begin CS_i := CS_i + a_ji ;  WCS_i := WCS_i + j* a_ji; end;
   S1 := CS_i  - a_N+1i ;   S2 := WCS_i  -  a_N+2i ;
```

(2.30)

if abs *(S1)* > δ **and abs** *(S2)* < δ **then** $a_{N+1i} := CS_i$;

if abs *(S2)* > δ **and abs** *(S1)* < δ **then** $a_{N+2i} := WCS_i$;

if abs *(S1)* > δ **and abs** *(S2)* > δ **then begin** $j := S2/S1$; $a_{ji} := a_{ji} - S1$; **end;**

{Computation of the elements of the matrix A^{i+1}}

for $j = i+1$ **to** N **do**

begin

for $k = i+1$ **to** j **do**

$a_{jk} = a_{jk} - a_{ji} \times a_{ki}$;

end

end

Note, that the linear weighted encoder vector (2.4) is used here for coding of the QRS and OCS values.

Thus, using modified version of the WCS-method, the fault-tolerance versions of Choleski algorithm was designed.

Comparison of the constructions (2.27) and (2.30) shows that inserting the error detection and correction procedures increases the computational complexity of original algorithm (2.27) on the each i-th step on the 2(N-i) operations of multiplication with addition and 2(N-i) additions. This means, that the computational complexity of the whole algorithm increases on N^2 multiply-add operations and N^2 additions. Besides, due to the increasing of input matrix sizes, the computational complexity of the proposed algorithm (2.30) also is increased on $\cdot N^2$ multiply-add operations. As a result, the computational complexity of the whole fault-tolerant algorithm is increased approximately on $2 \cdot N^2 + O(N)$ multiply -add operations in comparison with the original algorithm (2.27). However, new algorithm enable to detect and to correct a single error in an arbitrary row or column of the input matrix A at the each algorithm step. Hence, it is possible to correct up to $N^2/4$ single errors during solving the whole decomposition task (2.26).

2.4. Designing of the fault tolerant versions of Faddeev and Jordan-Gauss algorithms

Starting with N×N, N×K, P×N and P×K input matrices **A**,**B**,**C** and **D,** respectively, the Faddeev algorithm is intended [20, 21, 29] for solving matrix equations of the type

$$X = C \cdot A^{-1} \cdot B + D \tag{2.31}$$

were the four input matrices form an (N + P)×(N + K) joint matrix $\tilde{\tilde{F}}$ when arranged in the following way:

$$\tilde{\tilde{F}} = \begin{bmatrix} A & B \\ -C & D \end{bmatrix} \tag{2.32}$$

The idea of the Faddeev algorithm consists of reducing the lower left quadrant of the matrix $\tilde{\tilde{F}}$ (i.e. **C**-matrix) to zero matrix, while in the lower right quadrant of the matrix $\tilde{\tilde{F}}$ (i.e. in a place of **D**-matrix), the resultant P×K matrix **X** is formed. In a order to performing above-stated operations with **A** being a non-singular matrix, the Gauss elimination algorithm is used. Hence, in the course of computation, the joint matrix $\tilde{\tilde{F}}$ is being transformed into the following matrix:

$$F' = F^{n+1} = \begin{bmatrix} R & B^* \\ O & X \end{bmatrix} \tag{2.33}$$

where **R** is the upper triangular matrix.

The main practical advantage of Faddeev algorithm is its versatility. This stems from the fact that expression (2.31) allows to solve a set of problems. Some of them are listed below:

- - solving a system ***AX = B*** of linear algebraic equations with one or more right-hand sides (depending upon the numbers of columns in **X**), i.e.

  $X = A^{-1}B$ for $C = I, D = 0$ **,**

  where ***I*** is the identity matrix;
- - matrix multiplication $X = CB$ for $A = I., D = 0;$
- - matrix multiply - add operation $X = CB + D$ for $A = I$;
- - matrix inversion $X = A^{-1}$ for $C = B = I, D = 0.$

There are other important modifications of the Faddeev algorithm. As a result, it can be employed, for example, in fast solving of linear programming problems using the Karmarkar algorithm.

To provide numerical stability of the Faddeev algorithm, Gaussian elimination with partial pivoting within columns [21, 23, 24] is usually used. As a result, at the i-th step (i=1,...,N) of the algorithm, the elimination of elements f_{ji}^{i} (j=i+1,..., N+P.), which belong either to the original matrix $\tilde{F} = F^{1}$ (for i=1) or to the partially transformed matrix $\tilde{F}^{i}$ (for i > 1), preceded by successive comparisons of f_{ji}^{i} (j = i + 1, ,N) with the pivot element f_{ji}^{i}. If

$$\left| f_{ji}^{i} \right| > \left| f_{ii}^{i} \right|$$

then the i-th and j-th rows of the matrix $\tilde{F}^{i}$ are interchanged and a Boolean variable v_{ji} is set to 1. In the opposite case, the row interchange does not take place, and v_{ji} is set to 0. After completing all comparisons and interchanges for a given step, the pivoting (i-th) row with the pivoting element f_{ii}^{i} is finally derived. Then the original Gauss elimination of the elements f_{ji}^{i} (j=i+1,...,N+P) starts. It is accompanied by transformations of rows of the matrix $\tilde{F}$ **,** from the (i+1) row to the (N+P) row.

However, to provide a correct realization of the algorithm, the selection of pivoting elements as well as corresponding interchanges are limited only to the upper (corresponding to the matrices A and B) quadrants of matrices $\tilde{F}^{i}$. Note, that the elimination process is carried out within all quadrants of $\tilde{F}^{i}$. Naturally, in the N-th step, the element f_{NN}^{N} is immediately taken as a pivoting one, without any comparison.

The described above version of the Faddeev algorithm can be expressed in the following form:

for *j:=i+1* **to** *N* **do**
  begin
    if abs(f_{ii}^{i}) < abs (f_{ji}^{i})
      then begin *s:=* f_{ii}^{i} *;* f_{ii}^{i} *:=* f_{ji}^{i} *;* f_{ji}^{i} *:=s;* v_{ji} *=1;* **end**
      else v_{ji} *=0;*

{row interchanges}

for $k=i+1$ **to** $N+K$ **do**

if $v_{ji}=1$ **then begin** $s:= f_{ik}^{\,i}$; $f_{ik}^{\,i} := f_{jk}^{\,i}$; $f_{jk}^{\,i} := s$; **end;**

end {j}**;**

{calculation of multipliers m_{ji} }

for $j:=i+1$ **to** $N+P$ **do** (2.34)

$m_{ji} := f_{ji}^{\,i} / f_{ii}^{\,i}$;

{elimination}

for $j=i+1$ **to** $N+P$ **do**

for $k=i+1$ **to** $N+K$ **do**

$f_{jk}^{\,i+1} = f_{jk}^{\,i} - m_{ji} * f_{ik}^{\,i}$;

end;

The Jordan-Gauss algorithm [23, 24] is an efficient alternative to classical Gauss elimination for the solution of dense linear systems of the form

$$A \cdot \bar{x} = \bar{b} \tag{2.35}$$

where ***A*** is **NxN** matrix of the system coefficients. The main advantage is that it gathers together two phases, triangularisation and back substitution [24]. In the case when ***X*** and ***B*** are **NxK** matrices, this algorithm is the particular case of Faddeev algorithm in which the two input matrices ***A*** and ***B*** form an joint **(N+N)x(N+K)** matrix $\tilde{\tilde{F}}$ of the following form:

$$\tilde{\tilde{F}} = \begin{bmatrix} A & B \\ -I & O \end{bmatrix}, \tag{2.36}$$

where ***I*** is the identity matrix, and ***O*** is a zero matrix. Then the **N** step of Gaussian elimination is performed for transforming the matrix $\tilde{\tilde{F}}$ into the matrix ***F'*** (2.33).

To provide numerical stability of this algorithm, the Gaussian elimination with partial pivoting may be used.

The described above version of Jordan-Gauss algorithm can be expressed in the following form:

for $i:=1$ **to** N **do**

begin

{selection of the pivot element}

for $j:=i+1$ **to** N **do**

begin

if abs(f_{ii}^{i}) < abs (f_{ji}^{i})

then begin $s:= f_{ii}^{i}$; $f_{ii}^{i} := f_{ji}^{i}$; $f_{ji}^{i} := s$; $v_{ji} = 1$; **end**

else $v_{ji} = 0$;

{row interchanges}

for $k=i+1$ **to** $N+K$ **do**

if $v_{ji} = 1$ **then begin** $s:= f_{ik}^{i}$; $f_{ik}^{i} := f_{jk}^{i}$; $f_{jk}^{i} := s$; **end;**

end {j}**;**

{calculation of multipliers m_{ji} }

for $j:=i+1$ **to** $N+i$ **do** (2.37)

$m_{ji} := f_{ji}^{i} / f_{ii}^{i}$;

{elimination}

for $j=i+1$ **to** $N+i$ **do**

for $k=i+1$ **to** $N+K$ **do**

$f_{jk}^{i+1} = f_{jk}^{i} - m_{ji} * f_{ik}^{i}$;

end;

where $f_{jk}^{1} = a_{jk}$, $j=1,2,...,N$, $k=1,2,...,N$;

$f_{j(N+p)}^{1} = b_{jp}$, $j=1,2,...,N$, $p=1,2,...,K$;

$f_{(N+1)i}^{1} = -1$, $i=1,2,...,N$;

$f_{(N+i)l}^{1} = 0$, $i=1,2,...,N$, $l=i+1,i+2,...,N+K$.

As a result of the execution of this algorithm, the desired elements of the matrix $\boldsymbol{X}$ are determined as follows:

$$x_{jp} = f_{(N+j)(N+p)}^{(N+1)} , \ j=1,2,...N, \ p=1,2,...K \qquad (2.38)$$

The analysis of Faddeev and Jordan-Gauss algorithms and comparison the corresponding constructions (2.34) and (2.37) allow to make the conclusion, that they differ from Gauss elimination algorithm (2.16) only more range of changes of the indices j and k. Therefore, it is easy proved that the theorems 2.1, 2.2, 2.3 and their consequences are corrected for

Jordan-Gauss and Faddeev algorithms. Consequently, the fault tolerant versions of these algorithms will also consist of the all stages of the above described fault tolerant Gauss elimination algorithm.

However, for the realization of the detection and correction of the elements of *i*-th column of matrix $\tilde{F}^i$ and m_{ji} it is necessary to perform 2*N multiply-add operations and 2*N additions in the case of Jordan-Gauss algorithm, and 2*(N-i+P) multiply-add operations and 2*(N-i+P) additions in the case of Faddeev algorithm. For realization of the detection and correction procedure for the elements of i-th (leading) row of the matrix $\tilde{F}^i$ it is necessary to perform (N-i+K) operations of multiplication with addition and (N-i+K) operations of addition for both Jordan-Gauss and Faddeev algorithms. Moreover, for both mentioned algorithms, the resulting elements x_{jp} are not correct during computations. Therefore, these elements should be checked and corrected after calculations by original checking procedure of the WCS method. For the realization of this stage, it is necessary to perform N*K operations of multiplication with addition and N*K operations of addition.

In the Pascal-like form the fault tolerant version of Faddeev algorithm without pivoting may be represented by the following construction, were δ is the small machine depended constant (roundoff value):

for *i* := *1* **to** *N* **do**

begin

{ Errors detection and correction within elements of the i-th column of F^i }

$PCS_i := 0;\ QCS_i := 0;$

for $j := i$ **to** $N+P$ **do begin** $PCS_i := PCS_i + f_{ji}\ ;\ QCS_i := QCS_i + j * f_{ji}$; **end;**

$S1 := PCS_i - f_{N+P+1,i}\ ;\ \ S2 := QCS_i - f_{N+P+2,i}\ ;$

if abs $(S1) > \delta$ **and abs** $(S2) < \delta$ **then** $f_{N+1+i,i} := PCS_i\ ;$

if abs $(S2) > \delta$ **and abs** $(S1) < \delta$ **then** $f_{N+2+i,i} := QCS_i\ ;$

if abs $(S1) > \delta$ **and abs** $(S2) > \delta$ **then begin** $j := S2/S1\ ;\ f_{ji} := f_{ji} - S1;$ **end;**

{ Errors detection and correction within elements of the i-th row of F^i }

$PRS_i := 0;\ QRS_i := 0;$

for $k := i$ **to** $N+K$ **do begin** $PRS_i := PRS_i + f_{ji}\ ;\ QRS_i := QRS_i + k * f_{ji}$; **end;**

$S1 := PRS_i - f_{i,N+K+1}$; $S2 := QRS_i - f_{i,N+K+2}$;

if abs $(S1) > \delta$ **and abs** $(S2) < \delta$ **then** $f_{i,N+K+1} := PRS_i$;

if abs $(S2) > \delta$ **and abs** $(S1) < \delta$ **then** $f_{i,N+K+2} := QRS_i$;

if abs $(S1) > \delta$ **and abs** $(S2) > \delta$ **then begin** $k := S2/S1$; $f_{ik} := f_{ik} - S1$; **end;**

{Computation of the elements of the matrix M}

for $j := i + 1$ **to** $N+P+2$ **do**

$m_{ji} := f_{ji} / f_{ii}$;

{ Errors detection and correction within elements of the i-th column of matrix M }

$CS_i := 1$; $WCS_i := j$;

for $j := i+1$ **to** $N+P$ **do begin** $CS_i := CS_i + m_{ji}$; $WCS_i := WCS_i + j * m_{ji}$; **end;** (2.25)

$S1 := CS_i - f_{N+P+1,i}$; $S2 := WCS_i - f_{N+P+2,i}$;

if abs $(S1) > \delta$ **and abs** $(S2) < \delta$ **then** $f_{N+P+1,i} := CS_i$;

if abs $(S2) > \delta$ **and abs** $(S1) < \delta$ **then** $f_{N+P+2,i} := WCS_i$;

if abs $(S1) > \delta$ **and abs** $(S2) > \delta$ **then begin** $j := S2/S1$; $f_{ji} := f_{ji} - S1$; **end;**

{Elimination of the elements of matrix F}

for $j := i + 1$ **to** $N+P+2$ **do**

for $k := i + 1$ **to** $N+K+2$ **do**

$f_{jk} := f_{jk} - m_{ji} * f_{ik}$;

end;

where $f_{jl}^1 = a_{jl}$, $j=1,2,...,N$, $l=1,2,...,N$;

$f_{j(N+k)}^1 = b_{jk}$, $j=1,2,...,N$, $k=1,2,...,K$;

$f_{(N+p)l}^1 = -c_{pl}$, $p=1,2,...,P$ $l=1,2,...,N$;

$f_{(N+p)k}^1 = d_{pk}$, $p=1,2,...,P$, $k=1,2,...,K$.

Note, that the linear weighted encoder vector (2.4) is used here for coding of the QRS and OCS values.

Thus, the proposed fault tolerant versions of Jordan-Gauss and Faddeev algorithms permit to correct a single error in the each column of matrix $\tilde{F}^i$. This means, that it is possible to correct up to N^2 errors during whole time of the algorithm implementation.

2.5. Numerical properties of WCS method in the case of floating point realization

In [99] the numerical properties of single error correction (SEC) codes based on linearly and exponentially weighted encoder vectors was considered in detail for the case of ABFT floating - point implementation. The main result is next. In the cases when encoder vectors (2.3) and (2.4) are applied, respectively $\log_2 N$ and $\log_2(\log_2 N)$ extra bits are needed to provide numerical accuracy of computation, i.e. to provide that no false alarms occur in worst-case round errors. Therefore both (2.3) and (2.4) encoder vectors are few suitable for ABFT floating-point realization.

In [61] the three various stages of ABFT technique which are prone to numerical errors were identified: the coding phase, actual data computation phase, and error correction phase. Note, that the round errors of second stage is determined only numerical properties of applied algorithm. For example, in [60] the value of tolerance τ (so that a row (column) of resulting matrix will be accepted as error-free if the difference between the computed row (column) sum and checksum is less than τ) was determined for case of floating point implementations of LU- and QR- decomposition algorithms. In particular, it was be shown, that the value of tolerance τ is necessarily large for the LU - decomposition and Gaussian elimination with pairwise pivoting, but is acceptably small for the Gaussian elimination with partial pivoting and QR-decomposition.

It has been proved in the ref. [70], that the maximum round error during of data vector **a**(N) coding, is given by

$$E \le \varepsilon \cdot \delta \cdot |\mathbf{a}|^2 \cdot |\mathbf{x}|^2, \qquad (2.39)$$

where **x**(N) - encoder vector , $|\bullet|^2$ - Euclidean norm of vector, ε and δ - machine dependent parameters. Thus, the straightforward way to reduce the amount of error during coding is to minimize $|\mathbf{x}|^2$. However, it has been shown in [61], that we can not select an $|\mathbf{x}|^2$ as small as we wish because such vector x would not have the high reflectively, i.e. that the ratio of the change in the code value to the change in the data element would very small. High reflectivity of a code is essential in the error correction phase, because if the reflectivity of code is very small, two errors in the data element,

which are almost equal in value, but at the same time distinctly observable, will reflect the same amount of error in the check element which makes the discrimination of these two errors very difficult. Therefore, the strategy should be to make a compromise in selecting the norm of encoder vector which will give a small, and at the same time, will have moderately high reflectivity. What is why in the ref. [61] were presented some examples of experimental designed encoder vectors which have small value of Euclidean norm and high value of reflectivity. Its are, in particular, next:

1) average and weighted average encoder vectors:

$$\mathbf{p}^T = [\ 1/N\ \ 1/N\ ...\ 1/N\ ] \quad \text{and} \quad \mathbf{q}^T = [\ 1/N\ \ 2/N\ ...\ N/N\ ]; \tag{2.40}$$

2) normalised encoder vectors

a) for vector **a**(N)

$$\mathbf{p}^T = [\ c/|\mathbf{a}|^2 \quad c/|\mathbf{a}|^2 \ \ ...\ \ c/|\mathbf{a}|^2\ ] \tag{2.41}$$

b) for matrix A(M,N)

$$\mathbf{q}^T = [c/|\mathbf{A}| \quad c/|\mathbf{A}| \ \ ...\ \ c/|\mathbf{A}|\ ], \tag{2.42}$$

where $|\mathbf{A}|$ is the average value of matrix columns (or rows) Euclidean norms and c is a constant fixed by user.

Experimental evaluation of numerical error for proposed encoder vectors also were researched in [61]. The main result are next: when round errors are the larger problem, one should use normalized encoder vectors; for overflow problems, one should use average encoder vectors.

In order to estimating of the tolerance of the proposed algorithms to transient faults and evaluation of numerical error for different encoder vectors, the programmed environment „ABFT" (*Algorithm-Based Fault Tolerance*) was designed in Borland Delphi package using Object Pascal language.

This program allows:

- to examine the corrected executing of the proposed algorithms for different input data and checksums types : Single, Real, Double;

- to select the type of the encoded vector: Linear, Average or Normalized;
- to select the numerical accuracy of computations;
- to select the error values and to inject the single errors: for the each algorithm, in an arbitrary algorithm step, into an arbitrary matrix element;
- to type the results of the algorithm implementation and detected errors;
- to select the breakpoints: step over or after error finding;
- to read of input data from files and to write of the results to output files;

The environment was designed with Polish language user interface. Therefore, in the Appendix 1 the description of this environment is also represented in Polish language.

The main results of the proposed algorithms testing are following:

$\Rightarrow$ - the proposed fault tolerant Gauss elimination, Choleski, Jordan-Gauss and Faddeev algorithms permit to detect and to correct a single error in the each column of input matrix at each step of the corresponding algorithm implementation;

$\Rightarrow$ - for different input matrix elements and checksum data types (Single, Real or Double), the value of tolerance δ (so that a row (column) of resulting matrix will be accepted as error-free if the difference between the computed row (column) sum and checksum is less than δ) in the algorithms (2.25), (2.30) and (2.34) must be selected the same value for small size input matrix or more value for large matrix. For example, for input data of Single format (accuracy is equal 10E-7) and input matrix size equal 100, the error more than δ =10E-5 may be detected and corrected in the case of Choleski algorithm with the linear weighted encoded vector (2.4). Thus, the increasing of input matrix sizes needs in increasing of the tolerance value δ.

2.5. Conclusions of the chapter 2

1. The common property of the selected LA algorithms is the computation on the any i-th algorithm step (may be not one times) the elements of leading (i-th) row or/and column of matrix $\mathbf{A^i} = \{a_{ji}^{\mathbf{i}}\}$ and then modification of others matrix rows (columns) by means leading ones. Therefore, if at the i-th algorithm step the element of leading column (or row) is wrongly calculated, then errors will appear in all elements of the corresponding

row (or column) of $\mathbf{A}^{i+1}$. In both cases, these errors can not be corrected by original WCS method.

2. The several theorems were proved for the Gauss elimination, LU-decomposition, Choleski, Jordan-Gauss and Faddeev algorithms, which allowed to perform the modification of the origin WCS method. The main idea of the proposed modified WCS method (scheme) is the performing of the detection and correction procedures:

- - at each algorithm step;
- - among only elements of the leading row and leading column of input matrix.

3. The sufficient conditions of using of the modified WCS method for others LA algorithms were formed.

4. The fault tolerant versions of the above mentioned algorithms were designed using modified WCS method. The computational complexity of the whole fault-tolerant algorithm is increased approximately on $O(N^2)$ multiply -add operations in comparison with the original algorithm. However, new fault tolerant algorithms enable to detect and to correct a single error in an arbitrary row or column of the input matrix at the each algorithm step. Hence, it is possible to correct up to $N^2/2$, $N^2/4$ and N^2 single errors during realization of the whole Gauss, Choleski and Jordan-Gauss algorithms respectively.

5. Numerical properties of WCS method in the case of floating point realization were researched and the encoder vectors [61], which have small value of Euclidean norm and high value of reflectivity, were selected for using in the proposed algorithms. For estimating of the tolerance of the proposed algorithms to transient faults and evaluation of numerical error for different encoder vectors, the programmed environment „ABFT" (*Algorithm-Based Fault Tolerance*) was designed. The testing of the proposed algorithms proved that they are correct and enable to detect and to correct a single error in an arbitrary row or column of the input matrix at the each algorithm step.

CHAPTER 3. DESIGN OF FPGA-BASED PROCESSOR ARRAY ARCHITECTURES FOR LINEAR ALGEBRA ALGORITHMS IMPLEMENTATION

It was be shown in the chapter 1 of this manuscript, that the main LA algorithms may be effectively solved by different kinds parallel computers. Moreover, only specialized or reconfigurable parallel architectures (destined for the further realization as ASIC, FPGA-based or Raw-processors devices) may be effectively used for the real time LA applications. Therefore, the problems of designing of such specialized parallel architectures for main LA algorithms is considered in this chapter. The VLSI processor arrays [3, 7, 8, 22, 26, 27] are typical examples of such architectures. Using massive pipelining, these arrays exploit the regularity inherent in many algorithms to achieve high performance while keeping local communications and low I/O requirements.

3.1. Method for deriving dependence graphs of recursive algorithms

VLSI processor arrays can be designed systematically by applying linear (or affine) mappings to algorithms that are expressed as systems of recursive equations or, equivalently, by nested loops [2, 3, 7, 8, 25 - 33]. Such algorithms can be represented by regular or quasi-regular, lattice dependence graphs, or a composition of such dependence graphs (DGs). Each node of such a lattice DG corresponds to a certain iteration of the original algorithm, and is associated with an integer vector $K = (I_1, \ldots, I_n)$ because all the nodes are located in the vertices K of an integer lattice K^n subset Z^n. Arcs between nodes of this DG, or data dependencies between iterations of the algorithm, are compactly represented by the dependence matrix D, in which the j-th column is a dependence vector d_j. If the iteration corresponding to a node K_2 depends on the iteration corresponding to another node K_1, this dependence is represented by the difference $d = K_2 - K_1$. For strictly regular (or uniform) DGs, these dependence vectors (or simply dependencies) are constant for all nodes K (i.e. independent of K in K^n). Algorithms with regular data dependencies occur frequently in signal/image processing, and numerical applications [3].

For an algorithm given by nested loops with regular data dependencies, its lattice DG with the index set K^n and dependence matrix D can be constructed, in an analytic way, either using the methods described, e.g.,in Ref. [32, 100], or a method proposed in this paper. In comparison with known analytical methods, our method is more simple and feasible for the CAD implementation, and allows us to operate with a wider class of algorithms. For example, opposite to [32,100], we can deal not only with uniform recursive algorithms corresponding to perfect loop nests [101], but also with non-uniform ones which correspond to non-perfect (or composite) loop nests.

Our method is aimed to a construction of lattice DGs for algorithms given by composition of perfect and composite nested loops with regular or quasi-regular data dependencies. Starting with the program corresponding to the original algorithm, this method also allows us to find the dependence matrix D, which is employed by most of the known mapping methods. This feature becomes especially important for algorithms with a large dimension n.

We assume that an elementary loop nest (ELN) consists of a multilevel construction of nested **do**-statements including one another, and the corresponding loop body without any exit from it. The description of the nest in a Fortran-like notation is as follows:

do $I_1 := a_1$ **to** b_1 **step** c_1
  do $I_2 := a_2$ **to** b_2 **step** c_2
 . . . . . .
    do $I_n := a_n$ **to** b_n **step** c_n
      {statements of the loop body} (3.1)
    enddo
 . . . . . .
  enddo
enddo

Here a_j , b_j and c_j are expressions denoting the lower, upper limits and step of the loop at nesting level j $(j=1,2,...,n)$. Moreover, a_1 , b_1 and c_j are constants, while others of a_j and

b_j are given by affine functions of I_j [102]. The number of nodes in the DG corresponding to loop (3.1) is determined by differences $w_j = b_j - a_j$. While constructing DGs of algorithms with regular or quasi-regular dependencies, we assume that all these differences are fixed numbers. The statements of the loop body contain some indexed variables $X_{m1,\ldots,mQ}$, whose indices m are functions of I_j .

Each elementary nest is characterized by its dimension n (which is equal to the number of **do**-statements) and defines the corresponding iteration space. Each of its vertices represents a single execution of the loop body, and is defined by an iteration vector $K=\{i_1,i_2,\ldots,i_n\}$, where i_j is equal to the value of I_j during the corresponding iteration. If between two consequent **do**-statements there exist a loop body, then such loop construction will be called the composite loop nest. It can be written in the Fortran-like form as follows:

do I_1 := a_1 **to** b_1 **step** c_1
  [{loop body 1}]
 [enddo]
  do I_2 := a_2 **to** b_2 **step** c_2
   [{loop body 2}]
  [enddo]
. . . . . .
  do I_n := a_n **to** b_n **step** c_n
   {loop body t} (3.2)
  enddo
. . . . . .
   [{loop body 2}]
  [enddo]
 [{loop body 1}]
[enddo]

Here square brackets are used to denote that the corresponding statements may be absented, and t is the number of different loop bodies $(t \le n)$. Besides, the each loop body s $(s=1,2,\ldots,t)$ may be included only one times into the construction (3.2).

Note that the construction (3.2) can be splitted into t elementary loop nests. Each s-th ELN can be represented in the form (3.1) with $n = n_s$.

We restrict our analysis to nests in which the lexicographical order of executing its iterations is unambiguously determined before computations. It means, that each iteration has own lexicographical number, and, for two different iterations, firstly will be executed

the iteration with the smaller lexicographical number. Without a loss of a community we also assume in this paper that the arbitrary loop body consists of one assignment statement. As a result, algorithms under consideration can be written by composition of the elementary and/or composite loop nests.

3.1.1. Derivation of dependence graph for elementary loop nest

In order to derive the lattice DG of an elementary nest, the following parameters should be determined:

1) the dimension n of its iteration space;

2) the coordinates of all nodes in the DG ;

3) coordinates of all arcs (vectors) between the nodes.

The dimension n is equal to the number of **do**-statements in the nest.

To find the coordinates of nodes in the DG , we propose to use expressions denoting the lower, upper limits and step of the each loop variable I_j $(j=1,2,...,n)$. Note that they are equaled to the values a_j , b_j and c_j correspondingly in the construction (3.1).

For determining arcs between the nodes and their direction (i.e. coordinates of the connecting vectors), we base on the following confirmation.

Confirmation 3.1. If statement of the n-dimensional ELN (3.1) consists of the r different indexed variable $X^k_{m1,...,mQ}$ $(k=1,2,...,r)$ where indices m_p $(p=1,2,...,Q)$ are the affine functions from loop parameters I_j $(j=1,2,...,n)$, then variable $X^k_{m1,...,mQ}$ will transmit between DG nodes along vector-arc d^k with the coordinates $d^k = (h_1, h_2, ..., h_n)$, where $h_j = c_j$, if arbitrary index m_p is not a function from the variable I_j, and $h_j = 0$ in the opposite case.

The proof of this confirmation is based on the following properties of a DG. The presence of an arc between two nodes of the DG means that a certain indexed variable is transferred between these nodes. Hence, this arc will exist only if this variable has the same values of its indices in both nodes. Moreover, the vector-arc will be directed from the node with the less value of the lexicographical number. In according to above assumed definitions, the variable $X_{m1,...,mQ}$ has same values of its indices $m_1,...,m_Q$ at the various iterations of the algorithm, i.e. for various values of one or more loop parameters

I_j $(j=1,2,...,n)$ only in the case when all its indices are not dependent from these parameters. Therefore, for example, if for variable $X_{m1,...,mQ}$ such parameter I_j (one or more) exists then this variable should be transmitted to the all DG nodes which have same values of the all remained coordinates. The vectors-arcs will connect only such nodes in DG, which are differed only in the values I_j and are neighboring.

It is follows from regularity of the algorithm given by ELN (3.1) that derived vectors will connect the all nodes in its DG.

Consequence 3.1. If the statement of the ELN includes *e* different indexed variables with the same indices (index expressions) then *e* same vectors will connect the all nodes in its DG.

Note that the Confirmation 3.1 allows to determine the coordinates of the all vectors-arcs in ELN DG only if there are no variables with the same name presented on the left and right sides of assignment statement of the loop body in program (3.1), for example $X^k_{m1,...,mQ}$ and $X^k_{u1,...,uQ}$ respectively (such variables are named recomputing ones). In the opposite case, the DG may to have extra arcs between its nodes because a variable $X^k_{m1,...,mQ}$ which has been computed in a certain iteration $K=\{i_1,i_2,...,i_n\}$ is then used as an argument $X^k_{u1,...,uQ}$ for an iteration $K'=\{(I_1+h_1), (I_2+h_2) ,..., (I_n+h_n)\}$ (where h_j is the constant which is divisible to the value c_j $(j=1,2,...,n)$ and has same sign) executed afterwards.

For determining such vectors-arcs, we use following confirmation which ensues from assumed definitions and Confirmation 3.1.

Confirmation 3.2. Let the variables $X^k_{m1,...,mQ}$ and $X^k_{u1,...,uQ}$ are presented on the left and right sides of assignment statement of the ELN (3.1), respectively, and their indices m_p and u_p $(p=1,2,...,Q)$ are the affine functions from loop parameters I_{jm}, I_{ju} $(jm, ju \in \{1,2,...,n\};)$. Then the calculated in the iteration (node) $K=\{I_1,I_2,...,I_n\}$ variable $X^k_{m1,...,mQ}$ will transmit in the capacity of argument $X^k_{u1,...,uQ}$ to the all nodes $K'=\{(I_1+h_1), (I_2+h_2) ,..., (I_n+h_n)\}$ which belong to the ELN iteration space, where h_j $(j=1,2,...,n)$ are determined from systems of the *Q* equations

$$h_j = m_p - u_p \ , (p=1,2,...,Q) \ . \qquad (3.3)$$

The remained values of the values h_j should be determined in according to the Confirmation 3.1.

Consequence 3.2. It is follows from regularity of the algorithm given by ELN (3.1) that variable $X^k_{m1,...,mQ}$ will transmit along vector $d^k = K' - K = (h_1', h_2', ..., h_n')$ to the all nodes $K' = \{(I_1+h_1), (I_2+h_2), ..., (I_n+h_n)\}$ which belong to the ELN iteration space, where $h_j' = min(|h_j|)$, if $sign(h_j') = sign(c_j)$ and $h_j' = 0$ in the opposite case.

Consequence 3.3. If the variables $X^k_{m1,...,mQ}$ and $X^k_{u1,...,uQ}$ are presented on the left and right sides of assignment statement of the ELN (3.1), respectively, and all their indices are same , i.e $m_p = u_p$ for $p=1,2,...,Q$, then derived vector -arc will be equal zero.

Assuming that the original loop nest has the form (3.1), the procedure for constructing the DG of the s-th elementary nest is as following:

1. Determine the dimension n of the nest.
2. Determine the coordinates of the all DG nodes.

In order to this we construct the matrix V(n×3) of the limits of the iterations space of DG

$$V = \begin{bmatrix} a_1 & b_1 & c_1 & I_1 \\ a_2 & b_2 & c_2 & I_2 \\ \cdots\cdots & \cdots\cdots\cdots & & \\ a_n & b_n & c_n & I_n \end{bmatrix} \qquad (3.4)$$

where j-th row finds the lower, upper limits and step of the loop parameter I_j $(j=1,2,...,n)$. Note that they are equaled to the corresponding values a_j , b_j and c_j in the construction (3.1).

Then the coordinates of the first and last executed nodes of DG should be determined. In according to the assumed definitions they are equal

$$K_1 = (a_1, a_2, ..., a_n) \text{ and } K_z = (b_1, b_2, ..., b_n) \qquad (3.5)$$

respectively.

The coordinates of the remaining DG nodes are derived by sequential modification of the values of the each coordinate I_j on step value c_j . Note that obtained value should be satisfied to the expression

$$a_j \le I_j \le b_j \, . \qquad (3.6)$$

3. Construct an *n*-dimensional integer lattice K^n and locate nodes of the DG in its vertices in according to their coordinates.

4. In according to the Confirmation 3.1 determine coordinates of the *r* vectors-arcs d^k of the DG corresponding to transmission of the all indexed variables $X^k_{m1,...,mQ}$ $(k=1,2,...,r)$ of the ELN loop body. The determined non-zero vectors include into dependence matrix D of the ELN and connect the all DG node by derived vectors.

5. If there are no more variables with the same name and different sets of index functions corresponding to opposite sides of assignment statements of the loop body in program (3.1), then the procedure of constructing the DG is completed. In the opposite case, for each from these variables find the set of iterations (nodes) $K'=\{(I_1+h_1), (I_2+h_2) ,..., (I_n+h_n)\}$ in which these variables take part in the calculations and coordinates of vectors-arcs corresponding to transmission these variables in derived nodes K'. The obtained non-zero vectors include into dependence matrix D of the ELN and connect by each obtained vector only corresponding nodes K' of the DG.

3.1.2. Derivation of a dependence graph for the whole algorithm

The resulting lattice DG of the whole algorithm is constructed on the base of DGs of all the *t* elementary nests. Note that value *t* is equal to the amount of the the loop bodies of the algorithm and the s-th loop-body corresponds to the *s*-th ELN $(s=1,2,...,t)$. The dimension of the resulting DG will be given by

$$n = max\{n_1, n_2,...,n_t\},$$

where n_s - is the dimension of the *s*-th ELN.

Thus, for derivation of the DG for the whole algorithm, it is necessary to locate all the DGs corresponding to elementary nests inside an *n*-dimensional integer lattice K^n , in accordance with the previously determined coordinates of nodes of these DGs. For DGs with dimensions $n_s < n$, coordinates of their nodes should be completed by $(n - n_s)$ absent coordinates. Moreover, the nodes of different elementary DGs should be connected by arcs with the unit length [16], according to data dependencies between different elementary nests.

Note that in according to above assumed definitions the data dependencies from the ELN s_1 to the ELN s_2 , where s_1 , $s_2 \in \{1,2,...,t\}$, $s_1 \neq s_2$, exist only if the calculated in the

ELN s_1 variable $X^k_{m1,...,mQ}$, then takes part in the ELN s_2 as argument of the loop body $X^k_{u1,...,uQ}$. Therefore, in order to the determination of the values of $(n - n_s)$ absent coordinates of the s-th ELN ($n_s < n$) we apply the following confirmation which is based on the ideas described above.

Confirmation 3.3. If exist data dependencies from the n_{s1} -dimensional ELN s_1 to the n_{s2} -dimensional ELN s_2 , where s_1 , $s_2 \in \{1,2,...,t\}$, $s_1 < s_2$, then:

a) in the case when $n_{s1} < n_{s2}$, $(n_{s1} - n_{s2}) = \Delta n$, the each node of the ELN s_1 should be completed by Δn absent coordinates I_δ , $I_\delta \in (I_1, I_2, ..., I_n)$ and each absent coordinate I_δ is equal $I_\delta = a_\delta - c_\delta$, where a_δ and c_δ are the lower limit and step values of the coordinate I_δ in the ELN s_2 ;

b) in the case when $n_{s1} > n_{s2}$, $(n_{s2} - n_{s1}) = \Delta n$, the each node of the ELN s_2 should be completed by Δn absent coordinates I_δ , $I_\delta \in (I_1, I_2, ..., I_n)$ and each absent coordinate I_δ is equal $I_\delta = b_\delta + c_\delta$, where b_δ and c_δ are the upper limit and step values of the coordinate I_δ in the ELN s_1 .

Then the following confirmation should be used in order to deriving the coordinates of vectors-arcs connected the nodes between the different n-dimensional ELN's in according to the data dependencies of the algorithm. Note that this confirmation is the modified Confirmation 3.2.

Confirmation 3.4. Let the variables $X^k_{m1,...,mQ}$ and $X^k_{u1,...,uQ}$ are presented on the left and right sides of assignment statement of the ELN (3.1), respectively, and their indices m_p and u_p $(p=1,2,...,Q)$ are the affine functions from loop parameters I_{jm}, I_{ju} $(jm, ju \in \{1,2,...,n\};)$. Then the calculated in the iteration (node) $K=\{I_1,I_2,...,I_n\}$ variable $X^k_{m1,...,mQ}$ will transmit in the capacity of argument $X^k_{u1,...,uQ}$ to the all nodes $K'=\{(I_1+h_1), (I_2+h_2) ,..., (I_n+h_n)\}$ which belong to the iteration space of the algorithm, where h_j $(j=1,2,...,n)$ are determined from systems of the Q equations (3.3). The remained values of the values h_j should be determined in according to the Confirmation 3.1.

Consequence 3.4. It is follows from regularity of the algorithm represented by the composition of the elementary and/or composite loop nests that the variable $X^k_{m1,...,mQ}$ will transmit along vector $d^k = K' - K=(h_1', h_2', ..., h_n')$ to the all nodes $K'=\{(I_1+h_1), (I_2+h_2)$

,..., $(I_n+h_n)\}$ which belong to the ELN iteration space, where $h_j'=min(|h_j|)$, if $sign(h_j')=sign(c_j)$ and $h_j'=0$ in the opposite case.

Thus, the method for deriving the DG of the whole algorithm of the algorithm represented by the composition of the elementary and/or composite loop nests is formulated in the following way:

1. Look through the program of the algorithm, and assign numbers $s = 1, 2, ..., t$ to succeeding loop bodies, as well as extract all the t elementary loop nests corresponding to these bodies.

2. Find for each s-th ELN its dimension n_s and determine the dimension n of the DG of the whole algorithm.

3. For each elementary nest, construct its DG using the procedure described above.

4. For each s-th ELN with dimension $n_s < n$ we determine the values of $(n - n_s)$ absent coordinates of its nodes and complete its matrix of the iteration space V_s.

Then we complete the coordinates of the all obtained vectors-arcs by coordinates I_g $(g = n_s+1,..., n)$ which are absent in this ELN, where $I_g = c_j$, if I_g is the function of an arbitrary from coordinates I_j $(j=1,..., n)$ and $I_g = 0$ in the opposite case. The obtained non-zero vectors include to the dependence matrix D of the algorithm.

5. By means a joining modified matrices of iteration space of ELN's consist the matrix $V(n\times3)$ of the iteration space of the whole algorithm

$$V = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ \cdots\cdots & \cdots\cdots & \\ a_n & b_n & c_n \end{bmatrix}$$

in which the each row determines lower, upper limits and step of the coordinate I_j $(j=1,2,...,n)$ of the algorithm iteration space. Note that these values are equal to minimal values of the variables a_j, c_j and maximal values of the variable b_j from the all ELN's matrices V_s $(s=1,...,t)$ respectively.

6. Determine the coordinates of the first and last executed nodes of the algorithm DG. They are equal

$$K_1=(a_1, a_2, ..., a_n) \text{ and } K_2=(b_1, b_2, ..., b_n)$$

respectively.

The coordinates of the remaining DG nodes are derived by sequential modification of the values of the each coordinate I_j on step value c_j . Note that obtained value should be satisfied to the expression

$$a_j \le I_j \le b_j .$$

7. Construct an *n*-dimensional integer lattice K^n and locate nodes of the DG in its vertices in according to their coordinates $(I_1, I_2, ..., I_n)$.

8. In according to the Confirmation 3.4 and its consequence, for each pare of ELN which have data dependencies between their variables we determine the set of iterations (nodes) $K'=\{(I_1+h_1), (I_2+h_2) ,..., (I_n+h_n)\}$ in which these variables take part in the calculations and coordinates of vectors-arcs corresponding to transmission these variables in derived nodes *K'*. The obtained non-zero vectors include into dependence matrix D of the ELN and connect by each obtained vector only corresponding nodes *K'* of the DG.

3.1.3. Example. Deriving DG of the Gauss elimination algorithm

The algorithm corresponding to Gauss elimination algorithm without pivoting is represented by program (2.16) in the chapter 2. The original algorithm includes the following two loop body statements :

first - $m(I_2, I_1) := a(I_2 , I_1) / a(I_1 , I_1)$,

and second - $a(I_2, I_3) := a(I_2 , I_3) - m(I_2 , I_1) * a(I_1 , I_3)$.

Thus, the original algorithm consists of $t=2$ elementary loop nests, which are given by Fortran-like program fragments (3.7) and (3.8) respectively.

```
        {First ELN}
  do I1 := 1 to N-1 step 1
    do I2 := I1 + 1 to N  step 1
       m( I2, I1 ): = a(I2 , I1 ) / a(I1 , I1 )                    (3.7)
    enddo
  enddo

    {Second ELN}
  do I1 := 1 to N-1 step 1
```

do $I_2 := I_1 + 1$ **to** N **step** 1

do $I_3 := I_1 + 1$ **to** N **step** 1

$a(I_2, I_3) := a(I_2, I_3) - m(I_2, I_1) * a(I_1, I_3)$ (3.8)

enddo

enddo

enddo

1. In according to the proposed method assign numbers $s = 1$ and $s = 2$ the first and second ELN's respectively.

2. Find for each s-th ELN its dimension n_s and determine the dimension n of the DG of the whole algorithm :

$$n_{s1} = 2,\ n_{s2} = 3 \text{ and } n = 3.$$

3. For each elementary nest, construct its DG using the procedure described above.

a) Firstly, construct the iteration space matrices $V_{s1}(n_{s1} \times 3)$ and $V_{s2}(n_{s2} \times 3)$ for first and second ELN:

$$V_{s1} = \begin{bmatrix} 1 & N-1 & 1 & I_1 \\ I_1 + 1 & N & 1 & I_2 \end{bmatrix}$$

$$V_{s2} = \begin{bmatrix} 1 & N-1 & 1 & I_1 \\ I_1 + 1 & N & 1 & I_2 \\ I_1 + 1 & N & 1 & I_3 \end{bmatrix}$$

b) Then determine the coordinates of the first and the last executed nodes of DG. In according to the proposed method they are equal

- for first ELN $K_1=(1, 2)$ and $K_z=(N-1, N)$;
- for second ELN $K_1=(1, 2, 2)$ and $K_z=(N-1, N, N)$.

c) The coordinates of the remaining DG nodes $K=(I_1, I_2)$ (of the first ELN) are derived by sequential modification of the values of the each coordinate I_1 , I_2 on the step value $c_1 = c_2 = 1$. Note that obtained values should be satisfied to the expression

$$1 \le I_1 \le N-1$$

and

$$I_1 + 1 \le I_2 \le N.$$

For example, for N=4 the following set of nodes of the first ELN were obtained:
(1,2); (1,3); (1,4); (2,3), (2,4); (3,4).

Analogously, for second ELN we obtain the following set of the DG nodes:
(1,2,2); (1,2,3); (1,2,4); (1,3,2); (1,3,3); (1,3,4); (1,4,2); (1,4,3); (1,4,4); (2,3,3), (2,3,4); (2,4,3); (2,4,4); (3,4,4);.

d) Construct the n_{s1}=2 and n_{s2}=3 dimensional integer lattices and locate nodes of the DG's in its vertices in according to their coordinates.

e) In according to the Confirmation 3.1 determine coordinates of the r vectors-arcs d^k of the DG corresponding to transmission of the all indexed variables $X^k_{m1,...,mQ}$ $(k=1,2,...,r)$ of the ELN loop body.

The first ELN includes three such variables (r =3): $a(I_1 , I_1)$, $a(I_2 , I_1)$ and $m(I_2 , I_1)$. However, only indices of the variable $a(I_1 , I_1)$ are not dependent from the loop parameter I_2 . Consequently, variable $a(I_1 , I_1)$ will transmit between the nodes of the ELN DG along the vector $d^1 = (h_1, h_2)=(0, c_2)=(0,1)$. For remained two variables we obtain the vectors $d^2 = d^3 = (0, 0)$.

The determined vector d^1 is included into dependence matrix D_{s1} of the first ELN and connect the all DG nodes by this vector.

The second ELN also includes three such variables: $a(I_2 , I_3)$, $a(I_1 , I_3)$ and $m(I_2 , I_1)$. The indices of the variable $a(I_2 , I_3)$ are not dependent from the loop parameter I_1 . Consequently, variable $a(I_2 , I_3)$ will transmit between nodes of the ELN DG along the vector $d^1 = (h_1, h_2, h_3)=(c_1, 0, 0)=(1, 0, 0)$. For remained variables $a(I_1 , I_3)$ and $m(I_2 , I_1)$ we obtain the vectors $d^2 = (0, 1, 0)$ and $d^3 = (0, 0, 1)$ respectively .

The determined vectors d^1, d^2 , d^3 is included into dependence matrix D_{s2} of the second ELN and connect the all DG nodes by these vectors.

f) There are no variables with the same name and different sets of index functions corresponding to opposite sides of assignment statement in the first ELN. Therefore, the procedure of constructing the DG of this ELN is completed. This DG is shown in the fig.3.1.

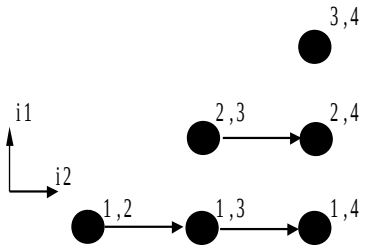


Fig. 3.1. Elementary DGs for first ELN

In the second ELN there are two pare of such variables:

- $a(I_2, I_3)$ in the left part and $a(I_2, I_3)$ in the right part of the loop body;
- $a(I_2, I_3)$ in the left part and $a(I_1, I_3)$ in the right part of the loop body.

In according to Consequence 3.3 determine that the vector corresponding to the data dependencies between first pare of the variables is equal zero.

For second variables pare in according to the Confirmation 3.2 we obtain:

$$m_1 = I_2 , m_2 = I_3 , u_1 = I_1 , u_2 = I_3$$

and

$$h_1 = I_2 - I_1 \quad , \quad h_3 = I_3 - I_3 ,$$

where $I_2 \in \{ I_1 +1, I_1 +2,..., N\}$. Consequently, $h_3 = 0$ and $h_1 \in \{1, 2,..., N - I_1 \}$ and $h_1' =1 , h_3' = 0$.

The value of the variable h_2 we determine by means Confirmation 3.1. In according to this confirmation obtain $h_2 = c_2 =1$.

As a result, the vector-arc $d^4 = (1, 1, 0)$ is determined which corresponds to the transmission of the calculated in the node $K=\{I_1, I_2, I_3\}$ variable $a(I_2 , I_3)$ to the node $K'=\{(I_1+1), (I_2+1), I_3\}$ as the argument $a(I_1 , I_3)$. The obtained vector include into dependence matrix D_{s2} of the second ELN and connect by obtained vector only corresponding nodes K' of this DG.

For example, for N=4 the following nodes pare in the second ELN should be connected: (1,2,3) and (2,3,3); (1,2,4) and (2,3,4); (2,3,4) and (3,4,4);.

The obtained DG of the second ELN is shown in the fig.3.2.

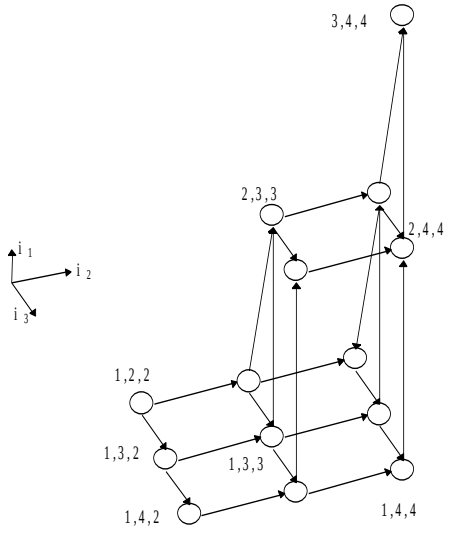


Fig. 3.2. Elementary DGs for second ELN

4. For first ELN with dimension $n_{s1} = 2 < n$ we determine the value of $(n - n_s) = 1$ absent coordinate of its nodes and complete its matrix of the iteration space V_{s1} .

Such coordinate is the coordinate I_3 , and its value for all nodes of the first ELN is equal

$$I_3 = a_3 - c_3 = (I_1 + 1) - 1 = I_1 ,$$

where a_3 and c_3 are the lower limit and step values of the coordinate I_3 (in the second ELN).

Then we complete the matrix V_{s1} and the all obtained vectors-arcs by coordinate I_3. The obtained non-zero vectors we include to the dependence matrix D of the algorithm. As a result, the following matrices V_{s1} and D_{s1} are obtained:

$$V_{s1} = \begin{bmatrix} 1 & N-1 & 1 & I_1 \\ I_1 + 1 & N & 1 & I_2 \\ I_1 & I_1 & 1 & I_3 \end{bmatrix}$$

$$D_{s1} = \begin{bmatrix} 0 & I_1 \\ 1 & I_2 \\ 0 & I_3 \end{bmatrix}$$

5. By means joining the modified matrices of the iteration spaces of the first and second ELN's we consist the iteration space matrix $V(n \times 3)$ of the whole algorithm as

$$V = \begin{bmatrix} 1 & N-1 & 1 & I_1 \\ I_1+1 & N & 1 & I_2 \\ I_1 & N & 1 & I_3 \end{bmatrix}$$

6. Determine the coordinates of the first and last executed nodes of the algorithm DG. They are equal

$$K_1=(1, 2, 1) \text{ and } K_z=(N-1, N, N)$$

respectively.

The coordinates of the remaining DG nodes are derived by sequential modification of the values of the each coordinate I_j on step value c_j . Note that obtained value should be satisfied to the expression (3.6).

7. Construct an *n*-dimensional integer lattice K^n and locate the DG nodes in its vertices in according to their coordinates (I_1, I_2, I_3).

8. In according to the Confirmation 3.4 and its consequence, we determine that the DG has vectors-arcs for transmission the calculated in the first ELN variable $m(I_2 , I_1)$ to the second ELN as argument $m(I_2 , I_1)$, and vectors-arcs for transmission the calculated in the second ELN variable $a(I_2 , I_3)$ to the first ELN as arguments $a(I_2 , I_1)$ and $a(I_1 , I_1)$.

For the first variables pare in according to the Confirmation 3.4 we obtain the next expressions:

$$m_1 = I_2 , m_2 = I_1 , u_1 = I_2 , u_2 = I_1 .$$

Consequently, the values h_1 and h_2 will be equal to

$$h_1 = I_2 - I_2 = 0 \text{ and } h_2 = I_2 - I_2 = 0.$$

In according to the Confirmation 3.1 the value of the variable h_3 is equal to

$$h_3 = c_3 =1.$$

As a result, the vector-arc $d^5 = (0, 0, 1)$ was determined. It corresponds to the transmission of calculated in the node $K=\{I_1, I_2, I_1\}$ variable $m(I_2 , I_1)$ to the node $K'=\{I_1, I_2, (I_1+1)\}$ as the argument $m(I_2 , I_1)$. The obtained vector include into dependence matrix D_{s2} of the second ELN and connect by obtained vector only corresponding nodes K' of this DG.

For example, for N=4 the following nodes pares in the algorithm DG should be connected:

(1,2,1) and (1,2,2); (1,3,1) and (1,3,2); (1,4,1) and (1,4,2);. (2,3,2) and (2,3,3); (2,4,2) and (2,4,3); (3,4,3) and (3,4,4).

Analogously, the vectors-arcs $d^6 = (1, 0, 0)$ were derived, which corresponds to the transmission of the calculated in the node $K=\{I_1, I_2, I_3\}$ variable $a(I_2 , I_3)$ to the node $K'=\{(I_1+1), I_2, I_3\}$ as the argument $a(I_2 , I_1)$, and $d^7 = (1, 1, 0)$, which corresponds to the transmission of the calculated in the node $K=\{I_1, I_2, I_3\}$ variable $a(I_2 , I_3)$ to the node $K'=\{(I_1+1), (I_2+1), I_3\}$ as the argument $a(I_1 , I_1)$. The obtained vectors include into dependence matrix D of the algorithm and connect by obtained vectors only corresponding nodes K' of this DG.

For the case N=4 the following nodes pares in the algorithm DG should be connected:
(1,3,2) and (2,3,2); (1,4,2) and (2,4,2);. (2,4,3) and (3,4,3); (1,2,2) and (2,3,2); (2,3,3) and (3,4,3). The resulting DG for Gauss elimination is presented in Fig.3.3.

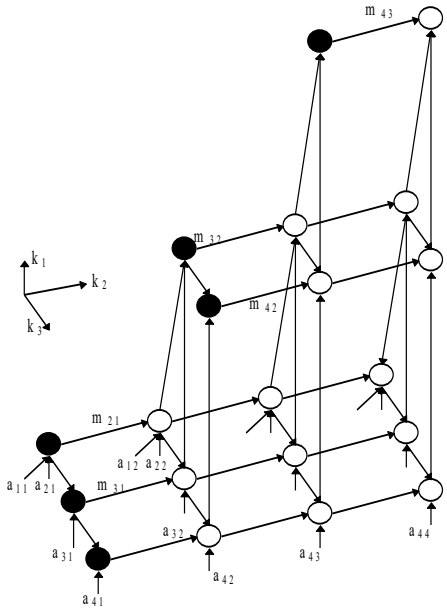


Fig. 3.3. DG of whole Gauss elimination algorithm

The dependence matrix D of this algorithm is given below.

$$D = \begin{bmatrix} 1 & 0 & 0 & 1 & I_1 \\ 0 & 1 & 0 & 1 & I_2 \\ 0 & 0 & 1 & 0 & I_3 \end{bmatrix}$$

3.2. Design of FPGA-based processor array architectures for the solving of main linear algebra tasks

3.2.1. Mapping overview

Architectures of VLSI processor arrays can be designed systematically [3, 7, 8, 22, 26, 26 - 33] using linear (or affine) mappings of algorithms which are expressed by systems of recursive equations or nested loops. In the course of mapping, a given algorithm AL with the dependence graph G is transformed into a set of structural schemes C = <S,T,Φ> of arrays implementing this algorithm, where S is a directed graph called the array structure, T is the synchronization function specifying the computation time of nodes in the DG, and Φ is the set of operation algorithms of PEs.

One of the most promising approaches to mapping recursive algorithms with regular dependencies into processor arrays [31] consists of finding linear mapping operator **F** which transforms the each node **K** of the algorithm DG to the corresponding node of the structure graph *S*:

$$\mathbf{F}: K^n \rightarrow K_F^{m+1}, \ \mathbf{F(K)= F\cdot K,} \ \forall \mathbf{K} \in K^n, \tag{3.9}$$

where m is the dimension of the PA structure *(m+1 ≤ n).*

Operator F represents the *(m+1) x n* matrix and composes of two components: space mapping F_S and time mapping F_T :

$$F = \begin{bmatrix} F_S \\ F_T \end{bmatrix} \in Z^{(m+1)\times n} . \tag{3.10}$$

As a result, the arbitrary DG node $\mathbf{K} \in K^n$ will be carried out in the processor element (PE) with coordinates $F_S \cdot \mathbf{K}$ at the tact number $\mathbf{F_T \cdot K}$.

Note, that the operator **F** should be satisfied to the following conditions:

1. $\mathbf{F_T \cdot d} > 0 , \forall \mathbf{d} \in \mathbf{D}$;

2. $\forall \mathbf{K_1, K_2} \in K^{\mathbf{n}} (\mathbf{K_1 \neq K_2} \Rightarrow \mathbf{F \cdot K_1 \neq F \cdot K_2})$; (3.11)

3. $rank(\mathbf{F}) = m+1$.

Thus, in according to the methodology [31], the set of all possible and nonequivalent allocation mappings $F_S(K)$ satisfying given constraints for links between PEs (which are located in vertices of a lattice $K^m \subset Z^m$) is firstly determined. For each of network topologies S corresponding to this set, an optimal schedule mapping which implements the algorithm correctly is find then. This mapping is constructed as a linear (or affine) function F_T with n unknown coefficients.

Using the existing mapping methods, efficient array architecture for implementation of the algorithm with regular data dependencies have been designed. The next step of the designing procedure consists of deriving the structure of the target application-specific system destined for implementation of several obtained PA architectures (corresponding to the set of selected algorithms). Note, that the number of different algorithms in this set is the main factor which determines such basic parameters of the target system as complexity and uniformity of PE's, their utilization and complexity of inter-processor links. In the case of ASIC-realization, a promising approach to solving of this problem is based on the designing processor arrays for such versatile algorithms which could solve several distinct problems (for example, in linear algebra, Faddeev algorithm [20, 29, 103] is inherently versatile). Only in the case of PA's implementation on the base of programmable devices such as FPGAs a full adaptation of implemented structure, highest hardware utilization and lowest cost/performance ratio may be derived. Therefore, in next sections, the design of the FPGA-based parallel system destined to the realization of main linear algorithms is represented. At first, as a example, the design of PA architecture performing Jordan-Gauss algorithm with the partial pivoting is described. Note, that to derive array architecture with desired features, some purposive transformations of the basic algorithm dependence graph are employed. Since the array architectures obtained in this way feature a strong dependence from sizes of matrices being processed, we show how these architectures should be modified in order to process a large size matrix on fixed-size arrays. At second, we show the different PA architectures for the implementation Faddeev, Cholesky, Householder and Hestenes algorithms. Finally, the FPGA-based structure of the application-specific parallel system

destined to the performing the above algorithms is described. This linear processor array consists of two-ported processor elements with configurable internal structure which are connected into a ring and is characterized by high overall performance, scalability and hardware utilization.

3.2.2. Design of linear processor arrays for the original Jordan-Gauss algorithm

Jordan-Gauss algorithm with partial pivoting is expressed in the form (2.37) (see chapter 2 of this manuscript).

In spite of using **if...then...else** statements in the algorithm (2.37), the order of execution of its operators is unambiguously determined before computations. This allows us to construct its basic dependence graph G_B in accordance to the proposed in the section 3.1 method. Nodes of G_B are distributed in nodes of the three-dimensional lattice $Q1=\{K=(i,j,k): 1 \le i \le N, (i+1) \le j \le (N+i), i \le k \le N+K\}$. This lattice can be visualized as a truncated pyramid possessing a rectangular base with the size of $(N+1)(N+K)$ nodes. The height of the lattice is N units (or layers). The graph G_B is shown in Fig.3.4, where $N=3$, $K=1$. It should be note that the i-th layer of G_B ($i=1,2,...N-1$) is composed of two sublayers for which we assume $z=1$ or $z=2$. We will call these two sublayers pivot or elimination sublayer, respectively. The first sublayer with $z=1$ consists of $(N-i+1)(N+K-i+1)$ subnodes, and corresponds to the selection of the pivot element within the i-th column of the matrix $\boldsymbol{F}^i$, as well as to the described above interchanges of its rows, from the i-th row to the N-th row. These interchanges are carried out under the control of variables v_{ji} generated during the selection process, where ($j=i+1,...,N$). The second sublayer with $z=2$ consists of $N \times (N+K-i+1)$ subnodes, and corresponds to the computation of coefficients m_{ji}, where $j=i+1,...,N+K$, followed by transformations of rows of the joint matrix from the $(i+1)$-st row to the $(N+i)$-th row. The highest N-th layer of G_B is composed of only the elimination sublayer with $N(K+1)$ subnodes.

The data dependencies (or arcs) between nodes of the graph G_B are represented by the five different vectors $\boldsymbol{d_1}, ..., \boldsymbol{d_5}$ which compose the dependence matrix $\boldsymbol{D}$ of the algorithm:

$$D = \begin{bmatrix} d_1 & d_2 & d_3 & d_4 & d_5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & N-i-1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad (3.12)$$

In the graph G_B , the critical paths have the length of N(N+5)/2 +K+1 subnodes. It gives the lower bound for the total computation time t^* required by the algorithm to process a particular matrix $\boldsymbol{F}$. Consequently, when matrices are processed individually, all two-dimensional processor arrays will manifest the low processor elements utilization

$$\eta^* = W/(t^* \cdot M) \approx O\left(N^{-1}\right).$$

Here $W \approx O\left(N^3/2\right)$ and $M \approx O\left(N^2\right)$ is the number of subnodes in the graph G_B and PEs in a 2-D structure, respectively.

Before passing on to the design process for one dimensional (or linear) arrays, we note, that the presence of global dependencies in G_B limits the set of array structures S with fully local communication between PEs. Indeed, for 2-D structures, only projection of the graph along the vector ***r**=[0 1 0]*** satisfies this condition. Hence, to simplify the design of 1-D array structures, we transform the three-dimensional graph G_B into a 2-D graph G_1 by projecting G_B along the vector ***r****.* As a result, all nodes lying at a straight line parallel to ***r**** merge into a single macronode, which represents a macrooperation performed on an entire column of $\boldsymbol{F^i}$. Then we again transform the graph G_1 by composing the pivot and elimination sublayers of the *i*-th layer of G_1 into one layer, where (*i=1,2,...N-1).* Having done this, we get the graph G_2 which is shown in the Fig.3.5,a , where *N=6, K=1.*

The set of 1-D structures of PAs with fully local interconnections corresponds to the following set of mapping operators $\boldsymbol{F_S}$:

$$F_S = \begin{bmatrix} f_{11} & f_{12} & f_{13} \end{bmatrix} \in \left\{ \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \right\} \quad (3.13)$$

One of them is the structure S_1 , which is shown in Fig.3.5,b , where *N=6, K=1*. This structure, which corresponds to the projection of G_1 along *i*-axis, contains *N* PEs of the first type, i.e. having a division unit in addition to a multiplication-addition unit, and *K* PEs of the second type, i.e. without a division unit.

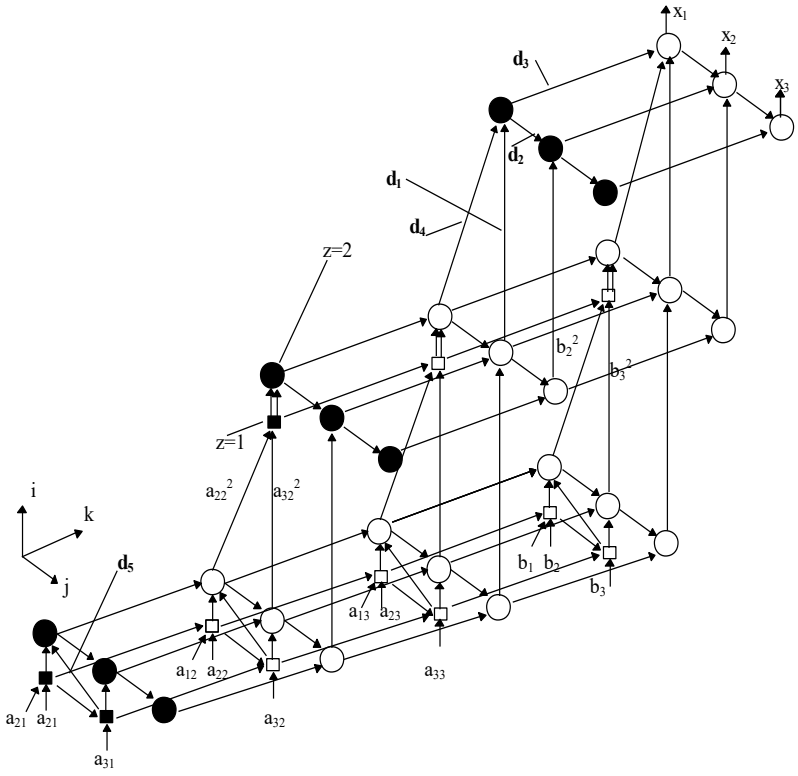


Fig.3.4. Basic dependence graph of Jordan-Gauss algorithm

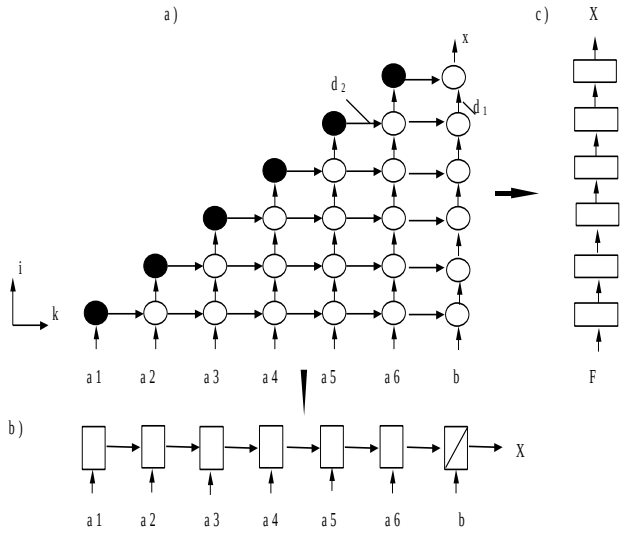


Fig.3.5. 2-D DG of Jordan-Gauss algorithm and 1-D architectures of processor arrays

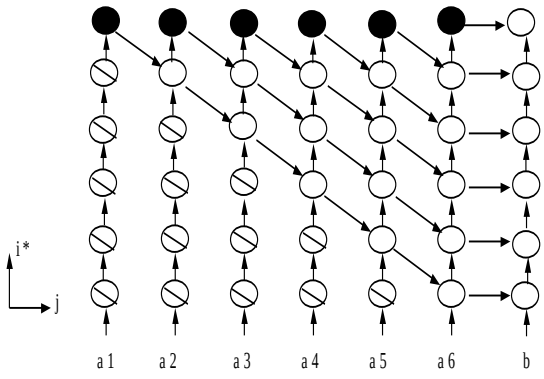


Fig.3.6. Transformation of Jordan Gauss algorithm DG

The drawbacks of the structure S_1 are comparatively large numbers of PEs and I/O channels, as well as, the presence of N PEs containing division units. These shortcomings can be eliminated by projecting the graph G_2 along the vector k-axis. This projection results in the structure S_2 , which has only N PEs (see Fig.3.5,c). Moreover, only the first and the last PEs of this structure perform I/O operations. However, the limitation of the structure S_2 is that all its PEs must perform divisions in addition to multiply-add operations.

In a order to obtaining the such array structure which minimizes the number of PEs containing a division unit, we try to transform the triangular part of the graph G_2. We place all diagonal macronodes of the graph G_2 along k-axis, and then redraw this graph preserving all interconnecting between its macronodes. This transformation can be thought as a rotation of the triangular part of G_2 by an angle of 45° clockwise. As a result, coordinates (i,k) of macronodes of the triangular part are changed according to the following formulae: $k^*=k$, $i^*=N-k+i$, where $i=1,2,...,N$, $k=i, i+1,...,N$. Then we complete the obtained graph with „empty" macronodes, which provide the input of matrix $\boldsymbol{F}$ in accordance with Fig.3.6, where the resulting graph G_3 is depicted. After projecting it along k^*-axis, we obtain the structure S_3 shown on the Fig.3.7. Lastly, to complete the design of a structural scheme C_3 which corresponds to both the graph G_2 and structure S_3 , we derive a schedule mapping $\boldsymbol{F_T}$, using the mapping methodology [31]. Aiming at the minimization of the algorithm execution time, we obtain the following schedule:

$$\boldsymbol{F_T} = N \cdot i^* + j + (N+1)k^* + z(N-1) + const\ . \qquad (3.14)$$

The scheme C_3 is a linear array with N PEs; only the last PE contains a division unit, while the rest of PEs , which are of the same type, are not provided with it. The internal structures of the k^*-th ($k^* \neq N$) and the N –th PEs are detailed in Fig.3.8,a and Fig.3.8,b respectively, where

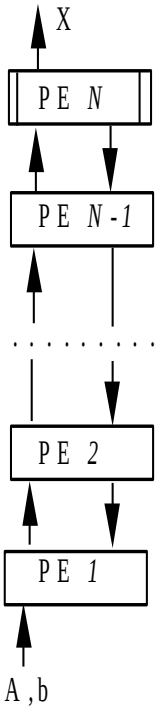


Fig. 3.7. Fixed-size array structure for Jordan-Gauss algorithm

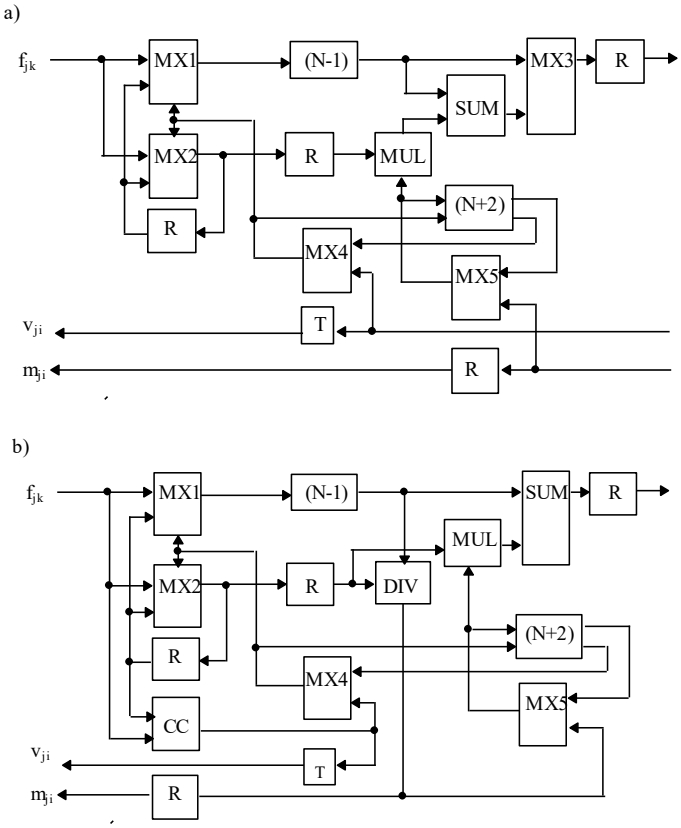


Fig. 3.8. Internal structure of i-th and n-th PEs

(N-1) and (N+2) are FIFO-buffers with corresponding lengths, MX denote multiplexes, R denote registers, CC is a comparer, DIV and MUL denote a division and multiply units. Because a new matrix $\boldsymbol{F}$ can be processed as soon as the input of the previous matrix $\boldsymbol{F}$ is completed, the above scheme is characterized by the pipelining period of $t=(N+1)(N+K)$ steps and the asymptotic processor utilization $\eta \approx 0.5$ for $K=1$.

A basic requirement in practical system designs for linear algebraic problems is an ability to process large size matrices on processor arrays with a fixed number of PEs

[3,29]. To provide this ability, two partitioning methods [3] are usually used: locally sequential globally parallel (LSGP) method and locally parallel globally sequential (LPGS) method. Both of them are based on the decomposition of a dependence graph (DG) of an algorithm into a set of regular subgraphs, but differ in the way how these subgraphs are mapped onto resulting structural schemes. In the LSPG method, one subgraph is mapped to one PE, and each PE sequentially executes the nodes of corresponding subgraph. Therefore, an additional local memory within each PE is needed. To avoid this disadvantage, one subgraph is mapped to one array in the LPGS method. All nodes within one subgfaph are processed concurrently, while all subgraphs are processed sequentially. As a result, all intermediate data which correspond to data dependencies between subgraphs can be stored in buffers outside the processor array. We employ this scheme in order to implement the ABFT Jordan-Gauss algorithm on a linear array with $n<N$ PEs, where n is a fixed number.

Starting with the graph G_2, we try to decompose it into a set of $s=]\ N/n[$ subgraphs having the „same" topology, where $]x[$ denotes the nearest integer equal to or greater then x. As evident from Fig.3.5, this can be done only if we „cut" the graph G_2 using a set of straight lines parallel to k-axis. These lines decompose the graph G_2 into q regular subgraphs with n layers each, where $q=1,2,...,s$. Then, the above described rotation of the triangular part of G_2 by the angle of 45° is individually used for every q-th subgraph. Lastly, after completing each of subgraphs with „empty" macronodes, a set of s subgraphs with the „same" topology is obtained (see Fig.3.9). Note, that such decomposition allows to reduce the number of „empty" nodes in G_3 from $3N(N - 1)/2$ to $3n(n-1)(s/2)$ nodes. Then we project each resulting subgraph onto i*-axis in order to obtain a fixed-size array structure shown in Fig.3.9. The total execution time T of the Jordan-Gauss algorithm realization is equal to

$$T = N \cdot (n+1) + \sum_{s=1}^{[N/n]} \left((N+2) \cdot (N + K - (s-1) \cdot n) \right) \qquad (3.15)$$

time steps and the asymptotic processor utilization $\eta \approx 1$ for $N>>n$.

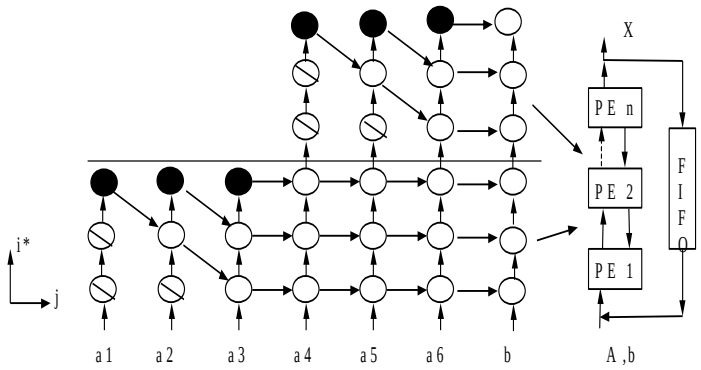


Fig.3.9. Partitioning of the Jordan –Gauss DG

3.2.3. Design of the fixed-size linear processor array for the fault-tolerant Jordan-Gauss algorithm

It was be shown at the chapter 1 of this manuscript, that using algorithm-based fault tolerance (ABFT) methods allows to derive the application-specific device which will be tolerant to the transient hardware errors occurred during algorithm implementation without using others fault tolerance methods. In other words, by means mapping the fault tolerant version of the applied algorithm (Jordan-Gauss algorithm here) will be obtained tolerant to the transient error processor array architecture. Therefore, we try do it.

The two-dimensional (2-D) DG G_4 of the fault-tolerance version Jordan-Gauss algorithm is shown in fig.3.10, where $N=6$, $K=3$. Nodes of G_4 are distributed in nodes of the 2-D lattice $Q_4=\{K=(i,g): 1 \le i \le N, (5i-4) \le g \le (5N+K-1)\}$. It should be noted that i-th layer of G_4 $(i=1,...,N)$ corresponds the i-th step of algorithm. There are six kind of nodes in the DG G_4. The nodes with coordinates $(i,5i-4)$, $(i,5i-2)$ and $(i,5i)$, $i=1,...,N$, (which are marked by ▤) correspond to, firstly, the calculations of the values *CS* and *WCS* of the i-th column and i-th row of matrix $\mathbf{F}^i$ and i-th column of matrix **M** respectively and, secondly, the definition of erroneous elements within these columns. The nodes with coordinates (i,5i-3) (which are marked by ●) correspond to the correction of the erroneous elements and selection of a pivot element within i-th column of the matrix $\mathbf{F}^i$ .The nodes with coordinates $(i,5i-1)$ (which are marked by □) correspond to the correction of the erroneous elements within i-th row (i.e. leading row) of matrix $\mathbf{F}^i$ and the calculation of the elements m_{ji} of the matrix **M**. The nodes with

coordinates $(i,5i+1)$ (which are marked by ■) correspond to the correction of the erroneous elements m_{ji} and the calculation of the elements $f_{j(i+1)}^{i+1}$, j=i+1,i+2,...,N+i . The nodes marked by • correspond to the calculation of the elements f_{jk}^{i+1} of the j-th column or k-th row of the matrix $\mathbf{F^{i+1}}$ respectively. The nodes marked by ○ note the " empty" nodes (no operations) and are included for the elimination of the global dependencies vectors (or arcs) in the graph. As it will be below shown, these nodes are few decrease of the PEs utilization in the proposed fixed size array.

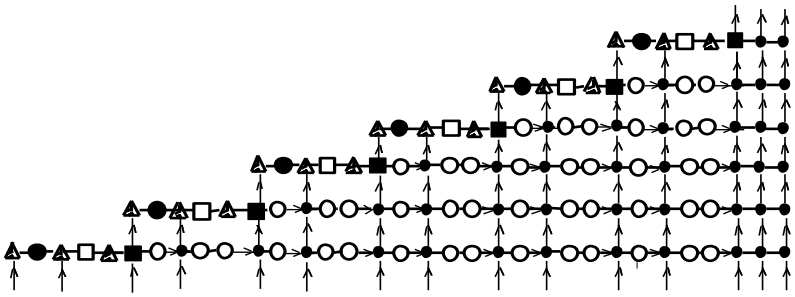


Fig. 3.10. 2-D DG of the fault-tolerant version of Jordan-Gauss algorithm

Thus, this DG is liked to the two-dimensional DG G_2 of the original Jordan-Gauss algorithm.

Therefore, in order to deriving of the linear fixed size array architecture for fault tolerant version of Jordan-Gauss algorithm, we employ the locally parallel globally sequential scheme of partitioning. Thus, starting with the graph G_4, we try to decompose it into a set of *s* subgraphs having the "same" topology. Note, that this can be done only if the graph G_4 is "cut" using a set of straight lines parallel to **g**-axis, as shown in fig.3.10. These lines decompose the graph G_4 into the set of regular subgraphs G^h with *n* layers each (see fig.3.11),

where $h = 1,...,s$. Note, that such decomposition allows to reduce the number of "empty" nodes in G_4 from $3N(N - 1)/2$ to $3n(n-1)*s/2$ nodes.

Then we project each resulting subgraph G^h onto **i**-axis in order to derive a fixed-size array

architecture shown in fig.3.11 (at right). This array, which is provided with an external FIFO buffer, featured simple scheme of fully local communications and a few number of I/O channels. The total execution time T of performing of the fault tolerant version of Jordan-Gauss algorithm is equal to

$$T = \sum_{i=1}^{N/n} (N+K-(i-5)n)(N+n-i*n)$$

times steps and processor utilization $\eta = W/(T \cdot M)$ where $M=n$ is the number of the PEs in the array, and W is the computational complexity of algorithm,

$$W = N^3/2 + N^2 (K+2,5) + 2NK$$

operations such as multiplication with addition.

Using these formulae and supposing, for example, that $K=1$ and $K=N$ for case $s = N/n = 30$ it is obtained

$$\eta = 0,72 \quad \text{and} \quad \eta = 0,84$$

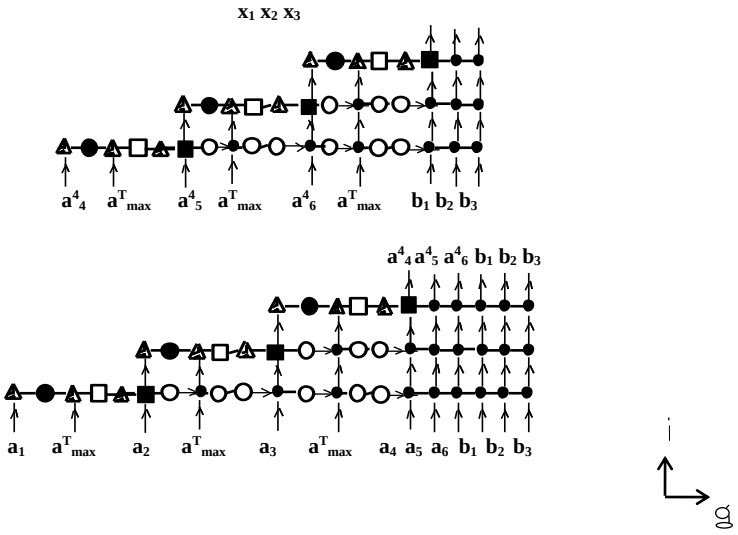


Fig.3.11 Partitioning of the fault-tolerant Jordan-Gauss algorithm

respectively. Note, that with an increase in parameters K or/and s, the value η also increases.

3.2.4. Deriving of FPGA-based parallel system architecture for the realization of main linear algebra algorithms

The basic dependence graphs of the fault-tolerant versions of following main linear algebra algorithms were obtained using our method for deriving dependence graphs of recursive algorithms:

- Gauss elimination algorithm for matrix decomposition and solving linear systems;
- Choleski algorithm for LL^T-decomposition of symmetric matrices;
- back substitution algorithm for solving linear systems with triangular matrices;
- Jordan-Gauss algorithm for solving linear equations or matrix inversion;
- Faddeev algorithm for solving matrix equations of the type $X=C\cdot A^{-1}\cdot B+D$;

Note, that this graph correspond also some others LA algorithms, for example, Householder reflections and QR-algorithms.

The analysis of the all basic graphs of the above algorithms shown that they can be represented by the two-dimensional graph G_3 (see Fig.3.12). This means that the all mentioned algorithms may be realized on the fixed size processor array architecture represented in the Fig.3.12 (at right). Note, that in according to the carried out algorithm, the external memory block must to perform here a RAM-function or a FIFO-function. Moreover, the internal structure of all processor elements it is the different for the different algorithms.

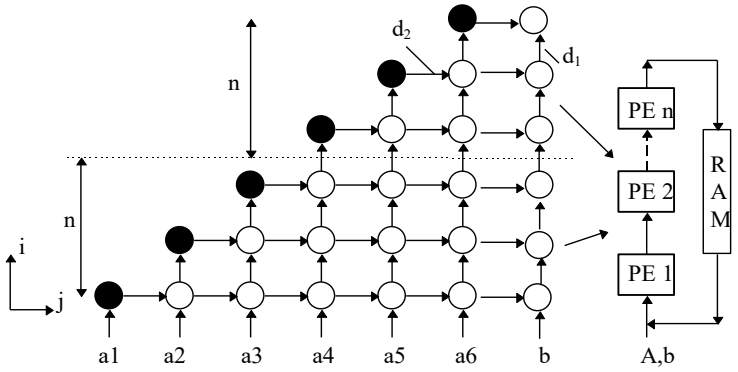


Fig.3.12

The internal structure of the processor elements for realization of Cholesky algorithm is shown in Fig.3.13, while the internal PE structure of the array for realization of Jordan-Gauss, Faddeev and back substitution algorithms is represented in the Fig.3.14. Here the blocks denoted (N), (M+1)

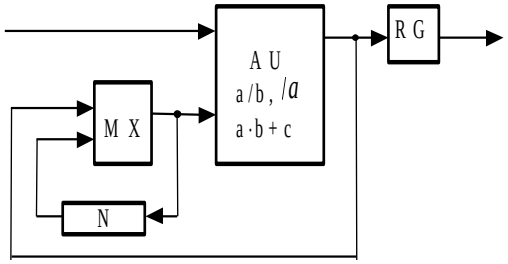


Fig.3.13.

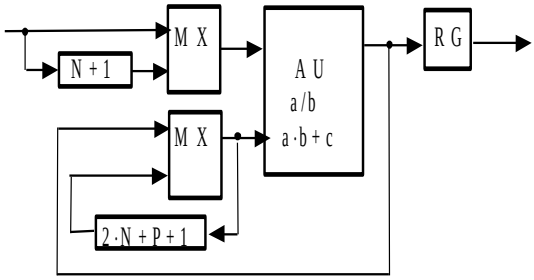
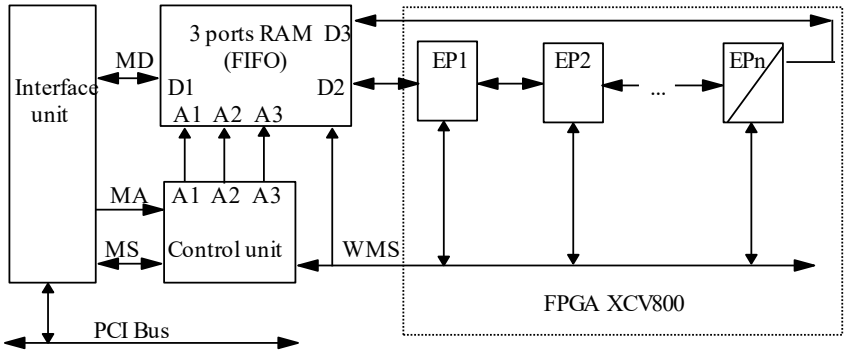


Fig.3.14.

and (2N+P+1) are FIFO-buffers with corresponding lengths (where P=1 for the Jordan-Gauss algorithm), MX denote multiplexers, T denote registers and AU denote a arithmetic units. These AU must perform the multiply-add or division or (only for Cholesky algorithm) square rooting operations.

Therefore, the processor array architecture represented in the Fig.3.12 was selected as functional prototype of the FPGA-based parallel system for realization of linear algebra algorithm. As a result the structure of this system was derived. It is shown in the Fig.3.15, where MD, MA, MS and WMS are data bus, address bus, control bus and internal control signal bus respectively. This array processor has a linear ring structure with three-port RAM unit for data feedback. The system is connected by the ISA or PCI bus interface unit (IU) to the external world (or host device, for example, to the AT

compatible computer). The input data is loaded from the interface unit into external RAM unit. Depending on the implemented algorithm, system control unit (CU) generates three groups of address signals to the RAM unit in such a way, that this memory unit performs the functions RAM or FIFO.



Rys.3.15.

In the Fig. 3.16 is represented the more detailed realization of the external memory block. It consists of three one-port RAMs with complex multiplexer MUX. In according to the control signals X1 and X2 (obtained from the CU) it connects one of RAM unit (for example, RAM1) to the data bus of the IU for input data load or resultant data output. At the same time, MUX connects the other RAM unit (for example, RAM2) to the input of the first PE of array for the data input, and the third RAM unit (for example, RAM3) to the output of the *n*-th PE for the data output. Depending on the implemented algorithm, the control unit also performs the load of the selected configuration data file from the configuration memory unit PROM to the FPGA-chips. Note, that multiplexer MUX, IO and CU units may be also realized on the FPGA or CPLD chips, for example, XC95000 family , while the processor array may be realized on the one or more FPGA chips of Virtex family. Note, that depending on the implemented algorithm, the complexity of array PEs is different. Therefore, for realization of different algorithms the different number of PEs must be generated in the FPGA-chips. This should be take to attention in the configuration data files saved in the PROM.

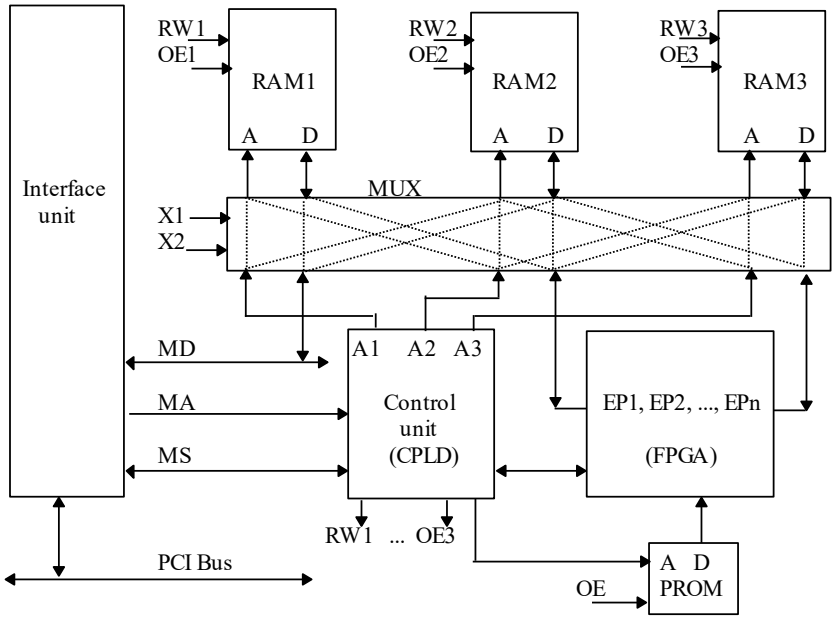


Fig.3.16

Thus, implementation application-specific parallel systems in FPGA has a set of advantages, such as full adaptation of implemented in FPGA structure to the applied fault-tolerant algorithms, high performance, achieving high rate of calculating precision, reducing both the way from idea to the market and development costs. Moreover, only in the case of system implementation on the base of FPGA, a highest hardware utilization, fault tolerance and lowest cost/performance ratio may be derived.

3.3. Conclusions to the chapter 3

1. The new method for the construction of the lattice DGs of algorithms given by nested loops has been proposed. In a contrast with known analytical methods, the proposed method is more simple and feasible for the implementation in CAD systems, and allows operating with a wider class of algorithms such as, for example, non-uniform recursive algorithms corresponding to non-perfect (or composite) loop nests.

2. The basic dependence graph of the Gauss elimination, Jordan-Gauss, Choleski, Faddeev, back substitution and some others algorithms and their fault-tolerant versions

were obtained using proposed method for deriving DG of regular algorithms for further mapping into corresponding processor array architectures.

3. The processor array architectures performing fault-tolerant version of Jordan-Gauss algorithm with the partial pivoting, Cholesky, Gauss elimination and back substitution algorithms has been designed. Note, that in the order to deriving of the array architectures with desired features, some purposive transformations of the basic algorithm dependence graphs are employed. Since the arrays architectures obtained by mapping of the corresponding DGs feature a strong dependence from the input matrix sizes, we showed how these architectures should be modified in order to process an arbitrary large task size on fixed-size arrays.

4. Based on the derived arrays architectures, the structure of the application-specific parallel system destined to the fault-tolerant implementation of these algorithms was obtained. The system consists of two-ported processor elements with configurable internal structure which are connected into a ring and is characterized by high overall performance, scalability and hardware utilization. Thus, was proved the confirmation, that using FPGA-chips and libraries of files with configuration data for these chips, it is possible to construct the fast adapted (to the implemented algorithms) application-specific system with high performance and lowest cost/performance ratio.

5. Implementation application-specific parallel systems in FPGA has a set of advantages, such as full adaptation of implemented in FPGA structure to the applied fault-tolerant algorithms, high performance, achieving high rate of calculating precision, reducing both the way from idea to the market and development costs. Moreover, only in the case of system implementation on the base of FPGA, a highest hardware utilization and fault tolerance and lowest cost/performance ratio may be derived.

CHAPTER 4. INVESTIGATION OF THE CURRENT-MODE GATES AND LOGIC AND DESIGNING THE BASIC BLOCK OF THE FPGA-CELLS

It was be underlined in the chapter 1, that the serious problem in the modern mixed analog-digital systems (ADS) is the noise immunity provision in the case of placing together the analog and digital circuit parts on a common chip surface. One of the methods for the solution of this problem is the implementation of the digital part of the ADS with the current mode gates [1,2]. Due to the nearly constant significance of the power supply current at the different gate states, the level of its noise is essentially lower in comparison with the classical (voltage) type of gates. Besides, based on the current-mode gates the lower hardware overheads digital circuits may be designed (see, for example, [3,4,5]). Therefore in this chapter, the problem of designing digital circuits based on the current-mode gates is considered. Firstly, the logical properties of the current-mode logic and the expressions to conversion of the Boolean functions to the current-mode ones are represented. The analysis of these properties and expressions causes to the extension of the set by means including the double-inverter and half-inverter gates. Moreover, the several identities of the current-mode logic were derived. Then, based on these identities and expressions, the approaches to minimizing the current logic functions and to designing digital current-mode circuits were proposed. By means applying the proposed approaches, the functional schemes of the current-mode one-bit adders, the four-bit fast adder unit with the parallel carry bit propagation and the basic block of FPGA-cells - look-up-table (LUT) were designed. The obtained circuits are characterized by smaller (up to 35%) hardware overheads in comparison with the similar circuits based on the classical voltage type of gates.

4.1. Current-mode gates and logic

There are three types of the main operation in the current mode logic: arithmetic addition, arithmetic subtraction and inversion [3].

The addition operation corresponds, at the physical level, the addition of currents, each from which represents the significance of the corresponding operand. In the functional

level this means the association of all operand lines into one node. Analogously, a arithmetic subtraction operation in this techniques, at the physical level, is performed by the subtraction of currents. Therefore, in the functional level, this means (for example, for expression *(X-Y)*) the association the line of the operand *X* with the output of the anti-inverter gate connected to the line of the operand *Y*, where Y $\in$ {0,1}.The examples of schemes for the implementation of the operations *(X+Y)* and *(X-Y)* are shown in the fig. 4.1.

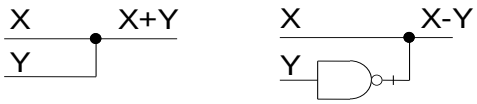


Fig. 4.1. Realizing arithmetic addition and subtraction operations with the current-mode technique

There are only several types of the inverter gates in the current-mode technique [3,4]. For example, the inverter gate of the first type is named an inverter. Its graphical image and the carried out logical function are shown in the fig. 4.2.

$$\overline{X} = \begin{cases} 1 \text{ if } X = 0,-1,-2,... \\ 0 \text{ if } X = 1,2,3,... \end{cases}$$

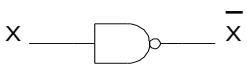


Fig.4.2. Current-mode inverter

Gates of the second type are named the anti-inverters. Their graphical image and the carried out logical function are shown in the fig. 4.3.

$$\hat{X} = \begin{cases} 0 \text{ if } X = 0,-1,-2,... \\ -1 \text{ if } X = 1,2,3,... \end{cases}$$

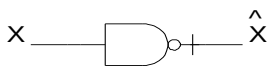


Fig.4.3. Current-mode anti-inverter

The other types of the current-mode inverter gates will be represented in the next paper sections.

It follows from expression fig. 4.1, fig. 4.2 and fig. 4.3, that arbitrary logical variable in this logic is (in a general case) multivalued one. This means, that appearance of such variable at the any input (output) of the current-mode gate corresponds the appearance at this input (output) the respective level of the current in the relative digits [2]. Moreover, the significance of the variable (or function) appeared on any gate output belongs to the set {-1, 0, 1}, while the significance of the variable appeared on any gate input (for example, as a result of an addition or subtraction operations) belongs, in a general case, to the set of integer numbers from the interval] -∞,∞[.

Due to such logical properties, the Boolean algebra identities are not suitable for the current-mode algebra, however, all Boolean operations can be realized with current-mode gates.

Confirmation 4.1. The arbitrary binary (Boolean) function can be realized with current-mode gates using the expressions (4.1) and (4.2) for the conversion of the main Boolean operations into corresponding current-mode logical functions.

Proof. If values of the logical variables or functions a and b belong to the set {0,1} then following expressions for the conversion of the main Boolean operations into corresponding current-mode logical functions are corrected:

$$a \cdot b = \overline{\overline{a} + \overline{b}}\ ,$$

$$\overline{a \cdot b} = \overline{\overline{\overline{a} + \overline{b}}}\ , \qquad (4.1)$$

$$a \vee b = \overline{\overline{a + b}}\ ,$$

$$\overline{a \vee b} = \overline{a + b}\ , \qquad (4.2)$$

where symbols „·”, „V” and „+” correspond to operations AND, OR and arithmetic addition respectively.

Besides, it is well known, that any binary logical function a of arbitrary complexity in the Boolean algebra may be represented by a logical expression, which uses only AND, OR and NOT, or NOR, or NAND operations. This means, that an arbitrary logical function may be transformed from the Boolean algebra to the corresponding current-

mode logical function by applying the expressions (4.1) and (4.2). Then, using the corresponding current-mode gates, the current-mode circuit for realization of the target logical function may be constructed. The confirmation is proved.

Furthermore, the current-mode algebra also has own logical identities. Several from these identities are presented below, were *a, b* and *c* belong to the set {0,1}:

1. $a+b=b+a$
2. $(a+b)+c=a+(b+c)$
3. $a+0=a$
4. $a+\bar{a}=1$
5. $\bar{\bar{a}}=a$
6. $\overline{a+1}=0$
7. $\overline{a+a}=\bar{a}$
8. $\overline{\overline{a+a}}=a$
9. $\overline{1-a}=a$
10. $\overline{a-1}=1$
11. $\overline{\overline{a+a-b}}=a$
12. $\overline{\overline{a+b}}=\overline{\bar{a}-b}=\overline{\bar{b}-a}$
13. $\overline{\overline{a+b+c+\ldots}}=\overline{\bar{a}-b-c-\cdots}$
14. $\overline{a+b+c+\ldots}=(\bar{a}+\hat{b}+\hat{c}+\cdots)$
15. $a-b=a+\hat{b}$, (4.3)

where symbol „ - ” corresponds to the arithmetic subtraction operation.

4.2. Approaches to the minimization of current-mode logical functions and designing of binary current-mode digital circuits

The first approach to the minimization of the current-mode logical functions follows from the expressions (4.1), (4.2). It consists of the deriving the corresponding logical expression for a target circuit firstly into Boolean algebra. Note, that the arbitrary from the known minimization methods may be used in this step (for example Vetch-Karnaugh’s diagrams). Then by means expressions (4.1) and (4.2) the transformation of obtained expressions in the current-mode ones is carried out.

The obtained current-mode expressions are not always optimized. Therefore, the criterions which should be applied during minimization of the logical expression into Boolean algebra will be now considered.

It follows from the identities (4.3), that:

1) if the logical function X in the Boolean algebra is the function from the *n* logical variables, then corresponding current-mode function also will be the function from the (maximum) same *n* logical variables;

2) the Boolean function NOR has the simplest implementation in the current-mode logic;

3) if the variables *a* and *b* from the expressions (4.3) are the logical functions from the same set of the variables *(x1,x2,...,xi)* and

$$a \bigcap b = \varnothing ,$$

(i.e. variables *a* and *b* are not equal to the logical „0" or „1" simultaneously) then the expressions (4.1), (4.2) are represented by the following more simple expressions:

$$\begin{aligned} a \cdot b &= \overline{\bar{a} + \bar{b}} , \\ \overline{a \cdot b} &= \bar{a} + \bar{b} , \\ a \vee b &= a + b , \\ \overline{a \vee b} &= \overline{a + b} . \end{aligned} \tag{4.4}$$

Thus, in the order to the reduction of the complexity of logical current-mode functions the following criterions should be applied during minimization of the corresponding logical expressions into Boolean algebra:

1) minimum number of the function arguments *(x1,x2,...,xi)*;

2) maximum number of such functions *a,b,c,...* for which the following identity is true:

$$a \bigcap b \bigcap c \bigcap \ldots = \varnothing \tag{4.5}$$

3) maximum number of the NOR and OR operations.

The second approach is is based on the following confirmation.

Confirmation 4.2. The arbitrary current-mode logical function may be represented as an algebraic sum of the set of several others current-mode logical functions.

The proof of this confirmation immediately follows from the fact that the operations of

arithmetic addition and subtraction are enabled in the current-mode algebra, and from the identities (4.3).

Thus, the second rule is consisted of (before realization) the minimization of current-mode target functions by means its representation as an algebraic sum of the set of more simple functions (named „radix" functions) which are selected in such a way, that the simplest resultant expression will be derived.

This approach may be better than the first one in the case of the minimization i realization simultaneously of the several logical current-mode functions which have the common arguments *(x1,x2,...,xi).*Then the first function (more simple) is minimized by means applying the first or the second from the proposed rules, and the each next function uses the previous obtained functions as the „radix" functions. Note, that the next reduction of the current-mode expression complexity may be derived by the applying the identities (4.3).

And the third proposed approach is consists of the applying the identities (4.3) to the minimization of the logical functions into current-mode algebra. This approach may be used for the next reduction of the current-mode expression complexity after applying the first or/and the second above discussed approaches.

The correctness and efficacy of the proposed approaches to the minimization of the current-mode logical functions will be shown below at the example of the adder circuits designing.

Based on the represented in the last chapter approaches to the minimization of the current-mode functions the following approach to the designing of binary digital circuits with the current-mode gates may be proposed.

In the case, when the target circuit must realize only one logical function then above approach consists of the following steps:

1) to derive the logical expression for the target function in the Boolean algebra;

2) by means known minimization methods (for example, Veitch-Karnaugh's diagrams) to obtain a such logical expression (for target function) which consists of:

- - minimum number of the arguments *(x1, x2, ... , xi)*;
- - maximum number of such functions *a,b,c,...* for which the following identity is true:

$$a \cdot b \cdot c \cdot \ldots = 0 \qquad (4.6)$$

3) by means the De-Morgan's theorem to substitute the all AND-functions to the corresponding NOR functions in the target function expression;
4) by means the identities (4.1), (4.2) and /or (4.4) to derive the current-mode expression for the target function from the Boolean one;
5) by means the identities (4.3) to minimize the number of the inversion and anti-inversion operations in the obtained current-mode expression of the target function;
6) based on the obtained optimized expression and corresponding current-mode gates to design the structural scheme of the target circuit.

In the case, when the target circuit must realize the several logical functions then above approach consists of the following steps:
1) to derive the logical expressions for the all target functions in the Boolean algebra;
2) by means known minimization methods (for example, Veitch-Karnaugh's diagrams) for each target function to obtain such logical expression which consists of the maximum number of the common AND-functions;
3) by means the above represented approach to minimize the first target function (more simple);
4) each next function to represent as algebraic sum the some previous target functions and absented (or excessive) AND-functions;
5) for each target function by means the De-Morgan's theorem to substitute the all AND-functions to the corresponding NOR-functions in the function expression;
6) by means the identities (4.1), (4.2) and (4.4) to derive the current-mode expression for the each target function from the Boolean one;
7) by means the identities (4.3) to minimize the number of the inversion and anti-inversion operations in the obtained current-mode expressions for the each target function;
8) based on the obtained optimized expressions and corresponding current-mode gates to design the structural scheme of the target circuit.

Example. Designing one-bit adders with the current-mode gates

One-bit adder is a combinatorial circuit implementing the function of addition of two input operands a_i, b_i and input carry bit c_i . Besides, an adder has outputs of sum s_i and output carry bit c_{i+1}. An adder is described by truth table 1 and Vetch - Karnaugh's diagrams D1 - D2 (see fig. 4.4). By means these diagrams the following expressions for the presentation of the sum s_i and output carry bit c_{i+1} in the Boolean basis may be obtained:

$$s_i = a_i b_i c_i \vee a_i \overline{b_i} \overline{c_i} \vee \overline{a_i} b_i \overline{c_i} \vee \overline{a_i} \overline{b_i} c_i,$$

$$\text{and} \quad c_{i+1} = a_i b_i \vee a_i c_i \vee b_i c_i \tag{4.7}$$

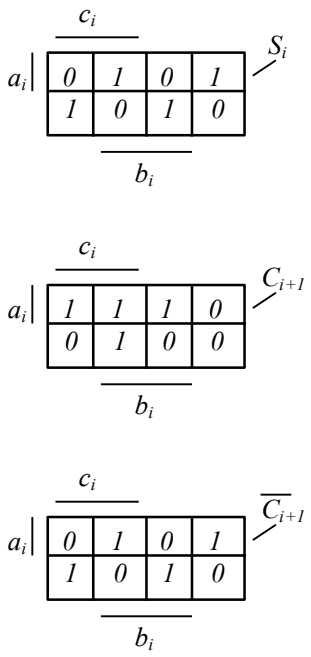


Fig.4.4. Vetch - Karnaugh's diagrams of the functions s_i and c_{i+i}.

Using the expressions (4.1) and (4.2), these functions in the current-mode algebra may be represented by following expressions:

$$s = \overline{\overline{a_i} + b_i + c_i} + \overline{a_i + \overline{b_i} + c_i} + \overline{a_i + b_i + \overline{c_i}} + \overline{\overline{a_i} + \overline{b_i} + c_i} \tag{4.8}$$

$$\text{and } c_{i+1} = \overline{\overline{a_i} + \overline{b_i} + \overline{a_i} + \overline{c_i} + \overline{b_i} + c_i}. \tag{4.9}$$

Note, that the corresponding of expressions (4.8) and (4.9) functional scheme of the one-bit adder consists of 14 current-mode inverters [3].

TABLE 4.1. Truth table of an adder

a_i	b_i	c_i	$\bar{a}_i$	$\bar{b}_i$	$\bar{c}_i$	$\hat{c}_i$	$\bar{c}_{i+1}$	c_{i+1}	s_i
0	0	0	1	1	1	0	1	0	0
0	0	1	1	1	0	0	1	0	1
0	1	0	1	0	1	0	1	0	1
0	1	1	1	0	0	0	0	1	0
1	0	0	0	1	1	-1	1	0	1
1	0	1	0	1	0	-1	0	1	0
1	1	0	0	0	1	-1	0	1	0
1	1	1	0	0	0	-1	0	1	1

In order to deriving the more simple adder circuit the above proposed approach to the design of the digital current-mode circuits was used. As a result, the following expression of the function output carry bit c_{i+1} was derived:

$$c_{i+1} = \overline{\bar{a}_i + \bar{b}_i + \hat{c}_i}. \tag{4.10}$$

For the synthesis of the function s_i as one of the arguments of the algebraic sum the obtained function c_{i+1} was used (in according to the proposed approach). It follows from the diagrams D1-D3 that the function s_i may be derived by means addition to the function $\overline{c_{i+1}}$ the function $\overline{\bar{a}_i + \bar{b}_i + \bar{c}_i}$ and then subtraction the function $\overline{a_i + b_i + c_i}$ from the function($\overline{c_{i+1}} + \overline{\bar{a}_i + \bar{b}_i + \bar{c}_i}$). As a result, the function s_i will be represented by following expression:

$$s_i = \overline{c_{i+1}} + \overline{\bar{a}_i + \bar{b}_i + \bar{c}_i} + \overline{a_i + \hat{b}_i + c_i}\,. \tag{4.11}$$

Note, that corresponding to the expressions (4.7) and (4.8) the functional scheme of the one-bit adder is consisted of the eight inverter gates.

The next reduction of the complexity of the function s_i will be performed by applying for this function the following identity:

$$\overline{\overline{a+b+c+\ldots}} = \overline{\overline{a}-b-c-\cdots} \ .$$

As a result, the function s_i will be represented by following expression

$$s_i = \overline{c_{i+1}} + (\overline{a_i} + \hat{\hat{b}}_i + \hat{c}_i) + \overline{\overline{a}_i + \overline{b}_i + \overline{c}_i} \qquad (4.12)$$

These expression determines (together with the expression (4.10)) the 7-gates versions of the one-bit adder correspondingly. This adder is represented on fig. 4.5.

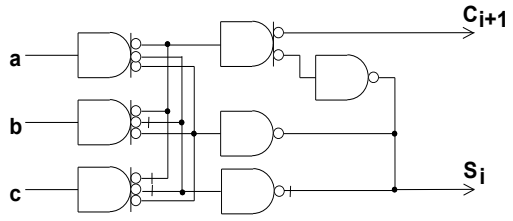


Fig. 4.5. The 7-th gates version of the one-bit adder

Note, that the last version of the one-bit adders is characterized on 20% smaller hardware overheads in comparison with its prototypes based on the classical voltage type of gates.

4.3. Ways to the further reduction of the current-mode functions and circuits complexity: introducing of the new types of the current-mode gates

The analysis of the expressions (4.1) and (4.2) shown, that the Boolean operations AND, NAND and OR are difficulty realized by means the current-mode inverter and anti-inverter gates. Therefore the two new types of the current-mode inverter gates were constructed for the next reduction of the current-mode functions complexity. The inverter

gate of the first type is named an double-inverter. Its graphical image and the carried out logical function are shown at the fig. 4.6.

$$\overline{\overline{X}} = \begin{cases} 0 \text{ if } X = 0,-1,-2,... \\ 1 \text{ if } X = 1,2,3,4,... \end{cases}$$

Fig.4.6. Current-mode double-inverter gate

Using double-inverter gate for example, for the realization of the Boolean operation OR the following circuit may be derived:

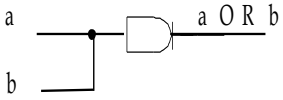


Fig. 4.7. Realization of the OR operation

Gates of the second type are named the half-inverters. Their graphical image and the carried out logical function are shown at the fig. 4.8.

$$\overleftrightarrow{X} = \begin{cases} 1 \text{if } X = 0 \\ 0 \text{ if } X \neq 0 \end{cases}$$

Fig.4.8. Current-mode half-inverter gate

Note, that this current-mode logical operation and the corresponding gate was designed for more simple realisation of the Boolean operation XOR in the current-mode algebra. The current-mode expressions for the implementation operation XOR and NXOR are presented below:

$$\overline{a \oplus b} = \overleftrightarrow{(\hat{a} + b)} = \overleftrightarrow{(a + \hat{b})}, \qquad (4.14)$$

$$a \oplus b = \overleftrightarrow{\left(\overleftrightarrow{\bar{a} + \hat{b}}\right)} = \overleftrightarrow{\overleftrightarrow{\left(a + \hat{b}\right)}}, \tag{4.15}$$

(where „⊕" is the symbol of the XOR operation).

Example of the current-mode circuit for realisation of the NXOR operation is shown in the fig. 4.9.

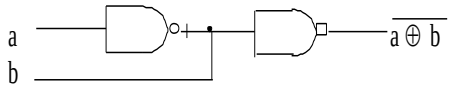


Fig. 4.9. Realization of the NXOR operation

It is follows from the expression (4.14) and fig. 4.9 that the current-mode realization of the NXOR operation has a lower hardware overhead in comparison with the realization of this operation in the Boolean algebra. Therefore, it is possible to obtain the such current-mode adder circuit which realized the function s_i in according to the following expression

$$s_i = a_i \oplus b_i \oplus c_i, \tag{4.16}$$

and therefore, is the more simple circuit.

In the current-mode algebra expression (4.16) may be represented as following:

$$s_i = \overleftrightarrow{\left(\overleftrightarrow{\left(\overleftrightarrow{a + \hat{b}}\right) + \hat{c}}\right)}. \tag{4.17}$$

Corresponding to the expressions (4.17) and (4.10) 6-gates adder circuit is shown in the fig. 4.10:

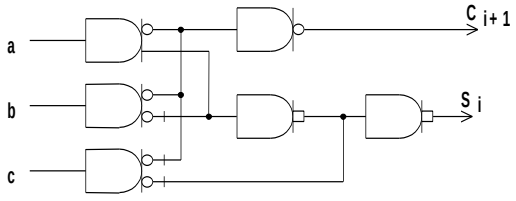


Fig. 4.10. The 6-th gates version of the one-bit adder

Note that the obtained version of the one-bit adder also is characterised on 33% smaller hardware overheads in comparison with its prototypes based on the classical voltage type of gates [3].

The extra advantage of the last derived adder circuit is the possibility the realisation (on the its base) of the fast *n*-bit current-mode adders with the parallel propagation of the carry bit. Organisation of the parallel carry bit propagation consists of the computations of the variables c_{i+1} in according to the following expression:

$$C_{i+1} = D_i \vee C_i F_i \ , \tag{4.18}$$

where

$$F_i = a_i \bar{b}_i \vee \bar{a}_i b_i \quad \text{and} \quad D_i = a_i b_i \tag{4.19}$$

For example, for the 4-bit adder the carry out variables c_5 - c_2 are realized as follows:

$$C_5 = D_4 \vee F_4 D_3 \vee F_4 F_3 D_2 \vee F_4 F_3 F_2 D_1 \vee F_4 F_3 F_2 F_1 C_1,$$
$$C_4 = D_3 \vee F_3 D_2 \vee F_3 F_2 D_1 \vee F_3 F_2 F_1 C_1 \ ,$$
$$C_3 = D_2 \vee F_2 D_1 \vee F_2 F_1 C_1 \ ,$$
$$\text{and} \qquad C_2 = D_1 \vee F_1 C_1 \, .$$

In the current-mode algebra these expressions are corresponded to the following ones:

$$C_5 = \overline{\overline{D_4 + \overline{F_4} + \overline{D_3} + \overline{F_4} + \overline{F_3} + \overline{D_2} + \overline{F_4} + \overline{F_3} + \overline{F_2} + \overline{D_1} + \overline{F_4} + \overline{F_3} + \overline{F_2} + \overline{F_1} + \overline{C_1}}}$$
$$C_4 = \overline{\overline{D_3 + \overline{F_3} + \overline{D_2} + \overline{F_3} + \overline{F_2} + \overline{D_1} + \overline{F_3} + \overline{F_2} + \overline{F_1} + \overline{C_1}}}$$
$$C_3 = \overline{\overline{D_2 + \overline{F_2} + \overline{D_1} + \overline{F_2} + \overline{F_1} + \overline{C_1}}}, \tag{4.20}$$
$$\text{and} \qquad C_2 = \overline{\overline{D_1 + \overline{F_1} + \overline{C_1}}} \, .$$

Using the new double-inverter gates the following simple current-mode circuit may be designed to the realization of the parallel propagation of the carry bit C_{i+1} :

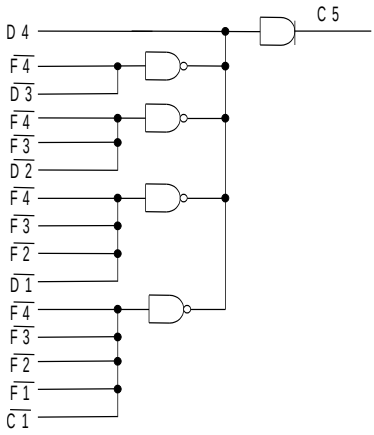


Fig.4.11. Realization of the parallel propagation of the carry bit C_{i+1}

Based on these expressions the fast 4-bit current-mode adder with the parallel carry bit propagation was derived. Its scheme is presented in the fig. 4.12.

Note, that derived circuits consists of 35 current-mode inverter gates and characterized on the 35% smaller hardware overheads in comparison with its prototypes based on the classical voltage gates.

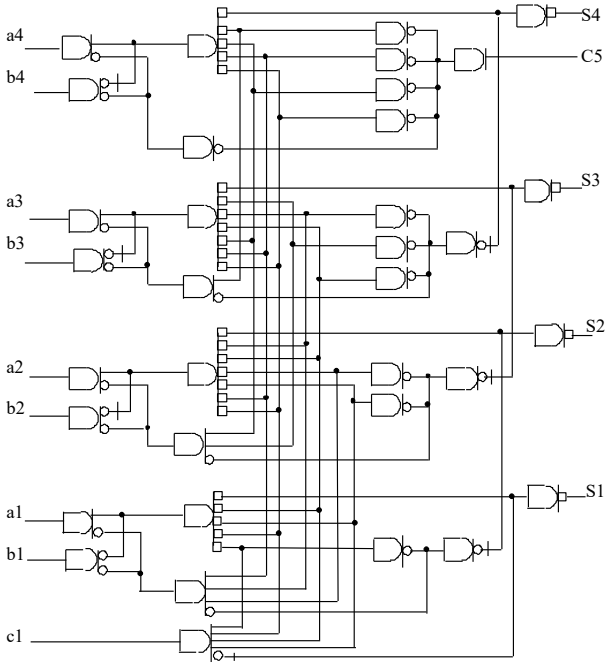


Fig.4.12. Fast 4-bit adder with the parallel propagation of the carry out bit

4.4. VHDL-models of digital circuits with the current - mode gates

Difficulties in a logical level designing of current - mode circuits are connected with multivalue logic (current – mode logic), which describes such devices. After all, some steps in the designing process are not formal, yet. Therefore, all designed current – mode circuits need to verify their work. Modeling is the best way for such verification. In general, there are two kinds of modeling digital electronic devices: low – level one ("layout" or "transistor" level, which use, for example, SPICE environment) and high – level one ("logical" level - VHDL environment). Low – level simulations make it possible to determine the most of basic characteristics of modeling schemes (included all time's parameters). Insufficiently velocity is the main disadvantage of the low level modelling. High level simulations based on extracted parameters fixing from low level ones. They are quick enough. Particularly, it's very important in the case of modelling digital circuits, which are composed of a great number of basic gates. Therefore in this paper, the problem of modelling digital circuits based on the current – mode gates is considered in this section.

4.4.1. Modification of standard's library ieee1164 for current- mode logic needs

Growing up the number of current – mode gates in circuits causes necessity of adequate formal design method and hardware description language (HDL) specialized for current – mode logic needs.

Such HDL should fulfil some requirements:

- Quick and precise simulations of described circuits;
- Similar or the same way of designing, modeling and simulations in compare with traditional Boolean logic and circuits;
- Language ought to be relatively easy, well - known and meets contemporary requirements for HDL.

VHDL is one of the hardware description language that fulfil all this requirements.

Well – known standard IEEE1164, which is the base for all HDL, includes all necessary components for traditional voltage – mode circuits and Boolean logic.

Programming environment Active – VHDL (product of Aldec Company) consists of a lot of different libraries, in which library IEEE1164 plays an important role. All changes involved into library IEEE1164 for modelling current – mode circuits refer to the one of its elements - std_logic_1164.

Standard element std_logic_1164 includes declarations of all types - logical levels (voltage – mode logic), declarations of all subtypes of logical levels, definitions and declarations of elementary functions (and, nand, or, nor, xor, nxor, not) which are described for both alone signals and vectors of signals, conversion functions between different types (subtypes) of logical levels, falling and rising edge detection functions and table and function of resolution.

Standard IEEE1164 defines 9 different logic levels for voltage – mode logic. Current mode logic needs another logic levels. It is enough for modelling purpose to use 11 current – mode logic levels (types): ‘U’ –uninitialized, ‘E’ – error, ‘C’, ’B’, ‘A’, ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’ - logical levels respectively from minus three through zero to plus five. All logical levels (except ‘U’ and ‘E’) correspond to values and directions of currents in the node. With increasing value of current rapidly growing up the consumption of energy. Therefore, the number of logical levels will not be considerably expanded.

In the case of current – mode circuits elementary voltage – mode logic functions (and, nand, or, nor, xor, nxor, not) are not useful for both alone signals and vectors of signals. Such logic functions may be realized by special connections of elementary current – mode gates (inverter, anti – inverter, double – inverter, half - inverter) which are described in the next part of this article.

Resolution table and resolution function determine the resulting value when several sources are concurrently feeding the same signal line. The resolution table lists all possible signal values in columns and rows and each cell contains information on what value will be generated if the two values are mixed. The current – mode resolution table consists of 11x11 cells (see table 4.2). Additive property of current – mode logic was took into account in building this table.

TABLE 4.2. The current – mode resolution table.

	--|**U**|	--|**E**|	--|**C**|	--|**B**|	--|**A**|	--|**0**|	--|**1**|	--|**2**|	--|**3**|	--|**4**|	--|**5**|
-- |**U**|	'U'	'E'	'C'	'B'	'A'	'0'	'1'	'2'	'3'	'4'	'5'
-- |**E**|	'E'	'E'	'E'	'E'	'E'	'E'	'E'	'E'	'E'	'E'	'E'
-- |**C**|	'C'	'E'	'E'	'E'	'E'	'C'	'B'	'A'	'0'	'1'	'2'
-- |**B**|	'B'	'E'	'E'	'E'	'C'	'B'	'A'	'0'	'1'	'2'	'3'
-- |**A**|	'A'	'E'	'E'	'C'	'B'	'A'	'0'	'1'	'2'	'3'	'4'
-- |**0**|	'0'	'E'	'C'	'B'	'A'	'0'	'1'	'2'	'3'	'4'	'5'
-- |**1**|	'1'	'E'	'B'	'A'	'0'	'1'	'2'	'3'	'4'	'5'	'E'
-- |**2**|	'2'	'E'	'A'	'0'	'1'	'2'	'3'	'4'	'5'	'E'	'E'
-- |**3**|	'3'	'E'	'0'	'1'	'2'	'3'	'4'	'5'	'E'	'E'	'E'
-- |**4**|	'4'	'E'	'1'	'2'	'3'	'4'	'5'	'E'	'E'	'E'	'E'
-- |**5**|	'5'	'E'	'2'	'3'	'4'	'5'	'E'	'E'	'E'	'E'	'E'

Descriptions of current - mode gates in VHDL are relatively easy. Here is the example of U1 gate (inverter). Descriptions of another gates are very similar.

```
entity U1 is
      generic(t_prop:time:=0.856ns);
      port (
        S_in     :    in   nstd_logic;
        S_out    :    out  nstd_logic);
end entity U1;
architecture A_U1 of U1 is
  begin
    process(S_in)
      variable s       : nstd_logic;
       begin
         if S_in'event
            then case S_in is
              when '5'|'4'|'3'|'2'|'1' => s:='0';
              when others  => s:='1';
             end case;
        end if;
        S_out<=s after t_prop;
  end process;
end architecture A_U1;
```

Each gate may have a few outputs, which realized, in general case, different current – mode logical functions. Time's parameters (for example, delay t_prop:time:=0.856ns) of such gates were fixing from SPICE simulations.

4.4.2. Simulations of VHDL-models of digital circuits with the current - mode gates

The VHDL –models of designed in the previous section the one-bit adder and the four-bit ALU with the parallel propagation of the carry bit were designed and investigated. All models were specified using structural description (except elementary gates). It doesn't mean that it is impossibly to described them using behavioral description. However, structural description makes it possibly to check up structures of circuits. It's very important in the case of huge projects, which consists of a large number of different gates. One-bit adder's model is the example of one of the easiest combinatorial, current – mode circuits. Three different current – mode gates were used in this model. Simulation of the model has confirmed the correctly work of the designed circuits (compare Fig. 4.13 with table 4.1). From Fig. 4.13 it isn't difficult to fix time delay of signal propagation through the circuit.

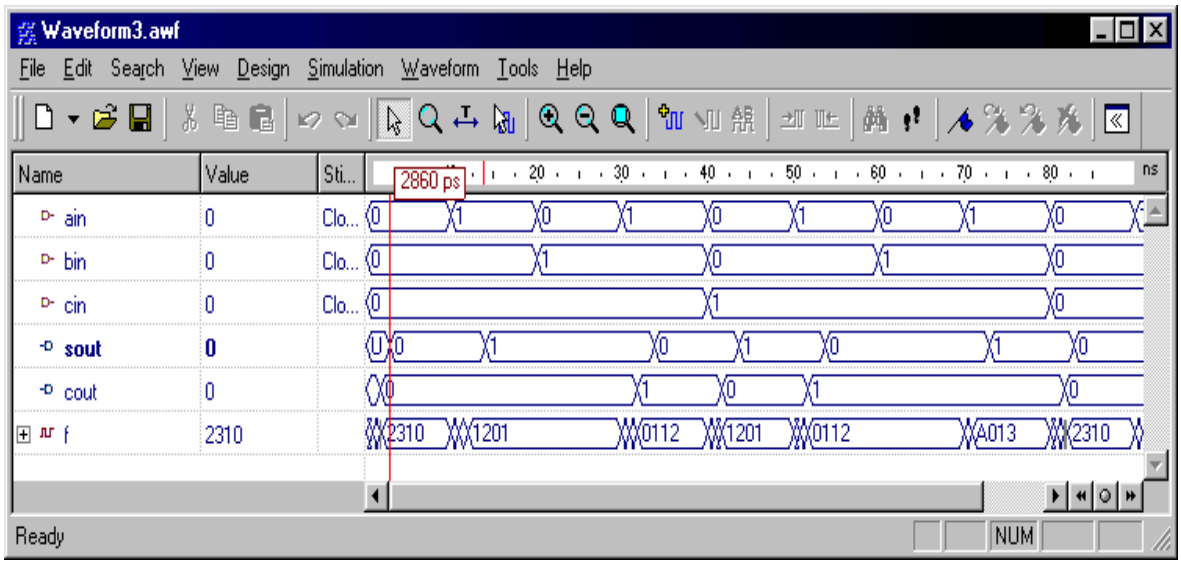


Fig.4.13. Results of simulation of designed one-bit current – mode adder.

Model of ALU 74S181 is more complicated. The set of operations which are implemented by Am74S181 unit is represented in the Table 4.3, where A and B - are operands, CI - is the input carry bit, M - is the control input of type function choice (arithmetic or logic) and E0,..,E3 - are the control inputs of function select.

TABLE 4.3. The set of operations of ALU 74S181.

N	**E3 E2 E1 E0**	**Functions**	
		Arithmetic functions (M=0)	Logic functions (M=1)
0	0 0 0 0	A+CI	NotA
1	0 0 0 1	(A v B)+CI	not(A v B)
2	0 0 1 0	(A v notB)+CI	notA & B
3	0 0 1 1	1+CI	0
4	0 1 0 0	A+(A & notB)+CI	not(A & B)
5	0 1 0 1	(A v B)+(A & notB)+CI	NotB
6	0 1 1 0	A+notB+CI	A(+)B
7	0 1 1 1	1+(A & notB)+CI	A & notB
8	1 0 0 0	A+(A & B)+CI	notA v B
9	1 0 0 1	A+B+CI	not(A(+)B)
10	1 0 1 0	(A & B)+(A v notB)+CI	B
11	1 0 1 1	1+(A & B)+CI	A & B
12	1 1 0 0	A+A+CI	1
13	1 1 0 1	A+(A v B)+CI	A v notB
14	1 1 1 0	A+(A v notB)+CI	A v B
15	1 1 1 1	1+A+CI	A

This ALU represents the combinatorial circuits and consists from two cascades. The first cascade implements the preparation of the operands in accordance in the signals state on the M and E0,...,E3 inputs, while second cascade implements only addition function.

Fig.4.14 illustrates the functional circuit diagram of the *i*-th bit of the first cascade of ALU (without the unit of the locking carry bit c_i), were *ai* and *bi* - are outputs of this cascade and $\bar{e}_j$ - are the control inputs.

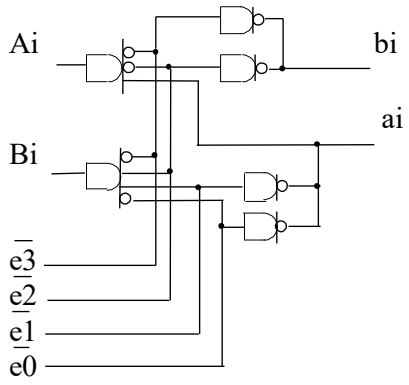


Fig. 4.14. The structure of the first cascade of the ALU 74S181.

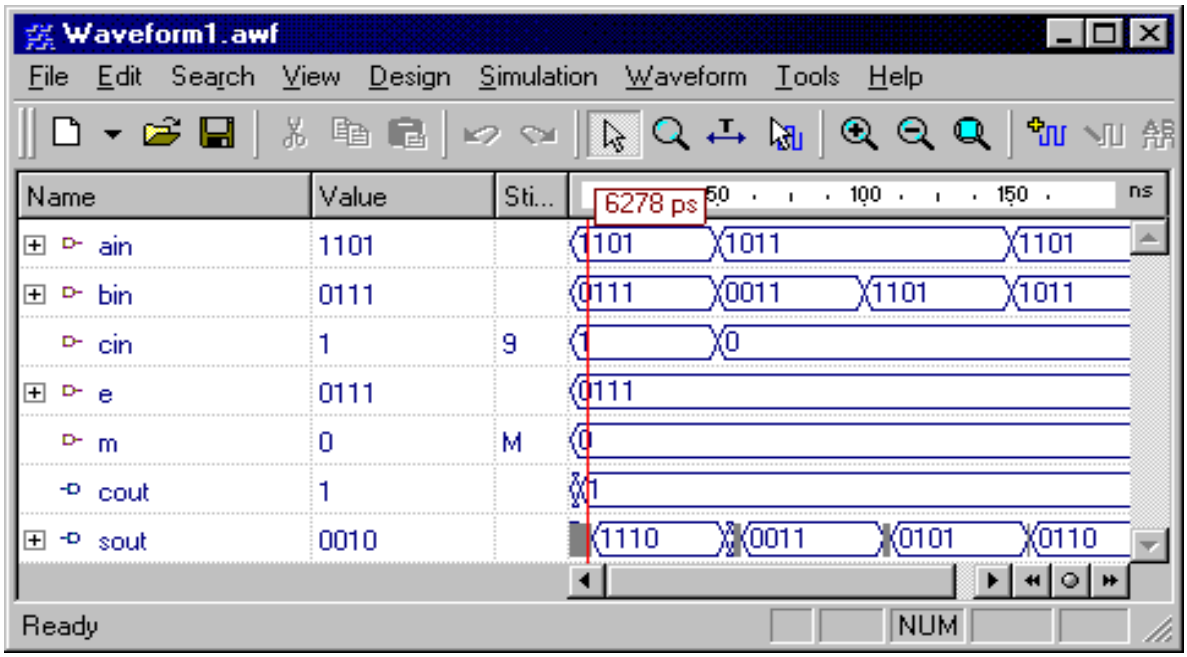


Fig.4.15. Results of simulations of designed current – mode ALU 74S181(m=0; e=1110).

The second cascade which is illustrated in the Fig.4.12 consists from the fast 4-bit adder which compute the final results of operations. Here c_1 represents the carry bit from

previous adder node, output *Si* represents the one bit of result, and C_5 represents the carry out bit of the adder.

Thus, the whole ALU circuit is consisted of 63 current – mode gates (27 U1 gates, 3 – U2, 1 – U3, 4 – U5, 4 – U12, 5 – U13, 4 – U113, 1 – U133, 4– U1111, 4 – U1133, 1 – U1333, 1 – U11112, 4 – U55555555). Simulation of one of the function is presented at Fig. 4.15

4.5. Basic block of the FPGA XC4000 series cell and designing its current-mode prototype

XC4000 series FPGA (ang. Field Programmable Gate Array) devises (chips) are implemented with a regular, flexible, programmable architecture of Configurable Logic Blocks, interconnected by a powerful hierarchy of versatile routing resources (see Fig.4.16),

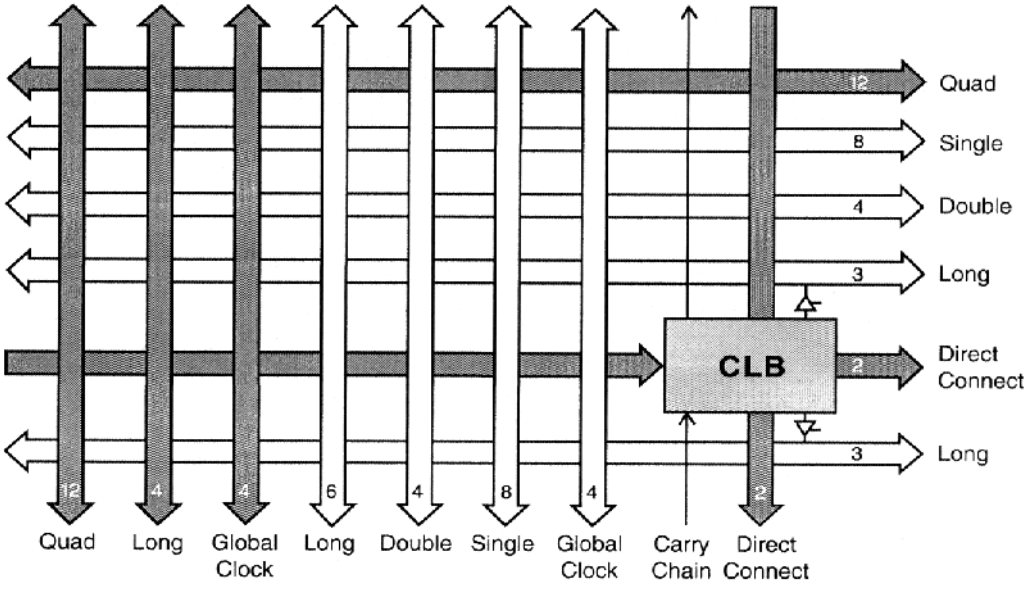


Fig 4.16. Simplified block diagram of FPGA XC4000 series cell

and surrounded by a perimeter of programmable Input/Output Blocks. They have generous routing resources to accommodate the most complex interconnect patterns

[10,11]. The devices are customized by loading configuration data into internal memory cells. The FPGA can either actively read its configuration data from an external serial or byte-parallel PROM (master modes), or the configuration data can be written into the FPGA from an external device (slave and peripheral modes).

XC4000 series FPGAs are supported by powerful and sophisticated software, covering every aspect of design from schematic or behavioral entry, floorplanning, simulation, automatic block placement and routing of interconnects, to the creation, downloading, and readback of the configuration bit stream. Because Xilinx FPGAs can be reprogrammed an unlimited number of times, they can be used in innovative designs where hardware is changed dynamically, or where hardware must be adapted to different user applications.

XC4000XLA and XC4000XV devices can run at synchronous system clock rate of up to 100 MHz, and internal performance can exceed 150 MHz. This is a high performance 3,3V family based on 0,25µ (5-layer metal) CMOS process and consisted of up to 500000 system gates and 270000 synchronous SRAM bits [11].

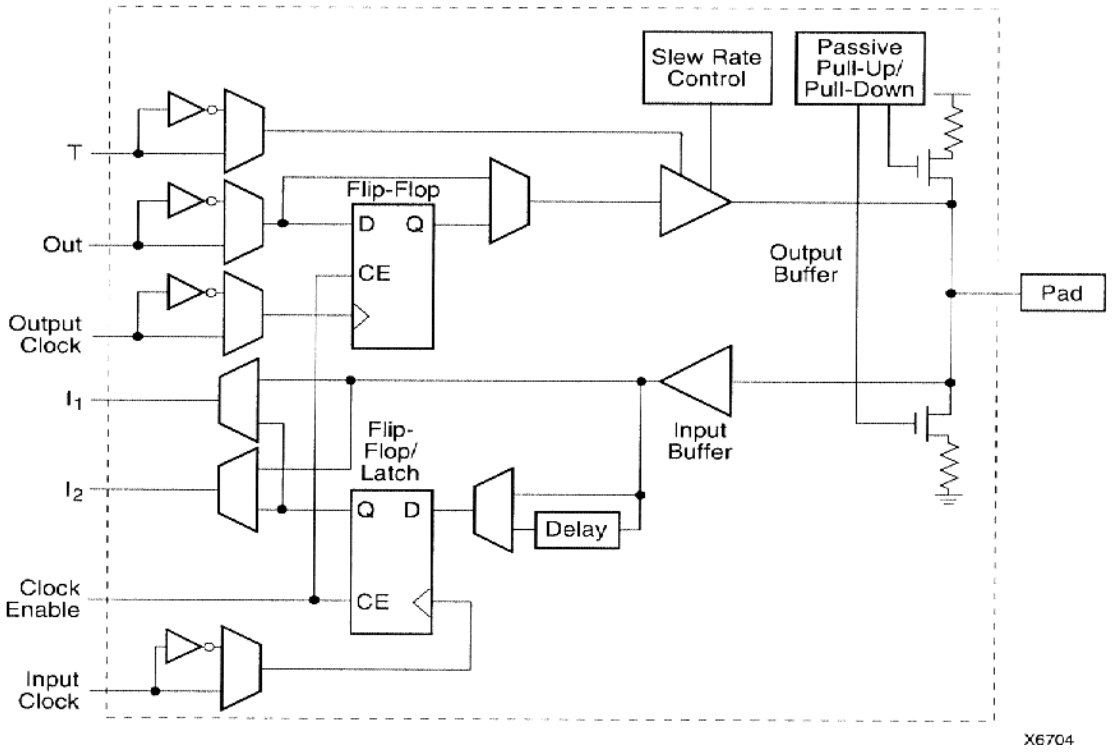


Fig 4.17. Simplified block diagram of XC4000 series IOB

Xilinx user-programmable gate arrays include two major configurable elements: configurable logic block (CLBs) and input/output blocks (IOBs). CLBs provide the functional elements for constructing the user's logic, while IOBs provide the interface between the package pins and internal signal lines (see Fig.4.17).

CLBs implement most of the logic in an FPGA. The principal CLB elements are shown in Fig, 4.18. Two 4-input function generators (F and G) offer unrestricted versatility. Most combinatorial logic functions need four or fewer inputs [11]. However, a third function generator (H) is provided. The H function generator has three inputs. Either zero, one or two of these inputs can be the outputs of F and G; the other input(s) are from outside the CLB. The CLB can, therefore, implement certain functions of up to nine variables, like parity check or expandable –identity comparison of two sets of four inputs. Each CLB contains two storage elements that can be used to store the function generator outputs.

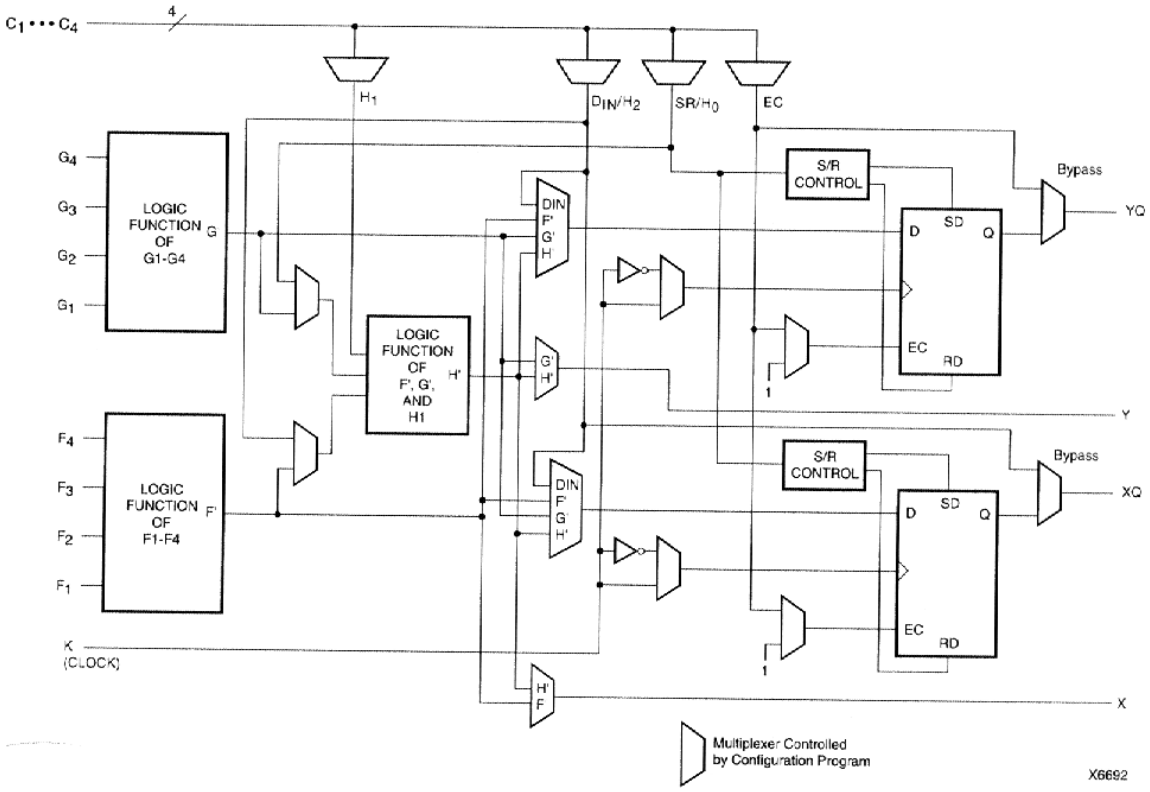


Fig 4.18. Simplified block diagram of XC4000 series CLB

However, the storage elements and function generators can also be used independently. DIN can be used as direct input to either of the two storage elements. H1 can drive the other through the H function generator. Function generator outputs can be also drive two outputs independent of the storage element outputs. Thirteen CLB input and four CLB outputs provide acces to the function generators and storage elements. These inputs and outputs connect to the programmable interconnect resources outside the block.

Four independent inputs are provided to each of two function generators (F1-F4 and G1-G4). These function generators, with outputs labeled F' and G', are each capable of implementing any arbitrary defined Boolean function of four inputs. The function generators are implemented as memory look-up-table (LUT). The propagation delay is therefore independent of the function implemented. A third generator, labeled H', can implement any Boolean function of its three inputs. This function generators is also implemented as a look-up-table. Two of its inputs can optionally be the F' and G' functional generator outputs. Alternatively, one or both of these inputs can come from outside the CLB (H2,H0). The third input must come from outside the block (H1).

Thus, the basic blocks of the FPGA cell are two 4-inputs and one 3-inputs look-up-tables. Analysis of the LUT operating shows that it can be composed of 2^n- cell one-bit FIFO block (where n is the number of LUT inputs, $n=4$ for F' and G' LUTs and $n=3$ for H' LUT) and 2^n- inputs multiplexer. The example of such 4-inputs LUT based on the classical voltage type gates is represented in the Fig.4.19.

It composes of 16 D-triggers 1-16, which are a base of the 16-cell one bit FIFO block, the multiplexer MUX and the AND gate. The multiplexer output is the output G' of the LUT, while the LUT inputs G1-G4 are the control inputs of the multiplexer. During FPGA configuration (i.e. during the process of loading design-specific programming data into FPGA), the signal on the PROGRAM input is equal to the logical one. It allows on serial

loading (with clock signals CLK) the values of target logical function Y(G4,G3,G2,G1) from external pin DIN to corresponding D-triggers.

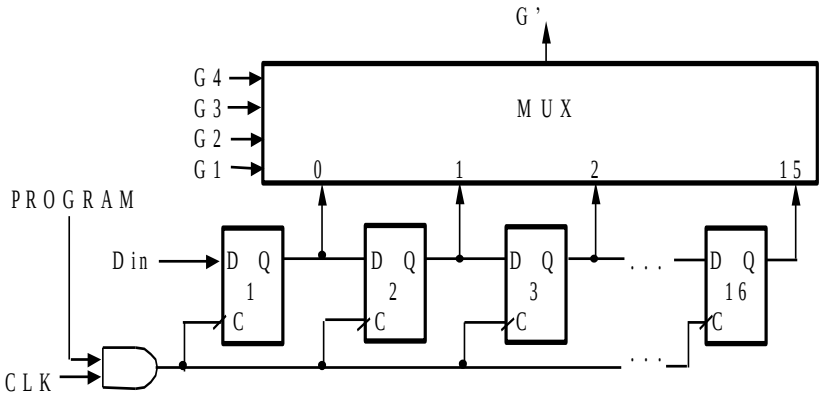


Fig. 4.19. Example of the internal structure of 4-inputs LUT (function generator G)

For example, the value of the function Y(0,0,0,0) must be written in the D-trigger 1, the value of function Y(0,0,0,1) must be written in the D-trigger 2, etc. During normal operating, the signal on the PROGRAM input is equal to the logical zero, due to all D-triggers save the values of the target function. At this time, the code on the LUT inputs G4-G1 determined the number of D-trigger which is connected to the LUT output G'.

Therefore, in order to deriving of the current-mode LUT structure, the current-mode D-trigger and multiplexer circuits should be designed. D-trigger is the sequential circuit, therefore it consists of both flip-flop and combinatorial circuit. Flip-flops are the elementary binary memory units used in the all types of sequential digital circuits. The set of operations, which are implemented by a flip-flop and structure of the current-mode flip-flop circuit is shown in the Fig. 4.20, where **f1** and **f2** are flip-flop inputs, and $\mathbf{Q^t}$ and $\mathbf{Q^{t+1}}$ are the values on the flip-flop output **Q** at the time step **t** and **(t+1)** respectively.
In according to the Fig.4.20, the flip-flop do not change its state ($\mathbf{Q^{t+1}=Q^t}$) if for the time step **t**, the signals on the input **f1** and **f2** are equal to the arbitrary value from the sets {1,2,3,...} and {1,0,-1,...} or vice versa. When inputs of the flip-flop are equal to , for example, **f1**=0, and **f2**=1, the flip-flop set in the state of logical one ($\mathbf{Q^{t+1}}$=1).

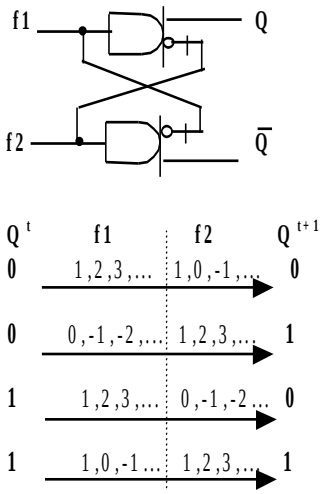


Fig. 4.20. Current-mode flip-flop circuit and its operations

The table of the D-trigger operations are represented by table 4.4.

Tabl. 4.4. Table of the current-mode D-trigger operations

C	**D**	**Q^t**	**Q^{t+1}**	**f1**	**f2**
0	0	0	0	1,2,3	1,0,-1
0	0	1	0	1,2,3	0,-1,-2
0	1	0	1	0,-1,-2	1,2,3
0	1	1	1	1,0,-1	1,2,3
1	0	0	0	1,2,3	1,0,-1
1	0	1	1	1,0,-1	1,2,3
1	1	0	0	1,2,3	1,0,-1
1	1	1	1	1,0,-1	1,2,3

Using the proposed above approaches to the designing of binary current-mode circuits, the following expressions for the functions f1 and f2 is derived:

$$f1 = \overline{D} + C \quad , \quad f2 = D + C. \tag{4.21}$$

These expressions determined the circuit of the current-mode synchronic low level sensitive D-trigger which is shown in the Fig. 4.21.

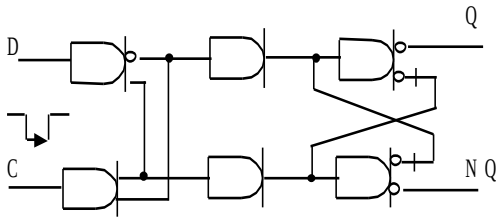


Fig. 4.21. Current-mode low level sensitive D-trigger

However, this version of the D-trigger is not suitable in FIFO blocks, in which each flip-flop should be triggered on either the rising or falling clock edge. In this case, the Master-Slave (MS) version of the D-trigger must be used. Such current-mode MS D-trigger is shown in the Fig. 4.22.

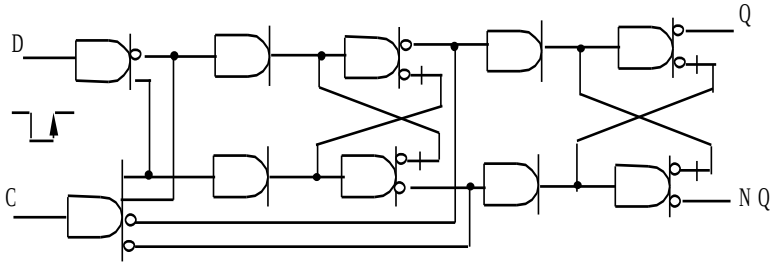


Fig. 4.22. Current-mode rising clock edge driven D-trigger

As a result, the current-mode prototype of 16-cells one-bit FIFO circuit is designed. It is represented in the Fig.4.23. Note, that the 3-inputs LUT H' consists of eight these D-triggers, which compose the 8-cell one-bit FIFO block.

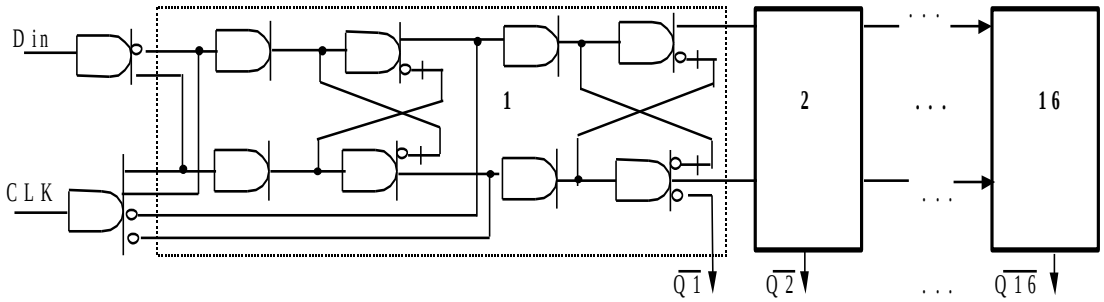


Fig. 4.23. Current-mode 16-cells one-bit FIFO circuit – base of the LUT F' and G'

Multiplexer MUX of the function generator H' has eight inputs, and implements the following function in the current-mode algebra:

$$H' = \overline{\overline{\overline{Q1} + \overline{H3} + \overline{H2} + \overline{H1}} + \overline{\overline{Q2} + \overline{H3} + \overline{H2} + H1} + \ldots + \overline{\overline{Q7} + H3 + H2 + H1}}$$

As a result, the current-mode prototype of the H' LUT multiplexer circuit is represented in the Fig.4.24.

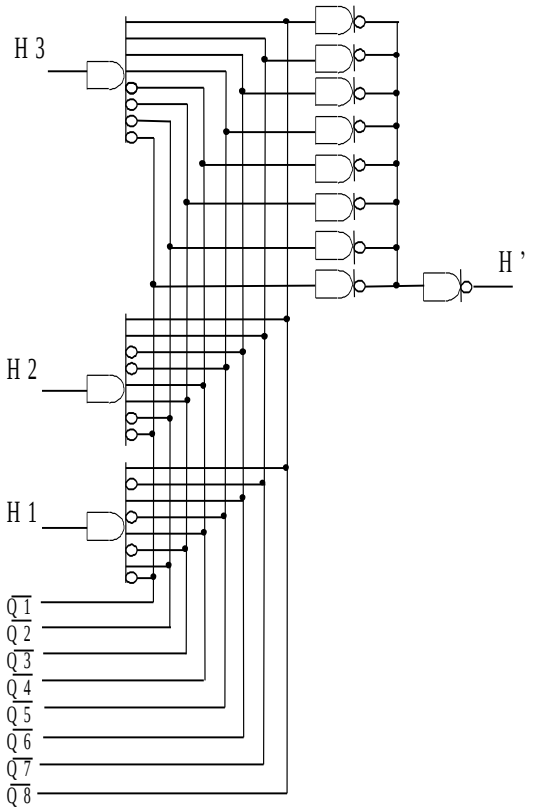


Fig. 4.24. Current-mode prototype of the LUT H' multiplexer circuit

4.6. Conclusions to the chapter 4

1. The researching of problems of the digital circuits designing with the current-mode gates showed the possibility the realization of the effective mixed analogue-digital systems with low level of common noise on a common chip.

2. The logical properties of the current-mode gates and logic and the several identities for the conversion of expressions from the Boolean algebra were a base of the proposed approach to the design of the digital current-mode circuits and allow to reduce the hardware overheads for circuits realization. As a result, the functional schemes of the several current-mode digital circuits were derived. The obtained circuits are characterized by smaller hardware overhead in comparison with similar ones based on others gate types.

3. Complicated current mode circuits need logical level's simulations. In a order for this, a standard element std_logic1164 of Active – VHDL library IEEE1164 has been changed for the current – mode purpose and the library of current – mode gates was created. Due to these libraries all designed current – mode circuits: adders, decoders, multiplexers, triggers, 32 functions ALU, FPGA function generator, etc., were simulated. Simulations of VHDL models have verified both the approach to designing and changes introduced into the standard VHDL library IEEE1164.

CONCLUSIONS

1. The several theorems were proved for the Gauss elimination, LU-decomposition, Choleski, and Jordan-Gauss algorithms, which allowed to improve of the origin WCS method (known fault-tolerance technique). Moreover, the sufficient conditions for using of the proposed method for others LA algorithms were formed.
2. The fault tolerant versions of the Gauss elimination, LU-decomposition, Choleski and Jordan Gauss algorithms were designed using modified WCS method. The computational complexity of the derived fault-tolerant algorithms is increased approximately on $O(N^2)$ multiply -add operations in comparison with the original algorithms. However, new fault tolerant algorithms enable to detect and to correct a single error in an arbitrary row or column of the input matrix at the each algorithm step. Hence, it is possible to correct up to $N^2/2$, $N^2/4$ and N^2 single errors during realization of the whole Gauss, Choleski and Jorgan-Gauss algorithms respectively.
3. For estimating of the tolerance of proposed algorithms to transient faults and evaluation of numerical error for different encoder vectors, the programmed environment „ABFT" (*Algorithm-Based Fault Tolerance*) was designed. The testing of the proposed algorithms proved that they are correct and enable to detect and to correct a single error in an arbitrary row or column of the input matrices at the each algorithm step.
4. The new method for the construction of the lattice DGs of algorithms given by nested loops has been proposed. In a contrast with known analytical methods, the proposed method is more simple and feasible for the implementation in CAD systems for the structural design of application-specific parallel processors, and allows operating with a wider class of algorithms such as, for example, non-uniform recursive algorithms corresponding to non-perfect (or composite) loop nests.
5. The fixed-size processor array architectures for the implementation of the fault-tolerant Jordan-Gauss, Cholesky, Gauss elimination and back substitution algorithms were derived. Based on this arrays architectures, the FPGA-based structure of the application-specific parallel system destined to the fault-tolerant implementation of these algorithms was obtained.

6. The logical properties of the current-mode gates and logic, and the identities for the conversion of expressions from the Boolean algebra were a base of the proposed approach to the minimisation of current-mode logical functions and designing of the binary current-mode circuits and allow to reduce the hardware overheads for circuits realisation. Using proposed approach, the functional schemes of the several current-mode digital circuits were derived. The obtained circuits are characterised by smaller hardware overhead in comparison with similar ones based on others gate types.
8. In the order to the simulation of the designed current-mode circuits, a standard element std_logic1164 of Active – VHDL library IEEE1164 has been changed and the library of current – mode gates was created. Based on these libraries, all designed current – mode circuits: adders, decoders, multiplexers, triggers, 32 functions ALU, FPGA function generator, etc., were simulated. Simulations of VHDL models have verified both the approach to designing and changes introduced into the standard VHDL library IEEE1164.

REFERENCES

[1]. C. Mead, L. Conway. Introduction to VLSI Systems. *Reading, Mass.:Addison-Wesley,* 1980.

[2]. H.T. Kung. Notes on VLSI Computation. In Parallel Processing Systems, D.J. Evans, ed. *New York: Cambrige University Press,* 1983, pp.339-356.

[3]. S.Y. Kung. VLSI Array Processors. *Englewood Cliffs,N.J.: Prentice Hall*, 1988;

[4]. Kung S.Y., Whitehouse H.J., Kailath T. VLSI and Modern Signal Processing. *Prentice-Hall, Englewood Cliffs, New Jersey,* 1988.

[5]. J.D. Ulman. Computational Aspects of VLSI. *Computer Science Press, Inc.,* 1984.

[6]. J. Isoaho, J. Pasawn, O. Vaino, H. Terhunen. DSP System Integration and Prototyping With FPGAs. *J. VLSI Signal Processing,* 1993, N 6, p. 155-172.

[7]. Parallel and Distributed Computing Handbook., Albert Y. Zomaya editor, *McGraw-Hill,* 1996, P.1199.

[8]. D. E. Culler, J.P. Singh with A. Gupta. Parallel Computer Architecture. *Morgan Kaufmann Publishers, Inc., USA,* 1999, 1025 p.

[9]. D. Burger, J.R. Goodman. Billion-Transistor Architectures. *Computer, Vol.30, N.9,* 1997, pp.46-48;

[10]. G.R. Goslin. A Guide to Using Field Programmable Gate Arrays (FPGAs) for Application-Specific Digital Signal Processing Performance. *Xilinx, Inc.,* 1995.

[11]. The Programmable Logic Data Book. *Xilinx, Inc.,* 1999.

[12]. M.H. Lipasti, J.P. Shen. Superspeculative Microarchitecture for Beyong AD2000. *Computer, Vol.30, N.9,* 1997, pp.59-66;

[13]. C.E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson and others. Scolable Processors in the Billion-transistor Era: IRAM. *Computer, Vol.30, N.9,* 1997, pp.75-78;

[14]. B. Cohen. VHDL Answers to Frequently Asked Questions. *Kluwer Academic Publishers,* 1997.

[15]. L. Hammond B. Nayfeh, K. Olukotun. A Single-Chip Multiprocessor. *Computer, Vol.30, N.9,* 1997, pp.79-85;

[16]. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar and others. Baring It All to Software: Raw Machines. *Computer, Vol.30, N.9,* 1997, pp.86-93.

[17]. A. Rushton . VHDL for Logic Syntesis. Second Edition. *Jon Wiley & Sons,* 1998.

[18]. J.P. Shen. Superspeculative Microarchitecture for Beyond AD2000. *Computer, Vol.30, N.9,* 1997, pp.59-66

[19]. J.H Wilkinson, The algebraic eigenvalue problem. *Oxford University Press, New York,* 1963.

[20]. Faddeev D.K., Faddeeva V.N Computational methods of linear algebra. *W.H. Freeman and Company,* 1963.

[21]. G. H. Golub, C.F.V.Loan. Matrix Computations. *Baltimore: John Hopkins Univ.Press,* 1983.

[22]. Moreno J.H., Lang T. Matrix computations on systolic-type arrays. *Kluwer Acad.Publ., Boston,* 1992.

[23]. Jennings A., McKeown J.J. Matrix computations. *Willey &Sons, Chichester,*1992.

[24]. Ortega J.M. Introduction to parallel and vector solution of linear systems. *Plenum Press, New York,* 1988.

[25]. JaJa J. An introduction to Parallel Algorithms. *Reading, Mass.: Addison Wesley,* 1992.

[26]. M. Cosnard, D. Trystram. Parallel Algorithms and Architectures. *International Thomson Computer Press, Boston,*1995.

[27]. D.J. Evans, ed. Systolic algorithms. *New York: Gordon and Breach,* 1991.

[28]. D.I. Moldovan. On the analysis and synthesis of VLSI algorithms. *IEEE Trans.Comp., 31,* 1982, pp.1121-1126;

[29]. Wyrzykowski R., Kanevski J.S., Maslennikov, O.V. Systolic-type implementation of matrix computations based on the Faddeev algorithm. *Proc. IEEE Int. Conf. Massively Parallel Computing Systems, Ischia (Italy),* 1994, pp.31-42.

[30]. Kanevski J.S., Maslennikov O.V., Wyrzykowski R. The solution of linear algebraic systems on VLSI processor array. *Cybernetics, 1996, N2,* p.25 - 32. (in Ukraine).

[31] Wyrzykowski R., Kanevski J.S., Maslennikov O.V. Mapping recursive algorithms into processor arrays. *Proc. Int. Workshop Parallel Numerics'94, Smolenice (Slovakia),* 1994, pp.169-191.

[32] Barada H., and El-Amawy A. A methodology for algorithm regularization and mapping into time-optimal VLSI arrays. *Parallel Computing,* 1993, 19, pp.33-61.

[33] Quinton P., Robert Y. Systolic algorithms and architectures. *Prentice Hall, Englewood Cliffs,* 1991.

[34]. A. Avizienis. Design of Fault-Tolerant Computers. *Proc. Fall Joint Computer Conf., AFIPS Conf. Proc., Vol.31, Thompson Books, Washington, D.C.,* 1967, pp.733-743.

[35]. Dependable computing and fault tolerant systems, Vol.5, . J.C. Laprie ed., *Springer-Verlag,* 1991.

[36]. J.C. Laprie, J. Arlat, C. Beounes, K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer, Vol.23, N7,* 1990, pp.39-51.

[37]. A.K. Somani, N.H. Vaidya. Understanding Fault Tolerance and Reliability, *Computer Vol.30, N4,* 1997, pp. 45- 50.

[38]. T.C.-K. Chou. Beyond Fault Tolerance, *Computer Vol.30, N4,* 1997, pp.47-49.

[39]. A. Avizienis. Toward Systematic Design of Fault-Tolerant Systems, *Computer Vol.30, N4,* 1997, pp.51-58.

[40]. D.P. Siewiorek, R.S. Swarz. Reliable Computer Systems Design and Evaluation, *Digital Press,* 1992.

[41]. J.F. Wakerly. Error Detecting Codes, Self-Cheking Circuits and Applications, *North-Holland, New York,* 1978.

[42]. W.W. Peterson, E.J. Weldon. Error-Correcting Codes, *The MIT Press,* 1994.

[43]. Koren I. A reconfigurable and fault tolerant VLSI multiprocessor array. *Proc. of the Eight Annual Symp. on Comp. Arch.,* pp.425-441, 1981.

[44]. S.Y. Kung, C.W. Chang, C.-W. Jen. Real-Time Reconfiguration for Fault-Tolerant VLSI Array Processors. *IEEE Trans.comp.,* 1986, pp.46-54.

[45]. J.-J. Wang, C.-W. Jen. Redundancy design for fault tolerant systolic array. *IEE Proc., Vol.137, Pt.E, N3,* 1990, pp.218-226.

[46]. V. Hecht, K. Ronner, P. Pirsch. A Detect-Tolerant Systolic Array Implementation for Real-Time Image Processing. *Journal of VLSI Signal Processing, 5,* 1993, pp.37-47.

[47]. Esonu M.O., Hariri S., Al-Khalili A.J. Fault - tolerant design methodology for systolic array architectures. *IEE Proc.-Comput. Digit. Tech., vol.141, N 1, January* ,1994.

[48]. D. Blough, G. Masson. Performance Analysis of a Generalized Concurrent Error Detection Procedure. *IEEE Trans.Comp., 1990, Vol.39, N1,* pp.47-62.

[49]. K. Grans. DUSPER - a New Fault Tolerance Technique Combining Duplication and Recovery. *Proc. 9th European Workshop on Dependable Computing, Gdansk,* 1998, pp.110-113.

[50]. Butner S.E. Triple time redundancy, fault-masking in byte-sliced systems. in *Tech. Rep. CSL TR 211, Comput. Syst. Lab.,Dep. of Elec. Eng., Stanford Univ.,Stanford, CA,* 1981.

[51]. Patel J.H. and L.Y.Fung. Concurrent error detection in ALU's by recomputing with shifted operands. *IEEE Trans.Comput., Vol.C-31,* pp.589-595, 1982.

[52]. D.J. Taylor, D.E. Morgan J.P. Black. Redundancy in Data Structures: Improving Software fault Tolerance. *IEEE Trans. Software Engineering, Vol.SE-6,* 1980, pp 585-594.

[53]. Manolakos E.S., Kung S.Y. Neighbor assisted recovery in VLSI processor arrays. *Europian Signal Processing Symposium, EUSIPCO'88, North Holland,* 1988.

[54]. C. Gong, R. Melchem, R. Gupta. Loop transformations for Fault Detection in Regular Loops on Massively Parallel Systems. *IEEE Trans. Parallel and Distributed Systems, 1996, Vol.7, N12,* pp.1238-1249.

[55]. M. Neyman. Non-deterministic Recovery of Computations in Testing of Distributed Systems. *Proc. 9th European Workshop on Dependable Computing, Gdansk,* 1998, pp.114-117.

[56]. J. Sosnowski, M. Pawlowski. Universal and Application Dependant Testing of FPGAs. *Proc. 2nd Int.Workshop on Dependable Computing, EDCC-2, Gliwice,* 1996, pp.111-120.

[57]. H. Krawczyk. Evaluation Criteria of Dependable Systems. *Proc. 2nd Int.Workshop on Dependable Computing, EDCC-2, Gliwice,* 1996, pp.29-40.

[58]. Huang K.H. and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput., Vol. C-33,* pp.518-528, 1984.

[59]. Jou J.Y. and J.A. Abraham. Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures. *Proc.IEEE, Vol.74, No. 5,* pp.732-741, 1986.

[60]. Luk F.T. and H. Park . An analysis of algorithm- based tolerance techniques. *SPIE Vol.696 , Advanced algorithms and architectures for signal processing,* pp.222- 227, 1986.

[61]. Nair V.S.S. and J.A.Abraham. Real-number codes for fault-tolerant matrix operations on processor arrays. *IEEE Trans.on Comp*., *Vol.39, No.4,* pp.426-435, 1990.
[62]. Schimmel D.E. and F.T. Luk. A practical real time SVD machine with multi-level fault tolerance. *SPIE Vol.698, Real time signal processing IX,* pp.142-148, 1986.
[63] Han J.-Y., Krishnan D.C. Linear arithmetic code and its application in fault-tolerant systolic arrays. *IEEE Proc., Southeastcon,* 1989,pp.1015-1020.
[64]. B. Vinnakota, N.K. Jha. Design of Multiprocessor Systems for Concurrent Error Detection and Fault Diagnosis. *Proc. Int. Symp. Fault-Tolerant Computing, Montreal,* 1991, pp.504-511.
[65]. S. Yajnik, N.K. Jha. Design of Algorithm-Based Fault Tolerant Systems with In-System Checks. *Proc. Int. Conf. Parallel Processing, Vol.1, St. Charles, Ill., Aug.*1993.
[66]. S. Yajnik, N.K. Jha. Graceful Degradation in Algorithm-Based Fault Tolerant Multiprocessor Systems. *IEEE Trans. Parallel and Distributed Systems, Vol.8, N2,* 1997, pp.137-153.
[67]. M.Vijay, R.Mittal. Algorithm-based fault tolerance: a review. *Microprocessors and Microsystems, Vol.21,* 1997, pp.151-161.
[68]. Luk F.T., Park, H. An analysis of algorithm-based fault tolerance techniques. *Proc. SPIE, 1986, V.696, Advanced algorithms and architectures for signal processing,* pp.222-227.
[69]. Bliss W.G., Lightner M. R., Friedlander B. Numerical properties of algorithm based fault-tolerance for high reliability array processors. *Maple Press, 1988,* pp.631-635.
[70]. Cohen A.M. Numerical Analysis. *New York: Wiley,* 1973.
[71]. Wyrzykowski R., Lienu J.-P., Maslennikowa N., Maslennikow O. An automatized procedure for deriving graphs of recursive algorithms. *Proc. Int. Conf. „CAD DD'95", Minsk (Bielarus),* 1995, p.21- 24.
[72]. Wyrzykowski R., Kaniewski J., Maslennikowa N., Maslennikow O. The fault-tolerant solution of main linear algebra tasks. *Proc. Int. Conf. „CAD DD'95", Minsk (Bielarus),* 1995, p.6 -10.

[73].Wyrzykowski R., Kanevski Ju.S., Maslennikov O.V., Maslennikova N.N. A Method for Deriving Dependence Graphs of Recursive Algorithms for Processor Array Design. *Proc. Int.Workshop "Parallel Numerics'95" Sorrento, Italy,* 1995, p.263-280.

[74]. Kanevski J.S., Maslennikov O.V, Maslennikova N.N., Wyrzykowski R. Algorithm-Based Fault Tolerant Matrix Triangularization on VLSI Processor Arrays. *Proc. of the Int.Workshop "Parallel Numerics'95", Sorrento, Italy, Sept.*1995, p.281-295.

[75]. Kanevski Ju.S., Maslennikov O.V., Maslennikova N.N. Algorithm-Based Fault Tolerant Solution of linear Systems on VLSI Processor Arrays. *Preprints of the 3-rd IFAC/IFIP Workshop on Algorithms and Architectures for Real-Time Control (AARTC'95),Ostend, Belgia,* 1995, p.459.

[76]. Kanevski Ju.S., Maslennikov O.V., Maslennikova N.N., Wyrzykowski R. Fault Tolerant Matrix Triangularization on VLSI. *Proc. of the Int. Conf. "ISPAT'95", Boston, USA, October* 1995.-p.2247 - 2251.

[77]. Wyrzykowski R., Kaniewski J., Maslenikowa N., Maslennikow O. Formalizowana metoda konstrukcji funkcjonalnych grafów regularnych algorytmów. *Biuletyn „Modelowanie elektroniczne", Kijów,* 1996, N 2, p.30 - 38.

[78]. Wyrzykowski R., Kanevski J., Maslennikova N., Maslennikov O. Algorithm - Based Fault Tolerant Solution of Linear Systems on VLSI. *Proc. Int. Workshop on Parallel Numerics' 96, Gozd Martujek, Slovenia,* 1996, pp.243 - 244.

[79]. Wyrzykowski R., Kanevski J., Maslennikova N., Maslennikov O., Ovramenko S. Formalized Construction Method of Array Functional Graphs for Regular Algorithms. *Engineering Simulation*, 1997, Vol.14, pp.217-232.

[80]. Wyrzykowski R., Kaniewski J., Maslenikowa N., Maslennikow O. Algorytmiczne niezawodna dekompozycja macierzy na systemach wieloprocesorowych. *Biuletyn „Modelowanie elektroniczne", Kijów,* 1997, N6, str. 41-49.

[81]. Kanevski J., Maslennikova N., Maslennikov O., Wyrzykowski R. The modified weighted checksums method for designing fault tolerant linear algebra algorithms. *Proc. 9-th Int. Workshop on Dependable Computing, Gdańsk, Poland,* 1998, pp.82 - 86.

[82]. Maslennikow O., Maslennikowa N., Guziński A., Kaniewski J., Pawłowski P. Design of adders with current-mode gates. *Proc. of the XXI Nat.Conf. on Circuit Theory and Elektronic Networks, Poznań, Poland,* 1998, pp.119-124.

[83]. Maslennikow O., Maslennikowa N., Guziński A., Kaniewski J. Approarches to Designing Digital Circuits with the current-mode gates. *Proc. of the 6-th Int.Conf. on Mixed Design, MIXDES'99, Krakow, Poland,* 1999, pp.89-94.

[84]. Gretkowski D., Berezowski R., Maslennikowa N. Opis i modelowanie cyfrowych ukladow pradowych z wykorzystaniem jezyka VHDL. *Proc. 2nd Nat. Conf. Reprogramowalne uklady cyfrowe, RUC'99, Szczecin,* 1999, pp.165-172.

[85]. Gretkowski D., Kaniewski J., Maslennikowa N., Soltan P. Current-mode digital Circuits Design and Modeling. *Proc. of the XXII Nat.Conf. on Circuit Theory and Elektronic Networks, Warszawa-Stare Jablonki, Poland,* 1999, pp.161-167.

[86]. Ju. Kanevski, O. Maslennikov, N. Maslennikova. Design of FPGA-based Processor Array Architecture for Linear Algebra Algorithms Implementation. *Proc. 3-th Int. Conf. Parallel Processing and Applied Mathematics, PPAM'99, Kazimierz Dolny, Poland,* 1999, pp.

[87]. J. Kaniewski, O. Maslennikow, N. Maslennikowa, L. Lacinski. Analytical Method for Deriving Dependence Graphs of Recursive Algorithms. *Proc. of the Int.Workshop "Parallel Numerics'99", Salzburg, Austria*, 1999, p.41-56.

[88]. A. Guzinski, A. Kielbasinski, Current-Mode Digital Circuits Operating in Mixed Analog-Digital Systems. *Bulletin of the Polish Academy of Sciences, Technical Sciences, Vol. 44, No. 2*, 1996, pp. 193-198.

[89]. R. Gonzalez, B. M. Gordon, M. A. Horowitz, "Supply and Threshold Voltage Scaling for Low Power CMOS", *IEEE J. Solid-State Circuits*, vol. 32, No. 8, August 1997, pp. 1210-1215.

[90]. Guzinski A., Pawłowski P., Kaniewski J., Czwyrow D., Maslennikow O. Current-Mode Digital Circuits for Low Voltage Mixed A/D Systems. *Bulletin of the Polish Academy of Sciences, Technical Sciences, Vol. 46, No. 4*, 1998, pp.443-458.

[91]. Pawłowski P., Guziński A., Kaniewski J., Maslennikow O, Czwyrow D. Low-voltage current-mode digital circuits. *Proc. of the XXI Nat.Conf. on Circuit Theory and Elektronic Networks, Poznań, Poland,* 1998, p.119-124.

[92]. D.K. Su, M.J. Loinaz, S. Masui, B. Woodley. Experimental results and modeling techniques for substrate noise in mixed-signal integrated circuits. *IEEE J. Of Solid-State Circuits, N4,* 1993, pp.420-430.

[93]. A. Iwata, M. Nagata. A concept of analog-digital merged circuit architecture for future VLSIs. *Analog Integrated Circuits and Signal Processing, N11,* 1996, pp.83-89.
[94]. A.Matsuzawa. Low-voltage and low-power circuit design for mixed analog/digital systems in portable equipment. *IEEE J. Of Solid-State Circuits, N4,* 1994, pp.470-480.
[95]. K. Shimohigashi, K. Seki Low-voltage VLSI design. *IEEE J. Of Solid-State Circuits, N4,* 1993, pp.408-413.
[96].R.T.L. Saez, M. Kayal, M. Declercq, M.C. Schneider Digital circuit techniques for mixed analog/digital circuits applications. *Proc. of Int. Conf. ICECS'96,* pp.956-959.
[97]. M. Ingels, M.S.J. Steyaert. Design strategies and decoupling techniques for reducing the effects of electrical interference in mixed-mode ICs. *IEEE J. Of Solid-State Circuits, N7,* 1997, pp.1136-1141.
[98]. K.Makie-Fukuda, T.Kikuchi, T.Matsuura, M.Hotta. Measurement of digital Noise in Mixed-Signal integrated circuits. *IEEE J. Of Solid-State Circuits, N2,* 1995, pp.87-92.
[99]. BLISS W.G., LIGHTNER M.R., FRIEDLANDER B. Numerical properties of algorithm based fault-tolerance for high reliability array processors. Maple Press, 1988, pp.631-635.
[100]. RAJOPADHYE S.V. Synthesizing systolic arrays with control signals from recurrence equations, Distributed Computing, 1989, 3, pp.88-105.
[101]. BANERJEE U. An introduction to a formal theory of dependence analysis, J. Supercomput., 1988, 2, pp.133-149.
[102]. DARTE, A., ROBERT, Y.: Mapping uniform loop nests onto distributed memory architectures, Parallel Computing, 1994, 20, pp.679-710.
[103]. Nash J.G., Hansen S. Modified Faddeev algorithm for concurrent execution of liner algebraic operations. *IEEE Trans.Comp., 1988, C-37, pp.129-137.*

APPENDIX 1.

Środowisko do opracowania i badania wiarygodnych wersji algorytmów numerycznych

Zadania środowiska

Środowisko służące do badania wiarygodnych wersji algorytmów numerycznych musi pozwalać na:

- testowanie algorytmów na różnych typach danych wejściowych takich jak: Single, Real, Double;
- wybór wektora kodującego (linear lub average), według którego uzupełniane będą dane wejściowe sumami kontrolnymi CS i ważonymi sumami kontrolnymi WCS;
- wprowadzanie błędnych danych w dowolnym kroku wykonywania algorytmów Gaussa, Jordana-Gaussa oraz Cholesky'ego;
- określenie dokładności sprawdzania prawidłowości obliczeń;
- wybór punktów wstrzymania pracy programu (po wystąpieniu błędu lub co krok) w trakcie wykonania algorytmów;
- odczyt danych wejściowych z pliku oraz zapis danych wyjściowych do pliku;
- wyświetlenie i drukowanie danych wejściowych, wyników i odnalezionych błędów.

Do zaprojektowania środowiska został wybrany pakiet Delphi firmy Borland. Delphi jest narzędziem programowym przeznaczonym do szybkiego opracowywania aplikacji (pracujących w środowisku Windows), łączy w sobie interfejs graficzny do projektowania programów z możliwościami programowania języka Object Pascal.

Środowisko ABFT (*ang. Algorithm-Based Fault Tolerant*) pracuje w systemie operacyjnym Windows 95 i umożliwia badanie wiarygodnych wersji algorytmów (omówionych w punkcie 5): eliminacji Gaussa, Jordana-Gaussa, Cholesky'ego.

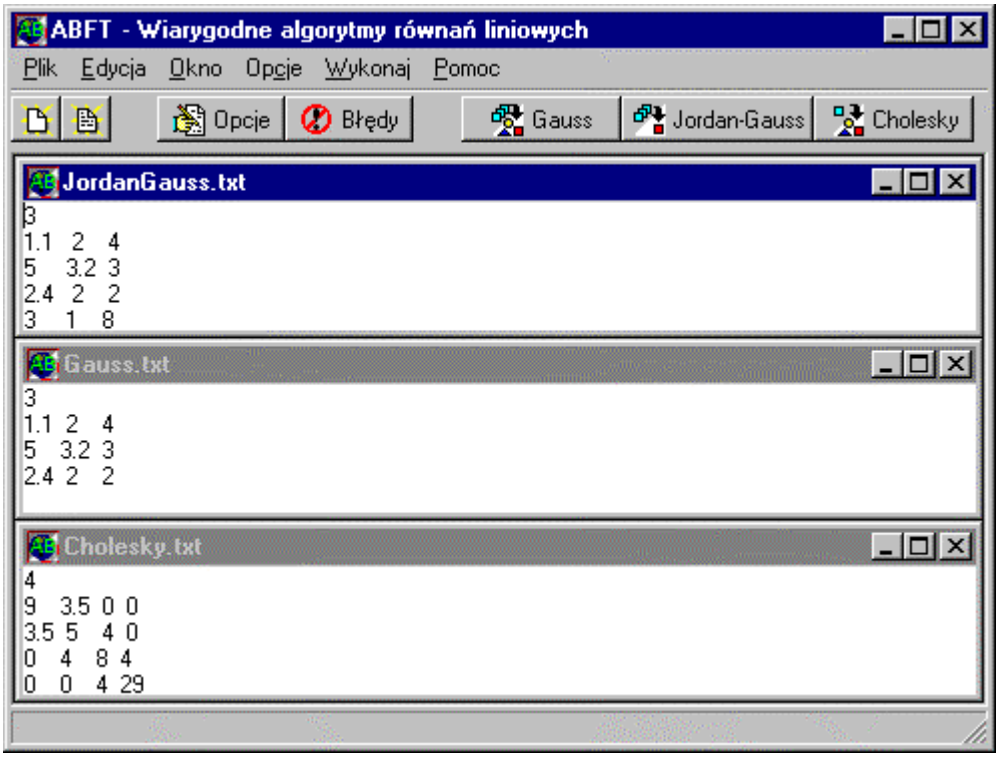
Formaty danych wejściowych i wyjściowych

Środowisko ABFT (*ang. Algorithm-Based Fault Tolerant*) pracuje w systemie operacyjnym Windows 95 i umożliwia badanie wiarygodnych wersji algorytmów (omówionych w punkcie 5): eliminacji Gaussa, Jordana-Gaussa, Cholesky'ego.

Formaty danych wejściowych i wyjściowych

Omawiane środowisko jest aplikacją z Interfejsem wielu dokumentów MDI (*ang. Multiple Document Interface*) i daje możliwość jednoczesnej pracy z wieloma plikami dokumentów. Każdy z tych dokumentów wygląda tak samo, ma takie same składniki i właściwości.

Środowisko ABFT umożliwia edycję wielu plików tekstowych, w których zapisane są dane wejściowe i wyjściowe. Przykładowe pliki podczas edycji w środowisku zawierające formaty danych wejściowych dla poszczególnych algorytmów:



Rys.6.2.1.

Dane wejściowe pobierane są z pliku po uruchomieniu procedury realizującej jeden z algorytmów.

Rozwiązując układ równań liniowych metodą Jordana-Gaussa, plik z danymi wejściowymi powinien zawierać: liczbę elementów macierzy odpowiadającej układowi równań, który rozwiązujemy, macierz oraz wektor wyrazów wolnych.

Jeżeli wykonujemy algorytm eliminacji Gaussa lub Cholesky'ego wystarczy podać liczbę elementów macierzy oraz macierz. Przy czym w przypadku algorytmu Cholesky'ego macierz danych wejściowych jest macierzą symetryczną.

Dane wyjściowe po wykonaniu poszczególnych algorytmów są następujące:

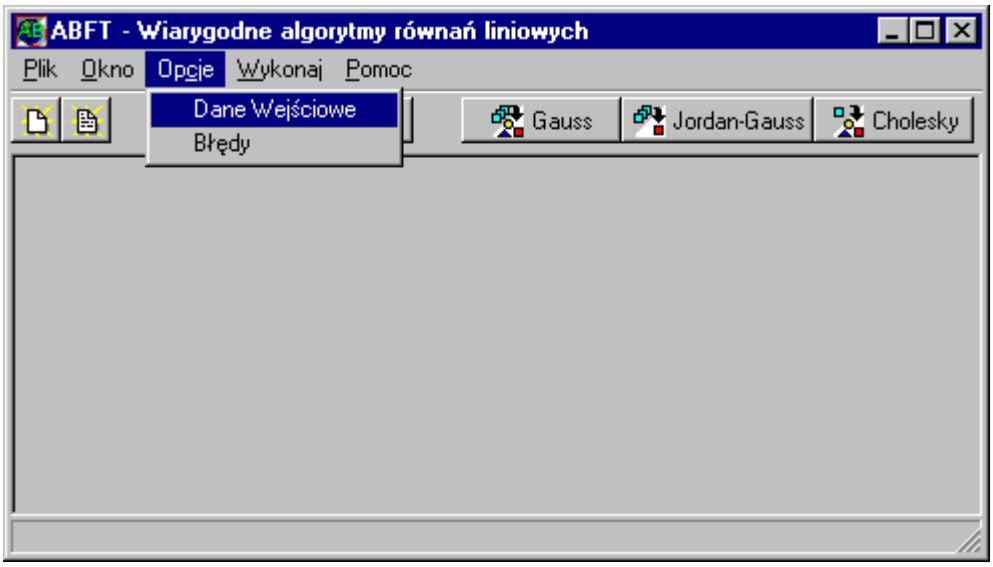
- eliminacja Gaussa – macierz trójkątna górna;
- rozkład Cholesky'ego - macierz trójkątna dolna;
- metoda Jordana Gaussa – rozwiązanie układu równań.

Struktura środowiska. Podstawowe tryby pracy

Aplikacja (środowisko ABFT) zawiera menu główne, które zawiera menu „Plik", „Edycja" (pojawiające się podczas edytowania plików tekstowych), „Okno", „Opcje" i „Wykonaj". Pierwsze trzy wymienione składniki menu głównego związane są z obsługą i edycją dokumentów, natomiast dwa pozostałe służą do badania wiarygodnych wersji algorytmów.

Menu **„Opcje"** umożliwia użytkownikowi ustalenie danych początkowych i wprowadzanie błędów przed rozpoczęciem wykonywania algorytmów Gaussa, Jordana-Gaussa, czy Cholesky'ego:

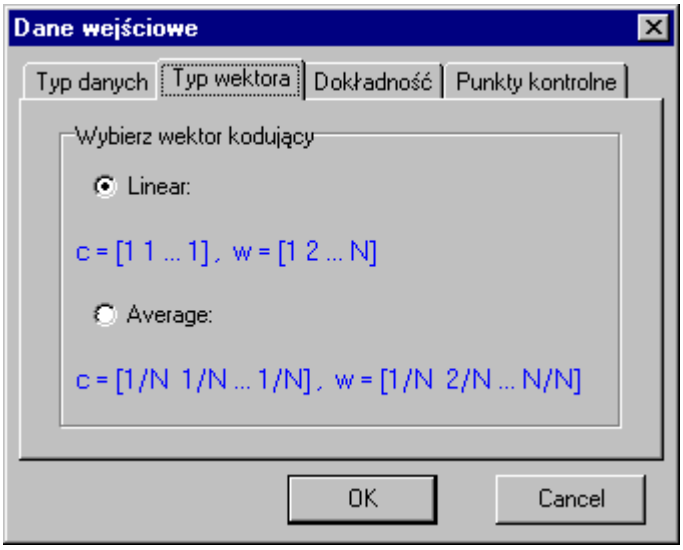
Rys.6.3.1.



Po wybraniu elementu „Dane wejściowe” otwierane jest okno zbudowane z zakładek:

- Typ danych — możliwość wyboru typu danych macierzy wejściowej oraz typu elementów CS i WCS. Podczas wykonywania wybranego algorytmu macierz i sumy kontrolne, którymi uzupełniana jest macierz mogą być typu (w zależności jaki typ wybrał użytkownik): Single, Real lub Double.
  Kod programu z algorytmami zawierającymi różne kombinacje typów danych wejściowych i sum kontrolnych przedstawiony w punkcie nr 9.
  Okno wyboru typów danych wygląda następująco:
  Rys.6.3.2.

- Typ wektora — wybór wektora kodującego (linear lub average), według którego obliczone będą sumy kontrolne CS i ważone sumy kontrolne WCS: Rys.6.3.3.

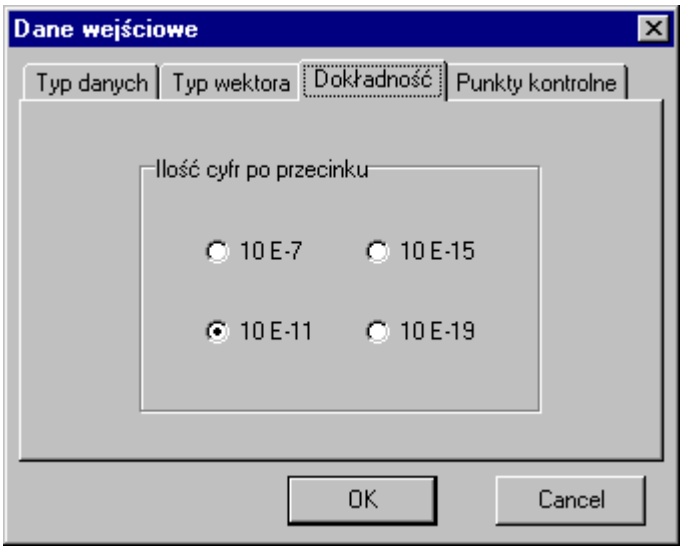


Fragment programu (realizującego algorytm Jordana-Gaussa) obliczenia i dodania do macierzy wejściowej sum kontrolnych:

```
wcsi:=0; csi:=0;
n:=n+2; n1:=n1+2;
for i:=1 to n-2 do
      begin
         for j:=1 to n1-2 do
           if typkod=2 then begin
                                csi:=csi+a[i,j]/(n-2);
                                wcsi:=wcsi+j*a[i,j]/(n-2);
                              end
           else if typkod=1 then begin
                                    csi:=csi+a[i,j];
                                   wcsi:=wcsi+j*a[i,j];
                                 end;
           a[i,n1-1]:=csi; csi:=0; a[i,n1]:=wcsi; wcsi:=0;
      end;
for j:=1 to n1-2 do
      begin
         for i:=1 to n-2 do
           if typkod=2 then begin
```

```
                    csi:=csi+a[i,j]/(n-2);
                    wcsi:=wcsi+i*a[i,j]/(n-2);
                  end
        else if typkod=1 then begin
                        csi:=csi+a[i,j];
                        wcsi:=wcsi+i*a[i,j];
                      end;
        a[n-1,j]:=csi; csi:=0; a[n,j]:=wcsi; wcsi:=0;
    end;
a[n-1,n1-1]:=1; a[n,n1]:=1;
```

- Dokładność — określenie dokładności sprawdzania prawidłowości obliczeń. Rys.6.3.4.

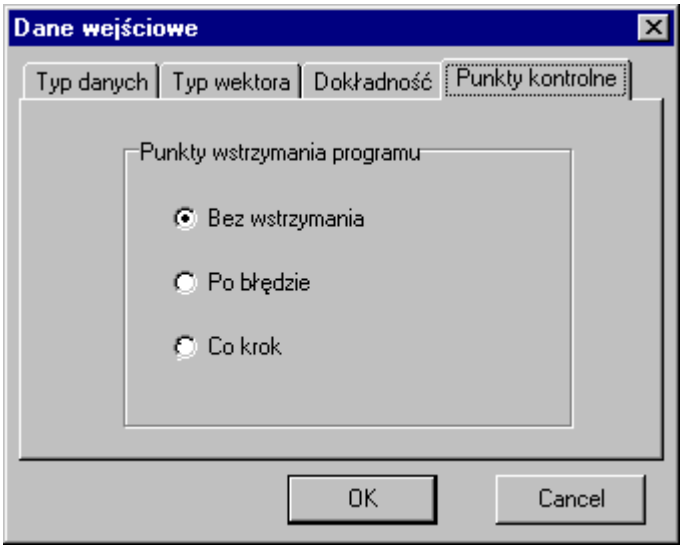


Dokładność jest liczbą określającą, na którym miejscu po przecinku dopuszczalna jest różnica między błędnym elementem a poprawną wartością. Oto przykład wykorzystania wybranej dokładności w programie (w procedurze sprawdzania poprawności obliczeń w kolumnie k macierzy w kroku k podczas wykonywania rozkładu Cholesky'ego):

```
Si:=0; Sj:=0;
for i:=k to n-2 do
 if typkod=2 then begin
                    cs:=cs+a[i,k]/(n-2);
                    wcs:=wcs+i*a[i,k]/(n-2);
                  end
 else if typkod=1 then begin
```

```
                    cs:=cs+a[i,k];
                    wcs:=wcs+i*a[i,k];
                  end;
Si:=a[n-1,k]-cs;
if abs(Si)<=abs(t) then Si:=0;
{t – wybrana dokładność, jeżeli różnica Si jest mniejsza od t to błędu nie ma}
Sj:=a[n,k]-wcs;
if abs(Sj)<=abs(t) then Sj:=0;
wcs:=0; cs:=0;
if (Si<>0) and (Sj<>0) then
  begin
     d:=Sj/Si;     {nr błędnego elementu}
      i:=Round(d);
      {poprawienie błędu:}
      a[i,k]:=a[i,k]+Si;
   end;
```

- Punkty kontrolne — wybór punktów wstrzymania pracy programu (po wystąpieniu błędu lub co krok) w trakcie wykonania algorytmów: Rys.6.3.5.



Podczas wstrzymania pracy programu wyświetlane są rezultaty obliczeń oraz informacje o wystąpieniu błędnych elementów (przykład w punkcie 6.4).

- Wybierając element menu **„Błędy”** mamy możliwość wprowadzania błędnych danych w dowolnym kroku wykonywania poszczególnych algorytmów.

Rys.6.3.6.

Podane przez użytkownika błędy zapisywane są do pliku „faults.dat” i podczas realizacji algorytmów są z niego odczytywane, a następnie wprowadzane do odpowiedniego elementu w macierzy:

```
{.....Odczyt błędów zapisanych w pliku:}
if bl=1 then
 begin
   rec:=0;
   nazwpl:='faults.dat';
   AssignFile(pl,nazwpl);
   RecSize:=SizeOf(Data);
   Reset(pl);
    repeat
      Seek(pl,rec);
      r:=rec+1;
      Read(pl,Data);
      tab[r,1]:=Data.num;
      tab[r,2]:=Data.kr;
      tab[r,3]:=Data.w;
      tab[r,4]:=Data.c;
```

```
            rec:=rec+1;
          until Eof(pl);
         CloseFile(pl);
       end;
      {.....wpis błędów według danych odczytanych z pliku „faults.dat” :}
      if bl=1 then
       begin
        for i:=1 to r do
         begin
          b:=tab[i,2];
          if b=k then
           begin
            l:=tab[i,3];
            j:=tab[i,4];
            if (l>n-2) or (j>n-2) then
             begin
              MessageDlg('Źle podany numer wiersza lub kolumny
podczas'+Chr(10)+
              'wprowadzania błędów',mtError,[mbOK],0);
              Exit;
             end
           else begin
                 a[l,j]:=a[l,j]+StrToInt(war); {war – wartość błędu}
             end;
           end;
         end;
       end;
```

Na wybór metody rozwiązującej równania liniowe pozwala menu **„Wykonaj”** oraz przyciski znajdujące się w pasku narzędzi, który umożliwia szybki dostęp do najczęściej wykonywanych funkcji.

Rys.6.3.7.

Funkcje poszczególnych elementów menu „Wykonaj” to biblioteki (pliki z rozszerzeniem *.dll ze skompilowanymi procedurami wiarygodnych wersji algorytmów). Kod programu metod rozwiązujących równania liniowe przedstawiony w punkcie 9.

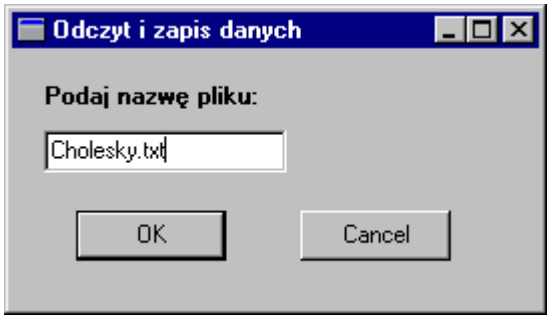
Obsługa programu (z przykładem)

Obsługę i działanie programu przedstawię na przykładzie algorytmu Cholesky’ego.

Procedurę rozkładu macierzy na iloczyn dwóch macierzy trójkątnych metodą Cholesky’ego uruchamiamy w środowisku ABFT przy pomocy elementu znajdującego się w menu „Wykonaj” (Rys.6.3.7.) lub za pomocą przycisku będącego elementem paska narzędzi.

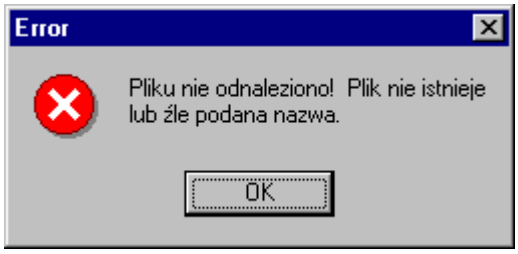
Rezultatem naciśnięcia wybranego przycisku będzie okno dialogowe, do którego należy wprowadzić nazwę utworzonego wcześniej pliku z zapisanymi w nim danymi wejściowymi:

Rys.6.4.1.



Jeśli plik nie istnieje lub nazwa została niepoprawnie wprowadzona spowoduje to błąd zasygnalizowany komunikatem:

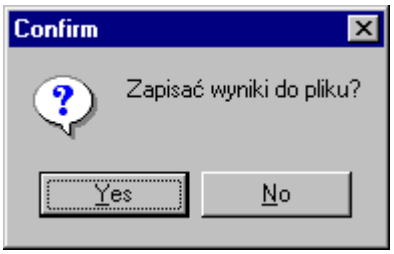
Rys.6.4.2.



Nastąpi przerwanie wykonywania procedury Cholesky'ego, aby kontynuować należy ją ponownie uruchomić.

Jeżeli nazwa została podana poprawnie, na ekranie pojawi się następne okno przy pomocy którego określamy czy dane wyjściowe mają być zapisane do pliku:

Rys.6.4.3.



Jeżeli tak, to podajemy nazwę pliku w taki sam sposób jak na Rys.6.4.1.

Dane początkowe i wprowadzanie błędów realizujemy za pomocą elementów znajdujących się w menu „Opcje" (jak pokazano na rys. od 6.3.2. do 6.3.6.).

Przykład działania programu:

✓ Mamy daną macierz wejściową (symetryczną) zapisaną w pliku „dane.txt" w postaci:

4 (liczba elementów macierzy)
9 3 0 0
3 5 4 0
0 4 8 4
0 0 4 29

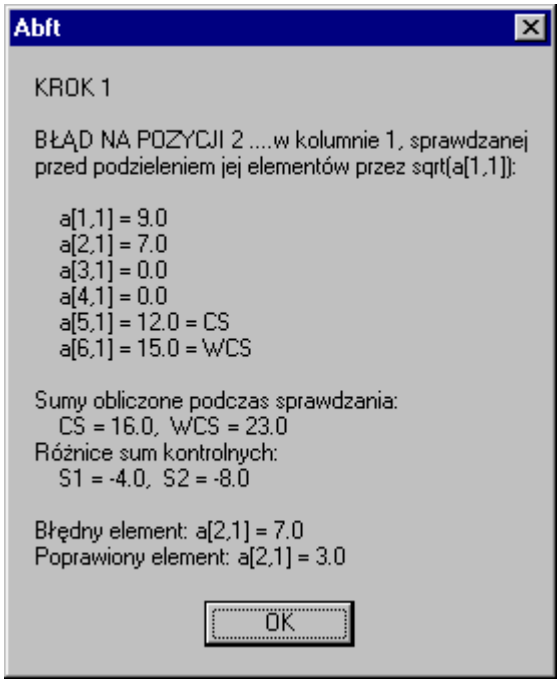
✓ Niech typ danych wejściowych i elementów CS i WCS będzie Real, typ wektora kodowania: linear, dokładność: 1E-7, punkty wstrzymania programu: po wystąpieniu błędu.

✓ Wprowadzone błędy: w kroku nr 1, w kolumnie 1, w wierszu 2; wartość błędu 4.

Uruchamiamy program realizujący wiarygodny algorytm Cholesky'ego (opisany w punkcie 5.3), podajemy nazwę pliku danych wejściowych („dane.txt") i wyjściowych („wy.txt").

W pierwszym kroku w kolumnie znaleziony zostanie błąd co spowoduje zatrzymanie programu i wyświetlenie rezultatów obliczeń:

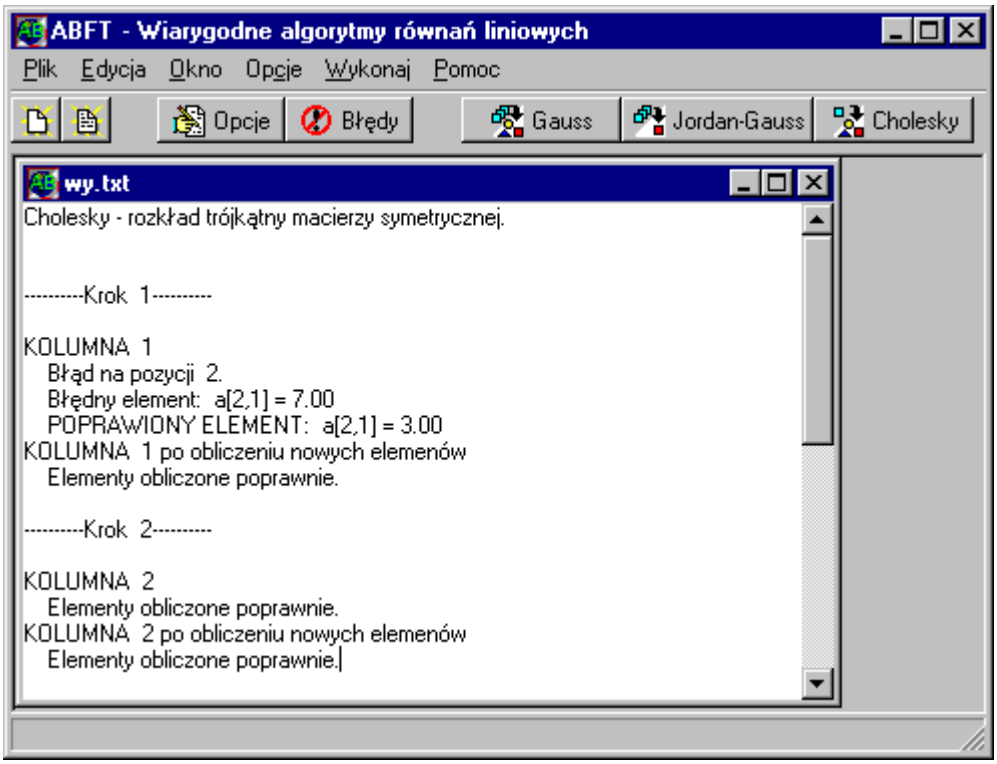
Rys.6.4.4.



Błędny element został poprawiony. Ponieważ została wybrana opcja wstrzymania programu po wystąpieniu błędu, po naciśnięciu klawisza „OK" kolejne kroki algorytmu wykonywane są bez wyświetlania informacji o przebiegu obliczeń.

Wyniki obliczeń możemy obejrzeć w pliku „wy.txt":

Rys.6.4.5.


ABFT - Wiarygodne algorytmy równań liniowych
Plik Edycja Okno Opcje Wykonaj Pomoc
Opcje
Błędy
Gauss
Jordan-Gauss
Cholesky
wy.txt
Cholesky - rozkład trójkątny macierzy symetrycznej.
---------Krok 1---------
KOLUMNA 1
Błąd na pozycji 2.
Błędny element: a[2,1] = 7.00
POPRAWIONY ELEMENT: a[2,1] = 3.00
KOLUMNA 1 po obliczeniu nowych elemenów
Elementy obliczone poprawnie.
---------Krok 2---------
KOLUMNA 2
Elementy obliczone poprawnie.
KOLUMNA 2 po obliczeniu nowych elemenów
Elementy obliczone poprawnie.