

# Exploiting FPGA-based Architectures and Design Tools for Problems of Reconfigurable Computations<sup>\*</sup>

I.Skliarova, A.B.Ferrari

*[iouliia@ua.pt](mailto:iouliia@ua.pt), [ferrari@ieeta.pt](mailto:ferrari@ieeta.pt)*

*Department of Electronics and Telecommunications,  
University of Aveiro,  
3810 Aveiro, Portugal*

## Abstract

*This paper addresses the design and implementation of a configurable “combinatorial processor”, a computational device, which can be used for solving different combinatorial problems. These can be characterized by a set of variables having a limited number of values with a corresponding set of operations that might be applied to these variables. Different mathematical models can be used to describe such tasks. We adopted a matrix representation, which is easier to treat in digital devices.*

*The operations on discrete matrices are unique and cannot be efficiently performed on a general-purpose processor. Although the number of such operations grows exponentially with the number of variables, to solve a particular combinatorial problem a very small number of such operations is usually required. Hence the importance of providing for the dynamic change of operations. The paper presents an approach allowing the run-time modification of combinatorial computations via reloading the RAM-based configurable logic blocks of the FPGAs.*

## 1. Introduction

Practical applications of reconfigurable computing systems already cover quite a wide and diversified field of applications [1], such as image processing, video processing, pattern recognition, neural networks, applications in high-energy physics, search of genetic databases, cryptography, etc. No definite conclusions on how their performance compares with general-purpose processors and DSPs can be drawn. The data available however indicates that reconfigurable computation achieves the best results when dealing with problems exhibiting significant levels of intrinsic parallelism and

requiring the processing of data represented in formats not supported by microprocessor's or DSP's ISAs.

In this paper we investigate the capabilities of FPGAs to be used for computations that are required for NP-hard combinatorial problems. Note that such problems cannot be efficiently solved on general-purpose computers.

Various problems of combinatorial optimization appear in different application areas, such as synthesis and optimization of digital circuits; mapping, placement and routing of microchips, topology and cartography (map-making) [2], artificial intelligence [3], etc. Examples of such problems are: finding the shortest and longest path, graph coloring, Boolean function optimization, covering and planarity problems, encoding problems, etc. Due to the universality and wide scope of applications the problems of combinatorial optimization are well studied and specified through such mathematical models that can be applied to different optimization tasks. Mainly the problems of combinatorial optimization can be formulated in terms of graphs, matrices [3], sets [4], Boolean functions and equations [5], etc.

Given the heterogeneity of combinatorial problems it does not make sense to construct some kind of universal device, which on the one hand would be rather complex and expensive and on the other would not be used in its full power. A particular co-processor might be constructed just for those combinatorial problems that need to be solved. An excellent platform for such kind of devices is based on reconfigurable circuits whose architecture can be customized to the specific problem through reconfiguration. The complexity of recent FPGAs allows to implement a complete combinatorial co-processor with the desired architecture. There are many approaches that make use of FPGA facilities in order to gain advantage of specific circuits [6], such as datapath [7] and memory [8].

This paper is organized in six sections. Section 1 is this introductory section. A reconfigurable processor for

---

<sup>\*</sup> This work was sponsored by the grant FCT-PRAXIS XXI/BD/21353/99

solving combinatorial tasks is considered in section 2. Section 3 suggests the structure of a dynamically modifiable core for the combinatorial processor. Section 4 presents a design example. Section 5 describes the developed design tools for problems of reconfigurable computations. The conclusion is in section 6.

## 2. A configurable combinatorial processor

It is known that the majority of combinatorial problems of logic design and artificial intelligence can be formulated on logic (Boolean, ternary or some other) matrices [3]. All of them are discrete in the sense that the number of values for the elements of these matrices is limited. In practice mainly three values (such as 0, 1 and don't care) are used. Below we will show that considering a fourth value allows to simplify the descriptions of many combinatorial tasks. The primary problem to be solved was presented in [3]. Let A and B denote some finite sets and # be a binary relationship between the sets:  $\# \subseteq A \times B$ . Let [A#B] be a logic matrix of this relationship, which has rows corresponding to elements from A, and columns corresponding to elements from B. For example,  $B = \{b_1, b_2, b_3, b_4\}$ ,  $A = \{B_1, B_2, B_3\}$ ,  $B_1 = \{b_1, b_4\}$ ,  $B_2 = \{b_2, b_3\}$ ,  $B_3 = \{b_1, b_3, b_4\}$ . Here A and B are connected by the relationship of  $\in$  that reflects belonging of elements from B to elements from A. This can be presented in matrix form [A#B]:

$$\begin{array}{cccc} b_1 & b_2 & b_3 & b_4 \\ 1 & 0 & 0 & 1 & B_1 \\ 0 & 1 & 1 & 0 & B_2 \\ 1 & 0 & 1 & 1 & B_3 \end{array}$$

A number of logic circuit design problems might be simplified by solving matrix logic equations  $Z = X \times Y$ , where Z, X and Y are logic matrices, and conjunction ( $\wedge$ ) on the one hand, and disjunction ( $\vee$ ) or exclusive disjunction ( $\oplus$ ), on the other hand, serve respectively as inside and outside operators when multiplying matrices [3]. Some of these matrices are given, and others have to be found. This problem can be decomposed for complex circuits into a set of well-specified and rather simple problems over separate matrices [9]. Independently of a clear specification of such problem most computations over logic matrices are NP-hard, i.e. their complexity depends exponentially on the quantity of input data. Besides, the majority of basic operations implemented on general-purpose processors are not so well suited for such kinds of computations and it is very desirable to accelerate the process of computations using specialized co-processors.

Let us consider an example of some operations over logic matrices. For many practical tasks we need more than 2 values for computations. For instance, in the case of minimization of Boolean functions at least three values are

needed for variables, 0, 1 and don't care (denoted -) and three values for functions, 0, 1 and any value (either 0 or 1). The latter might be also specified by -.

Consider some typical operations that must be applied to Boolean matrices in order to solve different combinatorial problems. For such purposes we will distinguish binary and unary operations that are applied to two and one matrix respectively. Suppose we have two given matrices X and Y. For certain applications they have equal numbers q of rows  $X_1, \dots, X_q$ ,  $Y_1, \dots, Y_q$  (we will use subscript in order to represent rows) and n of columns  $X^1, \dots, X^n$ ,  $Y^1, \dots, Y^n$  (we will use superscript in order to represent columns) and in the general case  $n \neq q$ .

For such matrices we can use traditional binary logic operations, such as OR, AND, XOR, NOR, NAND, etc. which are applied to either two rows  $X_i$  and  $Y_i$  with the same index i or two columns  $X^j$  and  $Y^j$  with the same index j.

For many practical applications, especially in the scope of VLSI design, for a given matrix X we need to construct a new matrix Z, which describes some new relationship extracted from X. Suppose  $X = [x_i^j]$ ,  $i=1, \dots, q$ ,  $j=1, \dots, n$  and it describes a non-directed graph. Examples of such relationships might be the following:

1. Orthogonality:  $X_i(X^i) \text{ ort } X_j(X^j) \Leftrightarrow X_i(X^i) \text{ XOR } X_j(X^j) \neq 0$ ;
2. Intersection:  $X_i(X^i) \text{ int } X_j(X^j) \Leftrightarrow X_i(X^i) \text{ XOR } X_j(X^j) = 0$ , etc.

All these operations can be either unary or binary.

For many practical tasks, such as the covering problem [3], we have to provide search for some specific properties of rows and columns; for example we might want to find a row that contains the maximum number of ones or a column with the minimal number of ones, etc.

An important class of applications includes logic matrices, which describe systems of Boolean functions presented in disjunctive or conjunctive normal forms. In such case matrices X and Y in matrix logic equations  $Z = X \times Y$  will have different numbers of rows and columns [10].

The problem becomes more complicated when dealing with logic matrices  $X = [x_i^j]$ ,  $i=1, \dots, q$ ,  $j=1, \dots, n$  whose components  $x_i^j$  have more than just two values. Typically we need three values, 0, 1 and -. Let's introduce one extra value (denoted +). For example, we can mark with + any unused (or unnecessary) area of a logic matrix. This value can also be interpreted from the point of view of logic. Indeed, if x is input then:

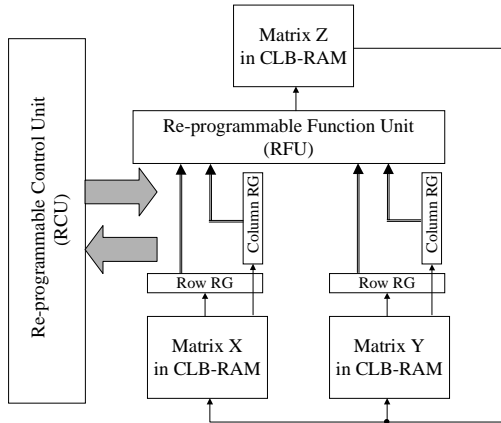
1. The value x might be inverted ( $\bar{x}$ ) denoting 0;
2. The value x might be taken without any change (x) denoting 1;
3. The value x might be ignored (i.e. disconnected) denoting -;

- The value  $x$  might be logically calculated with itself ( $x \# \bar{x}$ , where  $\#$  is some Boolean operation) denoting  $+$ .

We have provided encoding of these four operations with a 2-bit code. So our matrices have been represented as a 2-dimensional array  $X = [x_i^j]$ ,  $i=1, \dots, q, j=1, \dots, n$  where each individual element  $x_i^j$  has 2-bit size. The computational unit for the problem, which we are going to solve, can be specified in the following way:

- The most general task for this unit is calculations on a single or a pair of logic matrices with 2-bit primary elements.
- Most computations are homogeneous, i.e. they invoke the same operation(s) for regular data.
- We have to be able to perform the required operations either on rows or on columns of logic matrices.
- The volume of data is usually very large.

Taking into consideration all these requirements we have proposed the first architecture of a combinatorial processor depicted in fig. 1 that is going to be used for future analysis and estimation.



**Figure 1. Primary architecture of a combinatorial processor**

The processor contains 3 very fast blocks that are based on RAM implemented in CLBs of an FPGA. The size of RAM-based blocks is statically modifiable. They are used in order to store 3 matrices Z, X and Y, where the matrices X and Y are considered to be matrix operands and Z keeps the result of combinatorial computations. The matrices X and Y are loadable from outside and the matrix Z is readable from outside. This can be implemented with a dual-port RAM, as available in the FPGAs of the XC4000XL family. On the other hand the value of Z might be utilized as an operand (either X or Y) in future computations (see fig. 1).

There are two blocks that can be re-programmed during run-time. They are a Re-programmable Function Unit, RFU and a Re-programmable Control Unit, RCU depicted

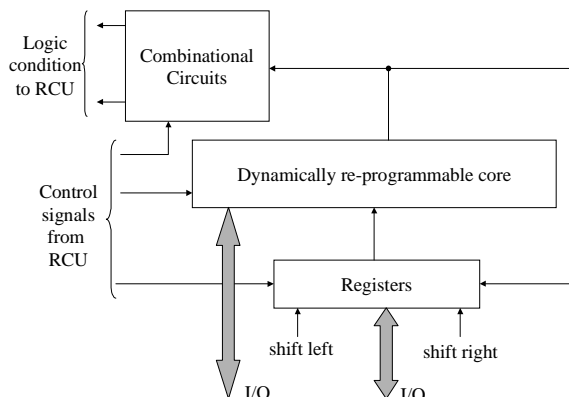
in fig. 1. We are considering two possible physical implementations of such blocks. The first one is based on partially dynamically reconfigurable FPGAs, such as either the XC6200 family of Xilinx [11] or ones that are based on context switching [12]. The second implementation is provided with the aid of statically reconfigurable FPGAs containing rapid re-programmable blocks, such as RAM-based CLBs in the FPGAs of the XC4000XL family. In order to implement RCU and RFU according to the first approach we can use methods and tools considered in [1]. In the case of the second approach we can design the dynamically re-programmable control unit with the aid of methods [13] that allow to construct RAM-based finite state machines with dynamically modifiable behavior.

The RCU might be described in the traditional FSM form as  $S=(A,B,C,\phi,\psi,a_1)$ , where  $A=\{a_1, \dots, a_M\}$  is the set of states,  $B=\{b_1, \dots, b_L\}$  is the set of inputs,  $C=\{c_1, \dots, c_N\}$  is the set of outputs,  $\phi$  is a transition function,  $\psi$  is an output function,  $a_1 \in A$  is an initial state. The values L and N, which are respectively the number of inputs and the number of outputs, are fixed statically. The other parameters, such as A,  $\phi$  and  $\psi$  might be dynamically altered using, for example, the technique [13]. It allows to modify the behavior of RCU and to implement different subsets of operations on the base of the same hardware. Input/output lines  $b_1, \dots, b_L, c_1, \dots, c_N$  have predefined (statically defined) connections with the other components of a combinatorial processor (see fig. 1). However we are able to analyze inputs  $b_1, \dots, b_L$  and to generate outputs  $c_1, \dots, c_N$  in different ways by modifying the parameters A,  $\phi$  and  $\psi$ . Finally it enables us to utilize the same hardware structure depicted in fig. 1 for different kinds of operations on matrices X, Y and Z.

### 3. Structure of dynamically Re-programmable Function Unit (RFU)

We are considering a primary structure of the RFU shown in fig. 2. It includes registers, combinational circuits for widely used computations and dynamically modifiable circuits. The first two components have been realized statically and the functionality of the last component can be altered during run-time. The registers are used in order to keep results of intermediate computations. They can also perform shift left/right functions. Combinational circuits allow to essentially accelerate frequently used computations, such as counting the number of ones in different rows/columns, testing row/column for predefined specific values (such as all zeros or all ones), etc. These circuits generate flags (logic conditions, such as  $b_1, \dots, b_L$ ) that are sent to the RCU in order to specify the desired selection of proper branches of the control algorithm [4]. Depending on the values of logic

conditions the RCU generates a sequence of control signals (such as  $c_1, \dots, c_N$ ) to perform the required operations on logic matrices.



**Figure 2. Primary structure of RFU**

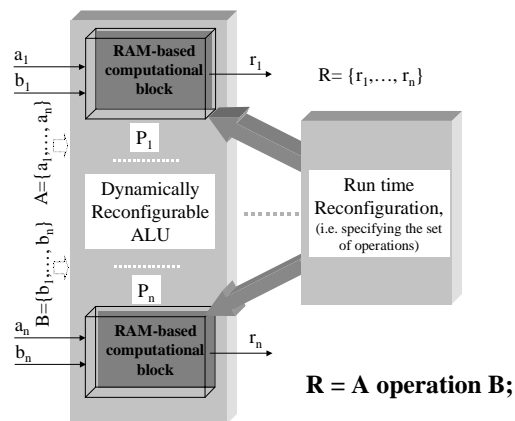
Consider some examples. Suppose it is necessary to perform an operation  $Z = X \oplus Y$ , where  $\oplus$  is an XOR operation and it denotes the following sequence of calculations:  $Z_1 = X_1 \oplus Y_1, \dots, Z_q = X_q \oplus Y_q$ . In this case the RCU will sequentially (in loop) increment the RAM addresses of  $X$ ,  $Y$  and  $Z$  and at each step  $i$  it will force trivial computations  $Z_i = X_i \oplus Y_i$ .

Finally consider an example where  $X_i$  and  $X^j$  are exchanged ( $X_i \leftrightarrow X^j$ ). In this case the RCU provides parallel reading of a row  $X_i$  and then sequential reading of a column  $X^j$ . The latter is performed by sequential reading of a bit  $j$  in the column  $RG$  (see fig. 1) and shifting it with incrementing the address of the respective matrix in RAM (see fig. 1).

Note that there exist a large number of different operations that we might want to perform on logic matrices. It is known that the number of Boolean functions of  $n$  variables is equal  $2^{2^n}$ . So even for trivial Boolean computations (over elements of Boolean matrices with two possible values 0 and 1) we have 16 different functions. In case of 4 feasible values for each element of a matrix, such as 0, 1, -, +, we have 65536 possible Boolean functions that might be performed. On the other hand for any real task we need a very limited number of such functions. That is why it is unreasonable to construct a complicated logic block for the respective computations and we have proposed to provide it with dynamically modifiable functionality (see fig. 3).

In our case the block is built from dynamically re-programmable computational primitives  $P_1, \dots, P_n$  (see fig. 3). Each  $P_i$  has two inputs  $a_i, b_i$  and one output  $r_i$ . The variables  $a_i, b_i$  and  $r_i$  are elements of matrices  $X, Y$  and  $Z$  respectively. Each variable has two bit size, which allows to represent values 0, 1, -, and +. By re-programming of each primitive from  $\{P_1, \dots, P_n\}$  we can implement any

(from 65536) Boolean function of 4 Boolean variables (in fact in any primitive we can realize two such Boolean functions because each block  $P_i$  has two Boolean outputs). The primitives shown in fig. 3 are constructed on the base of 2 RAM-based CLBs of XC4010XL. Run-time re-programmability is achieved with the aid of dual-port RAM organized as 16x2. The first port is used to provide the desired functionality. The second port enables to carry out run-time re-programming of a primitive in order to change the respective Boolean function. Finally the processor has two dynamically modifiable resources that affect the functionality of hardware. They are RFU and RCU with run-time alterable core.



**Figure 3. Dynamically reconfigurable core**

In conclusion we will demonstrate how to use 4 values of logic variables in practice. Let's consider the following product of logic variables:

$$x_1^\alpha x_2^b \dots x_n^\gamma,$$

where  $x^0 = \bar{x}, x^1 = x, x^- = 1, x^+ = 0$ . This representation is more correct than traditional ternary coding (0, 1 and don't care). Indeed there are 4 Boolean functions of one variable, which are  $\bar{x}, x$ , constant 1 and constant 0 and we are considering exactly these functions. The fourth value (+) is very convenient for presenting constants (or inversions for some functions, such as  $\oplus$ ). Consider any sum of logic variables:

$$x_1^\alpha \vee x_2^b \vee \dots \vee x_n^\gamma,$$

where  $x^0 = \bar{x}, x^1 = x, x^- = 0, x^+ = 1$ . This is not the same as in previous case. However the following representation can be applied to both logic expressions:  $x^0 = \bar{x}, x^1 = x, x^- = \bar{x}^+, x^+ = x\# \bar{x}$ , where # is the respective Boolean operation (AND in the first case and OR in the second case). Finally it denotes that if  $x^- = 0$  then  $x^+ = 1$  and vice versa. Since we know how to define don't care for any variable we can find the fourth value for this variable. By definition this value is different from the first three that are 0, 1 and don't care. Depending on the combinatorial problem that is going to be solved, the value + might be interpreted in a different manner. Let us say, for programmable logic arrays or

similar matrix-based circuits [4] this value can mark primary re-programmable units that do not need to be programmed at all. It makes possible to modify the functionality of the circuit after programming even for mask-programmable devices. Note that using the values 0, 1, - and + does not restrict a potential scope of the approach. Indeed, we might ignore the fourth value "+" or even the value don't care "-". Hence in the last case we will deal with the pure Boolean space [3].

#### 4. Design example

Large variety of combinatorial tasks required for solving problems of digital circuit optimization, artificial intelligence, etc. was considered in [3,10] where it was also shown that in case of the use of general-purpose computers the respective procedures involve intensive data exchange between central processor and RAM. As a result they are time consuming. Since for majority of practical applications the size of utilized vectors does not coincide with the size of RAM words we have to use many specific operations that allow to form vectors from RAM words and to divide vectors into RAM words. It invokes many supplementary operations that lead to low effectiveness of the respective combinatorial algorithms implemented in general-purpose computers. On the other hand widely used FPGA architectures are well suited for implementing operations that intensively invoke different data, including feasible pipelining and managing built-in memory. Besides many FPGAs contain RAM-based cells that have a speed comparable with fast gates and can be combined in structures with the desired organization, that in particular provide the proper size of words and the required number of words. Thus we are able to construct what we want. An example of a widely used combinatorial task is presented below.

Let us assume that we have to find the minimal column cover of a Boolean matrix [3,10]. It includes some predefined steps, such as searching for the row with the minimal number of ones, detecting the column, which covers the row to be found (i.e., which has one in this row) and has the maximum number of ones, removing the column to be found and all the rows that it covers, etc. In this case the RCU provides the predefined sequence of steps, that depend on generated logic conditions. The circuits in fig. 2 enable us to count the number of ones and to check which rows will be covered by the selected columns, etc. In order to set a column in a state "has been selected" we can use the fourth (specific) value (+) of its elements that is "not 0", "not 1" and "not -" (see section 2). On the other hand we can use special registers keeping an additional information about the matrix vectors (see fig. 2).

Consider the following matrix X:

$$\begin{array}{cccc}
 x^1 & x^2 & x^3 & x^4 \\
 1 & 1 & 0 & 0 & x_1 \\
 1 & 1 & 0 & 0 & x_2 \\
 0 & 1 & 0 & 1 & x_3 \\
 0 & 0 & 1 & 1 & x_4
 \end{array}$$

Fig. 4 depicts the flow-chart of the algorithm. At the first step row 1 has to be selected (because this is the first among the rows with minimal number of ones). It covers columns 1 and 2. Since column 2 has the maximum number of ones it is chosen in step 2. At the next step it is deleted together with all the rows that are covered by it (i.e. rows 1, 2 and 3). After that the new matrix will look like the following:

$$\begin{array}{ccc}
 x^1 & x^3 & x^4 \\
 0 & 1 & 1 & x_4
 \end{array}$$

Now we have to choose row 4 and column 3. They will be removed from the matrix and the latter becomes empty. As a result we obtain the solution that is represented by the columns 2 and 3.

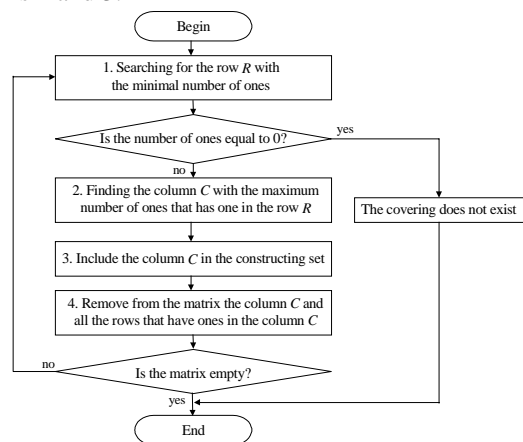


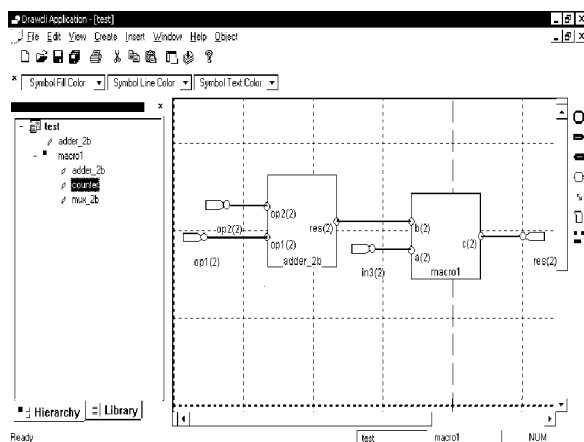
Figure 4. A flow-chart for finding the minimal column cover of a Boolean matrix

The algorithm (see fig. 4) includes one basic operation that is "counting the number of ones in a given vector". This operation will be implemented in the dynamically re-programmable core shown in fig. 2. Note that the flow-chart, shown in fig. 4, actually represents the control algorithm, which has to be realized in the RCU (see fig. 1). This algorithm can be converted to a standard description of a finite state machine (FSM) [4]. In section 3 we have already mentioned that the values L and N for the FSM are fixed. So we can implement the desired behavior just by altering the parameters A, φ and ψ.

#### 5. Design tools for problems of reconfigurable computations

To support the design of FPGA-based computational units a set of tools is being developed with the aid of

Visual C++ 6.0 and the MFC library. The user interface of the application is shown in fig. 5. The two overlapped left-hand windows named *Hierarchy* and *Library* are used to manage the hierarchy of a project and its library respectively. The right-hand window is a schematic editor. The user can create a library of basic elements and then use them in a schematic editor. There are two kinds of library elements: basic elements like adders, decoders, etc. and more complex elements, which are hierarchically composed of the first ones. Having a complete circuit specification it is possible to generate its VHDL description. Then the VHDL code can be analyzed and tested with the aid of the Xilinx Foundation software and implemented in an FPGA. All the experiments have been based on the XC4010XL FPGA.



**Figure 5. User interface of the design environment**

Now we are working on the integration of the developed software with the application GraphBuilder [14], which enables to describe control algorithms in the form of hierarchical graph-schemes, to synthesize the respective control circuits with dynamically modifiable functionality and to implement them in FPGA.

## 6. Conclusion

This paper describes the problems of computations in a discrete (logic) space and presents the results of the design of an FPGA-based combinatorial processor intended to be used as a fast co-processor for general-purpose computers. The processor can execute a subset of combinatorial operations on discrete matrices. The implemented subset can be changed during run-time if required. The primary architecture has been chosen in the assumption that it reflects the universal structure of combinatorial computations in the form of the basic logic equation  $Z = X$

# Y and its varieties. On the other hand the generality of such equation has been shown in publications, such as [3,8,10]. The major components of the combinatorial processor, such as the RFU and the RCU have been implemented and tested in hardware. An integrated environment targeted to the considered problems is being developed. Note that the complexity of new generations of FPGAs is increasing drastically [15]. As a result very complex combinatorial problems can be solved using the proposed approach even with a single FPGA chip.

## References

- [1] V.Sklyarov et al., "Synthesis of Dynamically Reconfigurable Circuits for Embedded Applications", Proceedings the 7th Japanese FPGA/PLD conference, Tokyo, 1999, pp. 31-38.
- [2] T. Tambouratzis, "A Consensus-Function Artificial Neural Network for Map-Coloring", IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics, Vol. 28, N. 5, October 1998, pp. 721-728.
- [3] A.Zakrevskij, "Combinatorial Problems over Logical Matrices in Logic Design and Artificial Intelligence", *Electrónica e Telecomunicações*, vol. 2, no. 2, pp. 261-268, 1998.
- [4] S.Baranov, "Logic Synthesis for Control Automata. Kluwer Academic Publishers, 1994.
- [5] Marco Platzner, "Reconfigurable Accelerators for Combinatorial Problems", *Computer*, April 2000, pp. 58-60.
- [6] S.Brown, "FPGA Architectural Research: A Survey", *IEEE Design & Test of Computers*, Vol. 13, No 4, 1996, pp. 9-15.
- [7] D.Chrepacha, D.Lewis, "DP-FPGA: An FPGA Architecture Optimised for Datapath", *VLSI Design*, Vol. 4, No 4, 1996, pp. 329-343.
- [8] S.Wilton, J.Rose, Z.Vranesic, "Architecture of Centralized Field-Configurable Memory" *Proc. Third ACM Int'l Symp. On Field Programmable Gate Arrays*, Assoc. For Computing Machinery, New York, 1995, pp. 97-103.
- [9] A.Zakrevskij, "Combinatorial theory of logical design. - Automatics and Computers", *Riga, Latvian Academy of Science*, 1990, No 2, pp. 68-79.
- [10] A.Zakrevskij, "Logic synthesis of cascade networks", Moscow, Nauka, 1981.
- [11] Xilinx, XC6200 Field Programmable Gate Arrays, Product Description, April 24, 1997 (Version 1.10).
- [12] Hartej Singh et al., "MorphoSys: A Reconfigurable Architecture for Multimedia Applications", *Proceedings of the XI Brazilian Symposium on Integrated Circuit Design*, Búzios, Rio de Janeiro, 1998, pp. 134-139.
- [13] V. Sklyarov, "Synthesis and Implementation of RAM-based FSM", *Proceedings of FPL'2000*, Villach, Austria, 2000.
- [14] A.Oliveira, A.Melo, V.Sklyarov, "Specification, Implementation and Testing of HFSMs in Dynamically Reconfigurable FPGAs", *Proceedings of the 9th International Workshop FPL'99*, Glasgow, UK, 1999, pp. 314-322.
- [15] 1999 Xilinx Data Book, 1999.