LOGIC DESIGN OF FPGA-BASED COMBINATORIAL PROCESSOR*

IOULIIA SKLIAROVA†, ANTÓNIO B. FERRARI‡

†University of Aveiro, Department of Electronics and Telecommunications, 3810 Aveiro, Portugal, Fax: +351 234 370 545, iouliia@ua.pt
‡ University of Aveiro, Department of Electronics and Telecommunications, 3810 Aveiro, Portugal, Fax: +351 234 370 545, ferrari@ieeta.pt

Abstract. This paper addresses the design of a Reprogrammable Combinatorial Processor (RCP) on the basis of reconfigurable circuits such as FPGA. The RCP is intended to be used for solving different combinatorial problems formulated over discrete matrices. From a structural point of view the RCP is a composition of a Reconfigurable Control Unit (RCU) and a Reconfigurable Function Unit (RFU). Each unit consists of hardwired (fixed) and programmable components. The paper considers and analyses methods, which can be used for logic synthesis and optimisation of RCP based on such decomposition.

Key Words. Combinatorial processor, logic optimisation, finite state machines, FPGA.

1.INTRODUCTION

Many practical applications invoke various combinatorial problems that have to be solved. There is a number of powerful algorithms and software tools that enable us to find their exact and approximate solutions on general-purpose computers. However the computational complexity of some combinatorial algorithms [1] makes it difficult (and sometimes even impossible) to find appropriate results in reasonable time or with available computational resources.

Because of that it is advisable to consider and to implement the respective accelerators such as coprocessors for general-purpose computers. It should be noted that the heterogeneous nature of combinatorial tasks makes it difficult to construct an effective specialized device, i.e. an ASIC, for example. On the other hand, we can analyse different combinatorial problems and select subsets of unique and common operations. As a result we can design a device including a fixed part (for common operations) and a reconfigurable part (for unique operations). This paper considers different aspects and formal methods of logic synthesis for the design of reprogrammable combinatorial processor (RCP) on the basis of reconfigurable hardware, namely commercially available FPGAs.

The paper is organized in 7 sections. Section 1 is this introduction. Section 2 presents architecture for RCP. The design of two major components of RCP, that are a reprogrammable function unit (RFU) and a reprogrammable control unit (RCU), is considered in sections 3 and 4 respectively. Section 5 describes methods that can be used in order to optimise the communication between RFU and RCU. Section 6 describes the process of RCU synthesis and discusses some optimisation techniques. The conclusion is in section 7.

2. ARCHITECTURE OF COMBINATORIAL PROCESSOR

Most combinatorial problems have discrete character. That is why they can be formulated on such mathematical models as graphs [1], sets [2], discrete matrices [3], etc. These models are often convertible in such a way that one model can be formally transformed into another. For example, in [3] it was

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

shown that they might be converted to a universal matrix representation. Logic matrices are very well suited for processing them in FPGA [4]. This property played an important role for the use of matrices as a basic mathematical model for resolving combinatorial problems in RCP.

A sequence of steps needed in order to solve a particular combinatorial problem, can be described by a discrete algorithm that must be applied to matrices, over which the problem has been formulated. The matrices themselves are considered to be input operands. The result might also be presented in a matrix form, but on the other hand it could be a vector or even a numeric value. Many examples of such algorithms are given in [5].

Fig. 1 depicts the proposed structure of a combinatorial processor [6]. The primary architectural blocks for executing operations are RCU and RFU. Both units have been constructed from reprogrammable components, such as RAM-based cells of an FPGA. Thus the functionality of any unit might be changed by modifying the contents of the respective RAM. The RFU is controlled by a set of microprograms generated by RCU for currently considered combinatorial problem. As soon as the problem is solved we can load a new set of microprograms in order to be able to resolve a new combinatorial task. Three selectors shown in fig. 1 provide reading and writing values from/to the desired row/column of matrices.

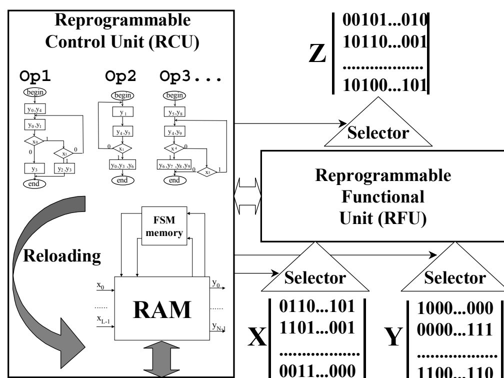

Fig. 1. Basic structure of a combinatorial processor

3. DESIGN OF REPROGRAMMABLE FUNCTION UNIT

The RFU realizes different operations on discrete vectors specified by microprograms of RCU. The structure of the RFU is shown in fig. 2 and it consists of fixed and programmable components. The first component includes the required registers and circuits for implementing typical operations, which can be reused for majority of combinatorial problems. The second component, which is a reprogrammable core, performs primary operations on discrete vectors. Note that on the one hand we can consider a practically infinite number of operations on vectors, but on the other hand any real task requires just a very limited number of them. Indeed the majority of combinatorial algorithms is based on multiple repetition of similar operations that have to be applied to different vectors of a matrix either sequentially or in parallel [5]. This defines the importance of modifications of the reprogrammable core. The idea is the following. On a given step of an algorithm the core is configured for the first set O_1 of operations. The number of such operations is usually very limited. Then the set O_1 will be repeated many times (i.e. it will be periodically applied to many vectors included in the respective matrix). After completing the step and jumping to another step the core will be reconfigured for the new set O_2 of operations. This process will be repeated until the end of the algorithm. Since the number of repeated operations is usually very large, the time of reconfiguration is negligible compared with the execution time.

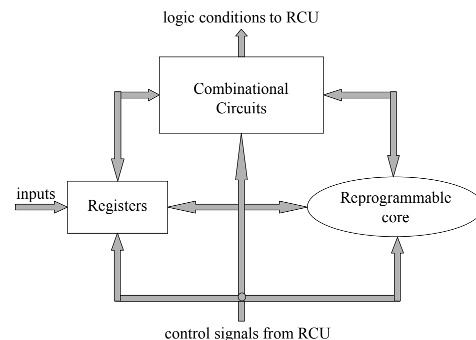

Fig. 2. RFU structure.

The maximum size of vectors n and matrices has been preliminary defined and fixed. Therefore the reprogrammable core is constructed from n RAM-based computational blocks for processing n-component discrete vectors. Thus modifying the RAM contents it is possible to realize various operations on vectors. The core has blocks for the following three types of operations:

- for binary operations on vectors, for example, **a**$\wedge$**b**;
- for operations whose result is either YES or NO, for example, **a** *ort* **b** ?
- for operations whose result can be presented in form of a number having a size that is more than one bit in the general case. For example, it might be necessary co count the number of 1s in a given vector.

4. DESIGN OF REPROGRAMMABLE CONTROL UNIT

The traditional mathematical model of a control unit is a Finite State Machine (FSM) [2]. We will consider the Moore model. At the structural level an FSM can be seen as a composition of a

combinational circuit, which calculates outputs and codes of succeeding states, and a register that stores the current state. We consider a RAM-based FSM that is organized in such a way that the combinational circuit is built from RAM-based FPGA cells. Hence to reprogram the device we have to reload the respective RAM.

In order to minimize the size of RAM the structure presented in fig. 3 has been proposed [7]. In this particular case an address of the Y RAM, that is formed from the code of the current state, points to the RAM word where the required output vector has been stored. The A and P RAM blocks provide conditional state transitions. FSM RAM defines the code of the next FSM state. In order to implement different algorithms of control it is sufficient to reload all the RAMs mentioned above. Note that the sizes of all the RAMs have been predefined and preliminary fixed. As a result the proposed structure permits to realize any algorithm, which satisfies the predefined constraints, such as maximum number of states, inputs, outputs and conditional transitions.

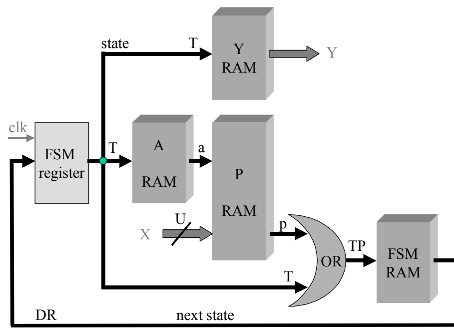

Fig. 3. The structure of RCU.

The process of RCU synthesis in our case assumes the translation of a given algorithm to a set of bitstreams for FPGA. These bitstreams have to be loaded to the RAMs in the predefined structure (in the predefined template). In order to generate the bitstreams we can carry out a set of steps, such as applying the method of fuzzy state encoding [8], replacement of input variables [2], mixing input and state bits [8] and microinstruction encoding. All these steps have been realized in a program written in C++, and as a result can be performed automatically. An entrance for the program is an algorithm of logic control presented in form of a graph-scheme (GS) [2]. The program creates a file with bitsteams that is used in order to program the RCU on the base of the template implemented in FPGA.

5. OPTIMIZATION OF COMMUNICATION BETWEEN CONTROL AND FUNCTION UNITS

The number of inputs and outputs for RCU is defined accordingly by the number of different logic conditions L and microoperations N in a given microprogram. Minimization of these values can be done by combining them in groups using some properties of intercommunications between RCU and RFU.

Microinstructions that are generated by the RCU cause the realization of the desired actions in the RFU. Let $Y=\{y_0,...,y_{N-1}\}$ be a set of microoperations in a microinstruction. At each step (in each cycle i) of microprogram execution we have to form a subset Y_i of Y that must control the RFU in cycle *i*. All the other microoperations $Y_i^0=Y\backslash Y_i$ are supposed to be zero. However some of them can be assigned any value as it does not affect the result of the algorithm execution. Indeed let a microoperation $y_j \in Y_i^0$ reset a counter and during the execution of the current microinstruction Y_i the counter has already been set to zero (for example, a previous microinstruction made this job). Suppose the microinstruction Y_i does not affect the state of the counter. In this case the microoperation y_j can be of any value during the execution of Y_i and this does not affect the functionality of RCP. This property is used to minimize the number of microoperations [9].

Suppose a given algorithm has T microinstructions. For each microinstruction Y_i, i=0,...,T-1, let us build the vector $\mathbf{g}_i = (g_{i0},..,g_{iN-1})$. The set of such vectors can be represented by a matrix **G**. Each element g_{ij} of **G** is equal to:

- 1, if during the execution of Y_i we must provide $y_j=1$;
- 0, if during the execution of Y_i we must provide $y_j=0$;
- -, if during the execution of Y_i the microoperation y_j can be of any value (i.e. either 0 or 1)

Constructing the vectors $\mathbf{g}_i$, i=0,...,T-1, might be done on the basis of the analysis of the desired functionality and logic interface between RCU and RFU.

For each column j (i.e. for each microoperation y_j) of matrix **G** let us consider the vector $\mathbf{v}_j = (g_{0j},..., g_{T-1j})$, j=0,...,N. Each of these vectors specifies an interval of Boolean space $I(\mathbf{v}_j)$ [10]. Let us define the relationships of orthogonality ($\mathbf{v}_j$ ort $\mathbf{v}_k$) and intersection ($\mathbf{v}_j$ int $\mathbf{v}_k$) [10] in such a way that:

$$(\mathbf{v}_j \text{ ort } \mathbf{v}_k) \leftrightarrow (I(\mathbf{v}_j) \cap I(\mathbf{v}_k)) = \varnothing;$$
$$(\mathbf{v}_j \text{ int } \mathbf{v}_k) \leftrightarrow (I(\mathbf{v}_j) \cap I(\mathbf{v}_k)) \neq \varnothing;$$

If ($\mathbf{v}_j$ int $\mathbf{v}_k$), then microoperations y_j and y_k can be replaced with a new one that corresponds to the following interval $I(\mathbf{v}_{jk}) = I(\mathbf{v}_j) \cap I(\mathbf{v}_k)$. In this case the desired functionality of RFU will not be altered.

Note that if ($\mathbf{v}_j$ int $\mathbf{v}_k$), ($\mathbf{v}_k$ int $\mathbf{v}_i$) and ($\mathbf{v}_i$ ort $\mathbf{v}_j$), then we can realize just one from two allowing unions: ($\mathbf{v}_j$ and $\mathbf{v}_k$) or ($\mathbf{v}_i$ and $\mathbf{v}_k$). The objective is to find out

such unions, which allow to minimize the number N. In order to solve this problem we have used the following algorithm [9]. For a given matrix **G** construct the graph Γ, having N vertices, such that each vertex corresponds to the respective microoperation y_i, i=0,...,N-1. Two vertices j and k are connected by an edge if and only if the relationship ($\mathbf{v}_j$ ort $\mathbf{v}_k$) is valid, where $\mathbf{v}_j$ and $\mathbf{v}_k$ are vectors corresponding to the microoperations y_j and y_k. Let us paint the graph Γ in a minimal number of colours χ in such a way that any two connected vertices do not have the same colour. Thus all the vertices having the same colour correspond to vectors $\mathbf{v}_i$, i=0,...,N-1, that are in intersection relationship. As a result the number of microoperations can be reduced up to χ, and it is obvious that $\chi \le N$.

Let us consider an example. Fig. 4 shows a GS of algorithm, which stores some data in a matrix. The actions of the algorithm are clarified with the aid of fig. 5. The discrete vector, that has to be stored in the matrix, is being transferred from external memory. It is assumed that before the execution of the considered fragment of GS the address counters have already been set to the corresponding states. Hence when y_0 is executed the flip-flop FDC is set to 1 in order to specify a memory read. The signal x_0 is appended to the memory output word indicating that the read operation has been completed. According to the GS in fig. 4 if $x_0 = 1$ the microoperation y_1 has to be executed and it causes writing the vector from the memory to the matrix. At the next step the microoperation y_2 resets the flip-flop and increments both counters, preparing addresses for read/write of the subsequent vectors. The logic condition x_1 indicates if the matrix has already been filled.

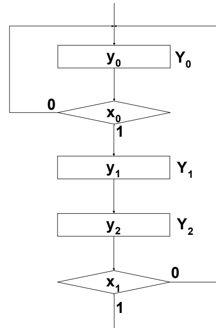

Fig. 4. The fragment of GS for recording a vector to a matrix.

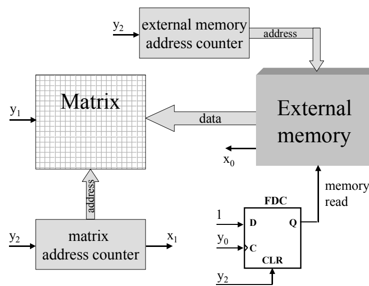

Fig. 5. Transferring vectors from external memory to the matrix.

In accordance with the considered above algorithm we obtain the following matrix **G**:

$$\mathbf{G} = \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{bmatrix} \mathbf{1} & - & \mathbf{0} \\ - & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix} \begin{array}{c} 0 \\ 1 \\ 2 \end{array} \end{array}$$

It says, for instance, that y_1 has a don't care value when we execute the microinstruction Y_0 because independently on what has been really written to the matrix the wrong line will be later properly corrected (i.e. rewritten during execution of Y_1). The graph of orthogonality relationship for the matrix **G** is depicted in fig. 6. As we can see the number of microoperations (i.e. the number of RCU outputs) for implementing the given microprogram can be reduced from 3 to 2 in such a way that the microoperations y_0 and y_1 will be replaced with a single microoperation y_{01} that must be generated for both microinstructions Y_0 and Y_1.

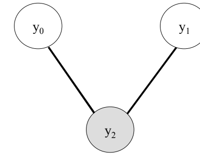

Fig. 6. The graph of the orthogonality relationship.

It is also advisable to minimize the number of logic conditions. In order to do this we can use extra information about possible changes of these conditions during the execution of microprogram, which can be acquired at the stage of analysis of joint functionality of RCU and RFU. Let us designate $S(x_i)$ the set of states, transitions from which depend on logic condition x_i. For each set $S(x_i)$ let us form a vector $\mathbf{e}=\{e_{i0},...,e_{iL-1}\}$, where L is the number of logic conditions. The set of all vectors can be presented in a matrix **E** such that each of its elements e_{ij} is equal to:

- 1, if in all the states from the set $S(x_i)$ $x_j=1$;
- 0, if in all the states from the set $S(x_i)$ $x_j=0$;
- -, if in the states from the set $S(x_i)$ the condition x_j can be of any value (i.e. either 0 or 1).

In each row i of **E** let us change e_{ii} to the value 1 constructing a new matrix $\mathbf{E}^1$. Let us say, that 2 vectors $\mathbf{e}_i^1$ and $\mathbf{e}_j^1$ are in ξ relationship, if and only if $e_{ij}^1=e_{ji}^1\neq -$, i≠j. Let us build a graph Γ, which describes a reverse relationship ξ. Vertices of Γ correspond to logic conditions. Two vertices x_i and x_j are connected with an edge if and only if for the vectors e_i and e_j the relationship ξ has not been satisfied. Let us paint the graph Γ in minimal number of colours χ. Now we can combine those logic conditions, which correspond to the vertices painted in the same colour. For such purpose we can use the following formula $x_{new} = \underset{i \in C}{\vee} (\underset{j \in C}{\wedge} x_j^{e_{ij}^1})_i$, where

$\{x_i,x_j,...,x_k\}$ is the subset of logic conditions that might be joined, $C=\{i,j,...,k\}$, and $x_j^{e_{ij}^1}$ is equal to:

- x_j, if $e_{ij}^1=1$;
- $\bar{x}_j$, if $e_{ij}^1=0$;
- 1, if $e_{ij}^1=-$.

As a result the final number of logic conditions will be equal to χ and it is obvious that $\chi \le L$.

Let us consider an example. Fig 7 depicts a GS of approximate algorithm for finding the minimal column cover of a Boolean matrix [5]. The algorithm can be informally described as follows. First we have to reset all auxiliary registers. Second we have to find out the row, which has the minimum number of 1s. If this row contains all zeros then the cover does not exist. Otherwise we have to choose the column with the maximum number 1s, which also has the value 1 in the selected row. This column is included into the covering, it is deleted from the matrix together with all rows that have the value “1” in this column. We have to repeat these steps until the matrix is empty. Let us consider the purpose of logic conditions in the microprogram. The signal x_0 indicates if we have to treat row/column at the selected address; the signal x_1 states if all rows/columns have been already processed for the current stage, the signal x_2 shows if the selected row has all zeros and finally the condition x_3 says that the matrix is empty.

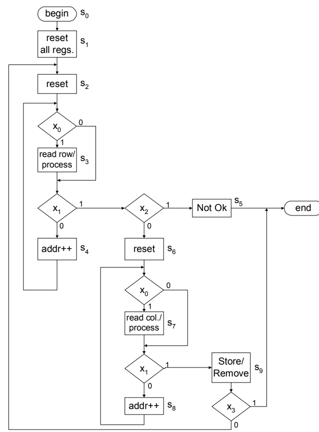

Fig. 7. GS for discovering the minimum column cover of a Boolean matrix.

For the algorithm in fig. 7 we get the following sets $S(x_i)$ and vectors $\mathbf{e}_i$:

$S(x_0)=\{s_2,s_4,s_6,s_8\}$; $\mathbf{e}_0=(-,-,-,0)$;
$S(x_1)=\{s_2,s_3,s_6,s_7\}$; $\mathbf{e}_1=(-,-,-,0)$;
$S(x_2)=\{s_2,s_3,s_4\}$; $\mathbf{e}_2=(-,-,-,0)$;
$S(x_3)=\{s_9\}$; $\mathbf{e}_3=(-,1,0,-)$;

Let us explain the construction of vector $\mathbf{e}_3$. The vector $\mathbf{e}_3$ corresponds to the set $S(x_3)=\{s_9\}$. In state s_9 conditions x_0 and x_3 can get either zero or one values, and that is why the respective components of $\mathbf{e}_3$ are equal to “-“. The logic condition x_1 in the state s_9 is always equal to 1. This is because we can advance to the step of saving the selected column and deleting the respective rows/columns only after processing all rows and columns. The logic condition x_2 in state s_9 is always equal to 0. This is because at a step of columns handling it is guaranteed that the selected row has at list one value “1”. The set of vectors $\mathbf{e}_i$ can be described in form of matrices $\mathbf{E}$ and $\mathbf{E}^1$:

$$\mathbf{E}=\begin{vmatrix} - & - & - & \mathbf{0} \\ - & - & - & \mathbf{0} \\ - & - & - & \mathbf{0} \\ - & \mathbf{1} & \mathbf{0} & - \end{vmatrix} \qquad \mathbf{E}^1=\begin{vmatrix} 1 & - & - & \mathbf{0} \\ - & 1 & - & \mathbf{0} \\ - & - & 1 & \mathbf{0} \\ - & \mathbf{1} & \mathbf{0} & 1 \end{vmatrix}$$

Let us build the graph of the reverse relationship ξ for the considered example (see fig. 8) and paint it with a minimal number of colours. As a result we obtain $\chi=3$, and thus we can reduce the number of logic conditions in the microprogram from 4 to 3. Instead of logic conditions x_2 and x_3 we can use the new condition calculated as follows:

$$x^{new} = x_2\bar{x}_3 \vee \bar{x}_2 x_3$$

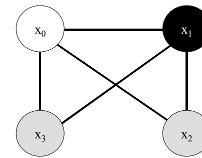

Fig. 8. The graph of the reverse relationship ξ.

Note that x^{new} can be further minimized. In order to do this we have to analyse the matrix $\mathbf{E}$, and select those codes of variables R_{01}, on which the function x^{new} gets either the values 1 or 0. In our case we have $R_{01}=\{-0,0-\}=\{00,01,10\}$. The set of all possible two-bit codes is $R_B=\{00,01,10,11\}$. Thus the codes in which the function x^{new} is not defined is $R=R_B \backslash R_{01}=\{11\}$. Following [9], the discovered codes R might be used to minimize the function x^{new}. Finally we obtain:

$$x^{new} = x_2\bar{x}_3 \vee \bar{x}_2 x_3 \vee x_2 x_3 = x_2 \vee x_3 .$$

6. OPTIMIZATION TECHNIQUE

When we perform the synthesis of the RCU it is necessary to solve a number of problems, which permit to optimise the structure of RCU. Let us consider some of these problems.

6.1. Distribution of input variables

The process of synthesis assumes replacement of input variables from the set $X=\{x_0,...,x_{L-1}\}$ with new variables from the set $P=\{p_0,...,p_{G-1}\}$ in such a way that $G \ll L$ [2]. Let $X(s_m) \subseteq X$ be a subset of input variables that affect transitions from the state s_m, m=0,...,M-1. Note that as a rule the condition $|X(s_m)| \ll L$ is satisfied for all $s_m \in S$ [2]. This characteristic permits to reduce the number of RCU

input lines after the replacement of input variables. If $G = \max_{s_m \in S} |X(s_m)|$, then each variable $x_l \in X(s_m)$, m=0,...,M-1, can be replaced with variable $p_g \in P$. If the FSM is in a state s_m, then $p_g=x_l$. If FSM is in another state s_k, $k \neq m$, then the variable p_g can be assigned any value and, in particular, it can replace another input variable $x_j \in X(s_k)$.

Note that if for a given algorithm the value G is greater than some pre-defined constraint, such as the number of logic conditions U that are used in order to form an address of P RAM shown in fig. 3, then we cannot realise this algorithm in a RAM-based FSM. However there exists a way to solve this problem. It can be done by adding some dummy states, which allows to decrease the number of logic conditions that affect individual state transitions. As a result we can reduce the value of G.

For such purpose we can use the following method [11]. Let two logic conditions x_i и x_j be in a linkage relationship if an output of a GS conditional node containing one of these two conditions is connected by an arc with an input of another GS conditional node containing another one of these two conditions. Let us build a weighted graph Γ, whose vertices correspond to logical conditions. Two vertices of Γ x_i and x_j are connected with an edge if and only if x_i and x_j are in linkage relationship. Each edge is assigned a weight w, which indicates how many times the respective logic conditions have been linked. At the beginning all weights are supposed to be zero. Then it is necessary to analyse inputs of all conditional nodes of the GS. If an input of a conditional node containing x_j is connected by arcs with outputs of f conditional nodes, containing variables x_{i0},...x_{if}, then the weights of edges (x_j,x_{i0})... (x_j,x_{f0}) have to be increased by 1/f. In order to constrain the value of G we need to cut the graph Γ into a minimum number of independent sub-graphs such that for each sub-graph the condition $|X_i| \leq U$ is satisfied, where X_i is the set of logic conditions, which are associated with the vertices of the respective sub-graph. When we cut the graph Γ we delete some edges. If we delete edges that totally have weight w, then we have to add to GS $\lfloor w \rfloor$ extra states. In this case new states have to be added in between those conditional nodes, which contain logic conditions that were connected in the graph Γ by the deleted edges. Thus in order to minimize the number of new states we have to cut the graph in such a way that the total weight of erased edges would be minimal.

Cutting the graph onto sub-graphs can be carried out with the aid of the following algorithm. First the graph Γ is cut onto 2 sub-graphs such that the first one has U vertices and the second one L-U vertices. Then we add all the required extra states and correct if necessary the weights of remaining edges (because adding some new states can lead to decreasing the weights of the remaining edges or even to deleting some of them). Then from the sub-graph with L-U vertices we extract again a new sub-graph with U vertices. Similar actions have to be performed until Γ will not be decomposed into sub-graphs with the desired properties (i.e. into such sub-graphs that each of them has no more than U vertices).

Note that addition of extra states lead to the situation where some state transitions, which could be executed during one clock cycle in the initial FSM, will now require two or even more clock cycles. It decreases the speed of the FSM. However for a majority of practical applications this is admissible.

Let us consider an example. Suppose that for the GS depicted in fig. 9 we have to synthesise a control unit based on the template shown in fig. 3 and let U=2. The graph Γ, constructed from the given GS is presented in fig. 10. In this case each extracted sub-graph must have not more than 2 vertices. Thus, first we extract sub-graph Γ_1, containing logic conditions x_0 and x_1. Since the total weight of the deleted edges is w=2, we need to add two additional states s_{10} and s_{11} in between conditional nodes keeping x_1-x_2 and x_0-x_4 accordingly (see fig. 9). In this case the new states do not change the weights of the remaining edges. At the succeeding step let us extract sub-graph Γ_2, containing logic conditions x_2 and x_3. Since the total weight of the deleted edges is w=1.5, we need to add one additional state s_{12} in between conditional nodes having x_3 and x_4 (see fig. 9). The remaining vertices x_5 and x_4 compose the last sub-graph Γ_3. Finally in the resulting GS there is no state transition affected by more than 2 logic conditions. It means that the set $X=\{x_0,...,x_{L-1}\}$ of input variables might be replaced with a new set $P=\{p_0,...,p_{G-1}\}$ and G=2.

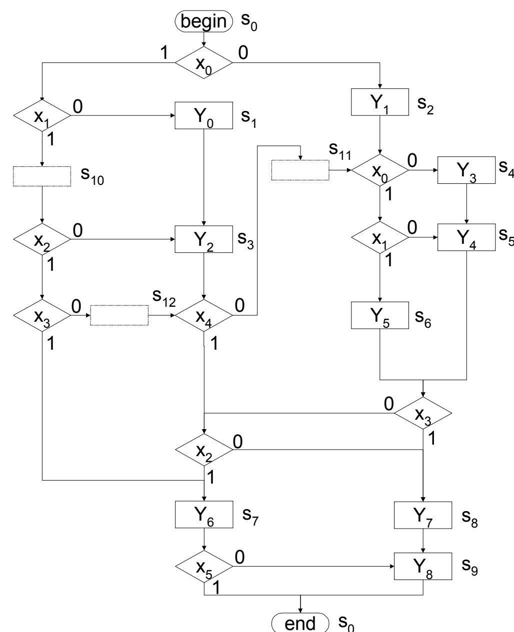

Fig. 9. An example of a GS for reducing the value of G.

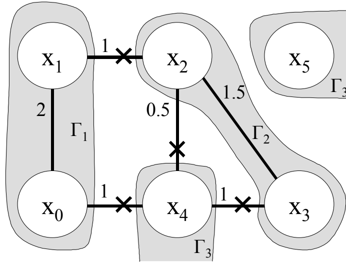

Fig. 10. Graph that reflects linkage relationship.

6.2. State encoding

The state assignment is based on the algorithm, which enables us to minimise the size of address and output buses for the FSM RAM (see fig. 3). We have used the technique of assignment based on codes with variable length (such codes are called fuzzy codes). It allows to mix bits that represent state codes with variables from the set $P=\{p_0,...,p_{G-1}\}$. This permits to decrease the size of the FSM RAM essentially.

Let K be the size of the FSM RAM address bus. In the proposed algorithm we are trying to minimise it in such a way that $K \to \lceil \log_2 M \rceil$, where M is the number of FSM states. Let $S=\{s_0, ...,s_{M-1}\}$ be the set of FSM states, $X=\{x_0,...x_{L-1}\}$ the set of FSM input signals, $Y=\{y_0,...y_{N-1}\}$ the set of FSM output signals, $T=\{t_0,...,t_{R-1}\}$ the set of all state transitions. Each state transition t_i can be presented in the following form: $t_i=\{s_{from}, s_{to}, X(s_{from}, s_{to})\}$, i=0,...,R-1, where s_{from}, $s_{to} \in S$, s_{from} is an initial state in the state transition and s_{to} is the next state, $X(s_{from}, s_{to})$ is the function of input variables that causes transition from s_{from} to s_{to}. When there are two state transitions $t_i=\{s_{from_i}, s_{to_i}, X(s_{from_i}, s_{to_i})\}$ and $t_j=\{s_{from_j}, s_{to_j}, X(s_{from_j}, s_{to_j})\}$, and if $s_{from_i}=s_{from_j}$ then t_i and t_j have a common fanin state and if $s_{to_i}=s_{to_j}$ then t_i and t_j have a common fanout state [12]. Let us divide the set T into M subsets: $T=\{TFI_0,...,TFI_{M-1}\}$, $TFI_0 \cap \ldots \cap TFI_{M-1}=\varnothing$, $TFI_0 \cup \ldots \cup TFI_{M-1}=T$ in such a way that each subset TFI_i, i=1,...,M-1, is composed of state transitions with a common fanin state. Thus all the transitions from the state s_0 form the set TFI_0, all the transitions from the state s_1 form the set TFI_1 and so on. Now let us consider the set T as a union of the following subsets: $T=\{TFIO_0,...,TFIO_{Q-1}\}$, $0 \le Q \le M$, $TFIO_0 \cap \ldots \cap TFIO_{Q-1}=\varnothing$ and $TFIO_0 \cup \ldots \cup TFIO_{Q-1}=T$, and each element $TFIO_i$, i=0,…,Q-1, is composed of such subsets TFI that have more that 1 element and include state transitions with a common fanout state. Obviously, if the control algorithm does not have conditional transitions then $TFIO_i=t_i$, i=0,...,R-1.

According to the method [8] each state transition t_i, i=0,...,R-1, has to be assigned a code $C'(t_i)$, and each set TFI_i, i=0,...,M-1, has to be assigned a code $C(TFI_i)$ (note that $C(TFI_i)=C(s_i)$), with the following requirements:

- $C(TFI_0)$ ort … ort $C(TFI_{M-1})$, i.e. the codes of all FSM states must be mutually orthogonal;
- $C'(t_0)$ ort … ort $C'(t_{R-1})$, i.e. the codes of all state transitions must be mutually orthogonal except if they have a common fanout state;
- $C(TFI_i)=C'(t_0) \wedge \ldots \wedge C'(t_{|TFI_i|})$, $t_0,...,t_{|TFI_i|} \in TFI_i$.

In order to provide this we use the following algorithm:

1. Calculate the value $K = \lceil \log_2 |M| \rceil$.
2. From the sets $TFIO_i$, i=0,…,Q-1, that have not been yet considered select the set $TFIO_{max}$ with the maximum number of elements: $|TFIO_{max}| \ge |TFIO_i|$, i=0,...,Q-1.
3. From the sets TFI_i, i=0,…,M-1, that have not been yet considered select the set TFI_{max} with the maximum number of elements: $|TFI_{max}| \ge |TFI_i|$, $TFI_{max} \subseteq TFIO_{max}$, $TFI_i \subseteq TFIO_{max}$.
4. Calculate the minimum number of bits B, by which the codes of state transitions that belong to TFI_{max} must differ: $B=\lceil \log_2 |TFI_{max}| \rceil$.
5. Select in the set TFI_{max} a transition t_{max}, which has a common fanout state with a maximum number of elements of $TFIO_{max}$.
6. Verify if we can assign to the selected transition t_{max} the code of any state transition t_{exist} that has been already considered and treated. It is possible if and only if the following two conditions are satisfied:
   - The state transitions t_{max} and t_{exist} have a common fanout state;
   - $C(TFI_{max})$ ort $C(TFI_i)$, i=0,...,M-1, $i \ne max$, i.e. the codes of all sets TFI_i, i=0,...,M-1, $i \ne max$, are still mutually orthogonal.
7. If we are not able to use any code, which has already been assigned, then it is necessary to arrange a new code such that the number of bits that distinguish different codes from the set TFI_{max} is less than or equal to B. If this condition is satisfied, verify if the codes in all the sets TFI_i, i=0,...,M-1, are still mutually orthogonal, i.e. $C(TFI_{max})$ ort $C(TFI_i)$, i=0,...,M-1, $i \ne max$. If this is true then use this code. Otherwise try to find another code. If we are not able to find out the solution increment B and repeat point 7.
8. If we are not able to code all the states by K bits, increment K and repeat all the steps starting from point 2.

Let us consider an example. Suppose that a control algorithm has been described by a GS depicted in fig. 11. It can be formally converted to a structural table of the respective FSM [2]. For our example we have M=6, L=4, R=10, Q=4 and $T=\{t_0,...,t_9\}=\{TFI_0,...,TFI_5\}$, $TFI_0=\{s_0,s_1,1\}$, $TFI_1=\{ \{s_1,s_2, \bar{x}_0x_1\}, \{s_1,s_3, \bar{x}_0 \bar{x}_1\}, \{s_1,s_1,x_0\}\}$ and so on. Fig. 12 depicts the sets $T=\{TFIO_0, ...,TFIO_3\}$.

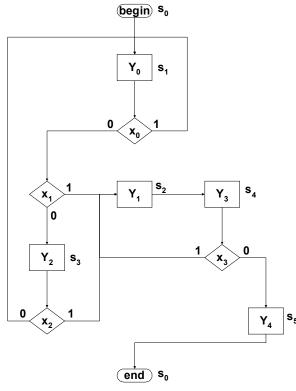

Fig. 11. A GS that describes a given control algorithm.

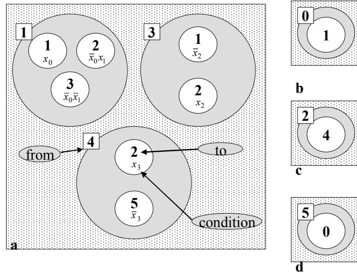

Fig. 12. The sets $TFIO_i$, i=0,...,3

Applying the proposed algorithm to the GS in fig. 11 enables us to find out the solution presented in table 1. The column s_{from} contains all the states, the columns $C(s_{from})$ and $C(s_{to})$ keep the codes of the respective states, the column $X(s_{from}, s_{to})$ presents the function of input variables that cause transitions from s_{from} to s_{to}. The column *FSM RAM* shows the addresses of the FSM RAM where the respective codes of the next states $C(s_{to})$ will be recorded. In each code $C(s_{from})$ the characters *i* point to those bits, which distinguish addresses of the states with a common fanin state. Let us enumerate the positions of *i* in each code from right to left, starting from zero and associate with them a vector **p**. The column ***x→p*** shows the values of **p** for each transition. For example, in the transition *t* from s_1 to s_1, we have $C(s_{from})=C(s_1)=1ii$ and $C'(t)=110$, thus **p**=010.

As we have already mentioned this algorithm has been implemented in a program written in C++. The program performs all the required calculations and generates the file to be used for programming the RCU, which will realize the given algorithm. In order to program the FSM RAM and the Y RAM (see fig. 3) the respective codes from the columns *FSM RAM* and *$C(s_{from})$* of table 1 are used. In order to program the A RAM all the states with branches (i.e. the states s_1, s_3 and s_4 shown in fig. 11) are sequentially coded by binary codes **a** with minimum size. All the states with unconditional transitions get the code "0…0". In order to program the P RAM we use the contents of the column ***x→p***, i.e. in the address that combines **x** and **a** (see fig. 3), we have to write the code from the respective row of the column ***x→p***.

6.3. Microinstruction encoding

The next step of synthesis is aimed at the optimisation of the components of FSM that generate outputs. It is done through microinstruction encoding and placement of the resulting codes into RAM taking into account all the constraints, such as the number of physical inputs/outputs, etc. In order to reduce the size of microinstructions, which defines the number N of outputs for Y RAM (see fig. 3), we can use different algorithms for special encoding. Let us consider one of them [9].

Let the RCU generate microinstructions from the set Y. Two microoperations y_i and y_j are compatible if and only if they appear together at least in one microinstruction Y_k. Let us construct the compatibility graph Γ. Microoperations correspond to vertices of Γ and two vertices y_i and y_j are connected with an edge if and only if the microoperations y_i and y_j are in a relationship of compatibility. In order to activate any microoperation y_i it is sufficient to attach it to one allocated field F_j, j=0,...,H-1 within the microinstruction.

Table 1. Structural table that keeps initial and final data for synthesis

s_{from}	$C(s_{from})$ = $C(TFI_{from})$	$X(s_{from}, s_{to})$	X → p	s_{to}	$C(s_{to})$	FSM RAM = $C'(t)$
s_0	000	1	000	s_1	100	000
s_1	1ii	x_0	010	s_1	100	110
	=	$\bar{x}_0 x_1$	011	s_2	010	111
	100	$\bar{x}_0 \bar{x}_1$	000	s_3	110	100
s_2	010	1	000	s_4	101	010
s_3	11i =	$\bar{x}_2$	000	s_1	100	110
	110	x_2	001	s_2	010	111
s_4	1i1 =	x_3	010	s_2	010	111
	101	$\bar{x}_3$	000	s_5	011	101
s_5	011	1	000	s_0	000	011

To minimize the size of microinstructions it is necessary to minimize the number of fields H and associate each microoperation y_i with just one field.

Thus the problem of distribution of microoperations between the fields of the microinstruction can be reduced to finding the minimum number H of subsets $F_0,...,F_{H-1}$, such that each subset F_i, i=0,…,H-1, contains only incompatible microoperations. This task can be solved by painting the graph Γ with a minimal number of colours. As a result all the microoperations corresponding to vertices with the same colour are incompatible and that is why they can be included in the same field.

Let us consider an example. The compatibility graph Γ constructed for the GS in fig.13 is depicted in fig. 14. The minimum number of colours for this graph is equal to 3. Thus we can introduce the fields F_0 (y_0, y_1, y_3, y_6, y_8, y_9), F_1 (y_2, y_4, y_7) and F_2 (y_5) with the respective microoperations in parenthesis. As a result the size of F_0 is equal to 3, the size of F_1 – 2 and the size of F_2 - 1. The total size of microinstruction is equal to 6. Note that if we apply one-hot encoding technique the size of microinstructions would be 10.

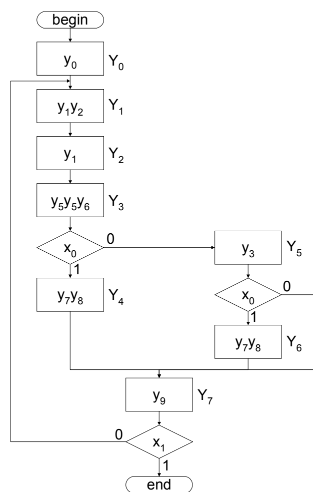

Fig. 13. An example of GS for microinstruction encoding.

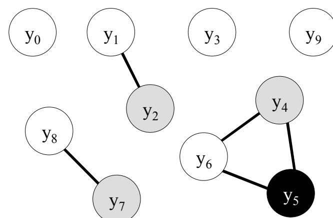

Fig. 14. Graph of compatibility of microoperations.

7. CONCLUSION

We have proposed a set of formal methods that can be used for the design of a combinatorial processor with a reconfigurable core on the basis of commercially available FPGAs. The modifications of the core, which are required for implementing different combinatorial algorithms, have been provided through reprogramming of RAM-based FPGA cells. The top-level architecture of the processor includes two primary blocks that are the reprogrammable control unit (RCU) and the reprogrammable function unit (RFU). The considered technique allows to take into account the mechanisms of communication between RCU and RFU and to minimize the number of connections between them. The paper also presents a set of methods that can be used for optimisation of RCU at different steps of synthesis.

8. REFERENCES

1. Micheli G. (1994). Synthesis and optimization of digital Circuits. McGraw-Hill, Inc., 570 p.
2. S.Baranov, "Logic Synthesis for Control Automata", Kluwer Academic Publishers, 1994.
3. A.Zakrevskij, "Combinatorial Problems over Logical Matrices in Logic Design and Artificial Intelligence", Electrónica e Telecomunicações, vol. 2, no. 2, pp. 261-268.
4. I.Skliarova, A.B.Ferrari, "Exploiting FPGA-based Architectures and Design Tools for Problems of Reconfigurable Computations", Proceedings of the SBCCI 2000 XIII Symposium on Integrated Circuits and System Design, Manaus, Brazil, September 2000, pp. 347-352.
5. A.D.Zakrevski, "Logical Synthesis of Cascade Networks", Moscow: Nauka, 1981.
6. I.Skliarova, A.B.Ferrari, "Development tools for problems of combinatorial optimization", Proceedings of the 4th Portuguese Conference on Automatic Control (CONTROLO'2000), Guimarães, Portugal, October 2000, pp. 552-557.
7. I.Skliarova, A.B.Ferrari, "Synthesis of reprogrammable control unit for combinatorial processor", submitted to IEEE DDECS 2001.
8. V.Sklyarov, "Synthesis and Implementation of RAM-based Finite State Machines in FPGAs", Proceedings of FPL'2000, Villach, Austria, August, 2000, pp. 718-728.
9. S.Baranov, L.Zhuravina, V.Sklyarov, "Computer Aided Design", Minsk, High School, 1981.
10. A.D.Zakrevski, "Algorithms of synthesis of discrete automata", Moscow: Nauka, 1971.
11. S.I.Baranov, V.A.Sklyarov, "Digital Devices Based on Programmable Matrix LSI", Moscow: Radio and Communications, 1986.
12. G.D.Hachtel, F.Somenzi, "Logic Synthesis and Verification Algorithms", Kluwer Academic Publishers, 1996.