Evolvable Hardware Implemented by FPGA

Petr Burian
University of West Bohemia in Pilsen, Department of Applied Electronics and Telecommunications,
Univerzitni 26, 30614 Pilsen, Czech Republic,
e-mail: burianp@kae.zcu.cz

Abstract **– This paper deals with the implementation of the evolvable hardware by FPGA devices. The present work examines the dependence of particular aspects of the genetic algorithm for the evolvable hardware domain usage. Practical implementation of this algorithm by the FPGA circuit is researched as well. In the paper, two practical examples of evolvable circuits are shown – the combinational logic circuit and the evolutionary FIR filter.**

INTRODUCTION

The evolvable hardware domain is a relatively new and modern method of designing circuits. It is based on reconfigurable structures. The evolvable system can change its function and parameters dynamically in time by means of configuration of these structures. The key problem of this domain is the search for suitable configuration; a suitable configuration produces a system (circuit) that provides required function. The design of evolvable hardware consists of two main tasks. The first task presents the selection of useful reconfigurable structure. The second one is about finding a suitable configuration for this structure; the system has to solve the optimization problem.

Various types of circuits – digital or analogue – can be developed by this method. From the point of view of an easier reconfiguration it is obvious that mainly digital structures are used. The use of FPGA devices is very appropriate. However, there are lots of projects that deal with evolvable analogue circuits.

The search for suitable configuration of reconfigurable structures is almost entirely assured by using evolutionary algorithms. These are special tools for solving optimization tasks. The principles of their work are based on biological processes.

EVOLUTIONARY ALGORITHMS

As has already been noted, evolutionary algorithms (evolutionary computational techniques) are profitable tools for optimization tasks solution. They are based on the Darwin's theory of evolution and survival of the fittest and make it possible to solve the optimization tasks that can be defined as a search for global/local maximum/minimum functions. However, their function does not have to be based only on Darwin's theory; there are algorithms that make use of other biological principles. For example, the Self Organizing Migrating Algorithm (SOMA) imitates herd behaviour, etc.

The basic representative of this group of algorithms is the Standard Genetic Algorithm (SGA). It is a relatively simple algorithm (*fig. 1*), the basic element of which is an individual. A group of individuals forms a population. An individual represents just one solution to the optimization problem. Various representations can be used for individuals, for example the classical binary format is used very often. Note that a suitable version of representation is very important for correct function of the evolution.

At first, the algorithm has to initiate individuals of a population, which is achieved by filling them with random values. Indeed, it is possible to use other methods of the initialization of the population (based on certain knowledge of the application). Afterwards, recombination operators are applied to the population. Generally, only two operators – the crossover and the mutation – are used. These operators change genetic information of the individuals, and thereby the proper solution to the task which is represented by individuals. After recombination, the algorithm has to evaluate the solutions that are represented by the individuals. The fitness function provides this evaluation: it tells us how high the quality of the solution contained in an individual is. If the required solution is found, the algorithm can be terminated. If not, the algorithm will select individuals for a new population and the whole process repeats. We can say that a new generation begins. The algorithm can be terminated if a certain number of generations is achieved. [1]

There are several types of recombination operators. The SGA most often exploits one-point crossover (*fig. 2*) and the mutation by means of bit negation. However, the type of crossover and mutation used depends on particular application. Note that operators can also be very complex (more than two parents, vectors mutation, etc.).

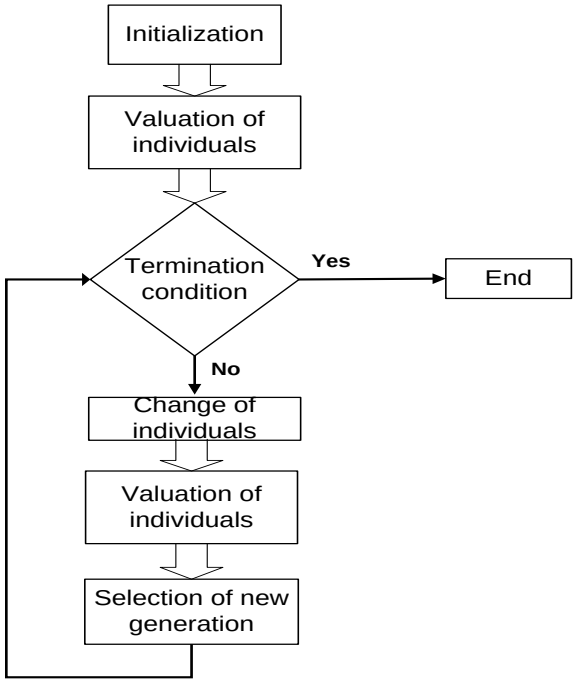

Fig. 1: Principle of SGA

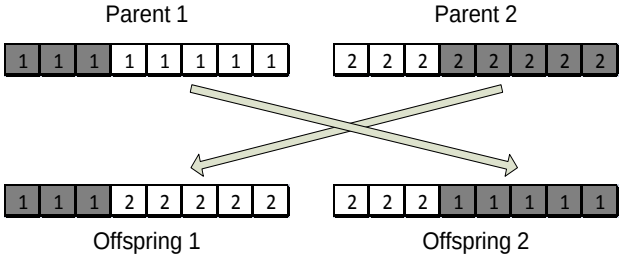

Fig. 2: One-point crossover

EVOLVABLE CIRCUITS VS. EVOLUTIONARY CIRCUIT DESIGN

The interconnection between evolutionary algorithms and reconfigurable structures can be used in two similar domains. However, "Evolvable Circuits (hardware)" and "Evolutionary Circuit Design" are not the same notions.

The evolutionary design deals just with the design of circuit by means of evolutionary techniques. The evolution is used only for development. This method of circuit design makes use of the mathematical model either of the reconfigurable structure or of the system with variable parameters. This model is used for the fitness function. A new and innovative solution to the circuits can be found by these techniques. The resulting circuit can be implemented as real hardware.

However, in case of evolvable hardware, both parts – the reconfigurable structure and the evolutionary algorithm – are really implemented. The evolution can run whenever. It means that the circuit can change its function dynamically in time. By this approach we can get a circuit which performs various functions. To change its function, we need to define required new function at first and then the evolution can be started. If the evolution finds a suitable configuration, the system will perform required function.

Unfortunately, there are several problems that complicate the use of these techniques. The developer has to implement the evolutionary algorithm and the valid configurable circuit (structures) effectively. The key element of these systems is the fitness function. The valuation of individuals can be very difficult – time-consuming. For that reason, the speed of evolution (searching for configuration) depends especially on the implementation of fitness function. It is very important to find a suitable solution in a reasonable time, in contrast to the evolutionary design where the quality of the final solution is the most important thing. [2]

ISSUES OF THE DYNAMIC FITNESS FUNCTION

As it has already been noted, the fitness function is a key element of the evolutionary algorithm. This function expresses the quality of a found solution to the optimization task. The fitness function can be classified into two categories: it can be either static or dynamic. The kind of the function depends on a particular application. For example, in case of the evolvable combinational logic circuit, we could say that (although not always) static fitness function is used. The combinational logic circuit can be described by a truth table. The aim of the optimization is therefore evident already at the beginning of the evolution. For that reason, the fitness function is invariant throughout the evolution. However, if an adaptive behaviour is required, the fitness function must be time-variable because the application has to react dynamically on the environment variable. For example, in case of adaptive filters, the function has to react on the actual input data of the filter. This fact affects the implementation of the evolvable component. In variable environment the evolution runs de facto continually; in contrast to static environment, where the evolution can be terminated if a sufficient solution is found. [2]

It is necessary to modify the algorithm and its fitness function so that the algorithm can work with variable environment. The dynamic fitness function has to be transformed into a static (quasi-static) function; then the algorithm works with the static fitness function and the adaptive character is conserved at the same time. A suitable transformation into the static function ensures that all individuals within one generation have the same evaluative criteria.

EVOLVABLE COMBINATIONAL LOGIC CIRCUIT

This part of the paper describes the practical use of the standard genetic algorithm and reconfigurable structures. The goal is to design a combinational logic circuit that can dynamically change its function. It means a circuit with a variable truth table. This project outlines the development of these systems on a simple example (4 inputs and 2 outputs). It should also point to the possibilities of the implementation of these systems by FPGA devices.

There are several approaches that can be used. The evolution (evolutionary algorithm) can be implemented by a personal computer or by some microcontroller. Thus, the algorithm and reconfigurable structures do not need to be implemented by one device. The processor can perform evolution and another device (PLD, FPGA, transistors array) implements the

structures needed for reconfiguration. The next viewpoint has to do with the calculation of fitness functions. There are two fundamental principles – extrinsic and intrinsic evolution. The first one means that fitness function and the whole process proceed in circuit simulator (in a computer or microcontroller). The real implementation of the circuit is created only for the best circuit solution at the end of the evolution. This type of evolution is often slower and can also have disadvantages (insufficiently exact circuit model, etc.). In case of intrinsic evolution, the fitness function for all individuals is computed in real hardware. This fact makes the evolution faster (although not always) in contrast to the extrinsic evolution and pertinent differences between a circuit model and a real implementation are eliminated. In this project, the evolution and reconfigurable structure were implemented by one FPGA device. It enables a fully-autonomous evolvable system.

The selection of a suitable reconfigurable structure is a very important stage of the development. We can choose from several variants. The key parameter for the design of a profitable structure is to define the "function level". This parameter tells us what elementary circuit components are used. The system can work on a very low level – transistors. The gate level represents a higher function level. Furthermore, the system can implement higher functions (an adder, a multiplier, max, min, etc.). The lower level makes it possible to find a more innovative and unconventional solution. However, the low level means a longer configuration bitstream (chromosome) for the circuit. Unfortunately, longer chromosome can cause increasing time needed for successful evolution. The problem is called "scalability". This feature can be depressed by using a higher function level. [2]

The structures based on Cartesian Genetic Programming fit the evolutionary combinational circuits. [2] These structures consist of array of function cells. The function cells can implement various logic functions (AND, OR, XOR, etc.). These cells are interconnected by group of multiplexers. Feedbacks are not allowed. The structures have a fixed number of inputs and outputs. The configuration bitstream (fixed length) determines the function of the cells and interconnection via multiplexers. For this project, a simple structure from [3] was assumed. The diagram of this structure and its bitstream are shown in the *fig. 2*.

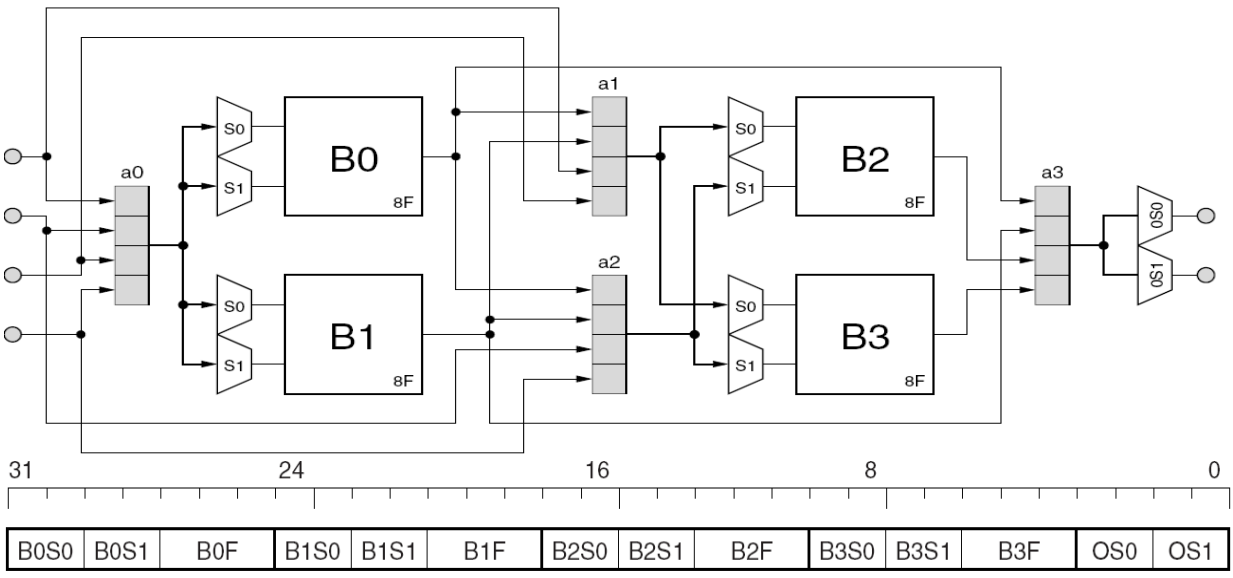

Fig. 2: The reconfigurable structure and its configuration bistream [3]

It disposes of 4 inputs and 2 outputs. The circuit is controlled by 32-bits configuration bitstream. Overall, there are 2^{32} different configurations. However, it can implement only

178,764 unique logic functions. It is obvious that certain logic function can be implemented by various methods. Every function cell has 2 inputs and 1 output and implements 8 logic functions: NOR, x AND not y, not x AND y, AND, OR, not x OR y, x OR not y, NAND. The groups of the multiplexers control the connection between the cells, between the inputs and the cells, and between the cells and the outputs. [3]

Two possibilities of the implementation of the standard genetic algorithm by FPGA device were taken into account. It was possible to take use of a soft-core processor and algorithm created in C language. This way of development would be very fast. However, the computational power of soft-core processors is not too high. These processors come in useful for the control of peripheries. As long as high speed is required, it is necessary to create an algorithm by user logic. However, the soft-core processor can be profitably employed for the control of this created logic (algorithm). This way of development was chosen. Both the reconfigurable structure (the circuit) and the standard genetic algorithm were implemented in one FPGA circuit. The FPGA Altera Cyclone II circuit was used.

The whole evolvable combinational logic circuit is composed of the Avalon periphery for a soft-core NIOS II processor. This approach is very profitable for debugging and testing. We can observe and control the process of the algorithm very easily by the NIOS II processor and its J-TAG console. However, the algorithm itself runs very quickly in the created logic.

The whole system is divided into several parts (*fig. 3*). The individuals and their fitness are stored in the embedded memory of the FPGA circuit. This memory is very fast. An SGA with 64 individuals was used. Every individual needs 32 bits for its representation and 8 bits for its fitness. However, the whole algorithm needs 4 times more memory, because we have to store also the former generation, the offspring and the mutants. On the whole, we need about 1280 bytes.

Several modules perform main operations of the algorithm. They are called: crossover, mutation, selection, fitness and elitism. These modules are controlled by means of control register. This modular architecture is advantageous, because we can simply change particular part of the algorithm. The modules are connected with the memory by a multiplexer. This multiplexer is also controlled by control register. Therefore, this register controls the whole process of the evolution. We can read and change its value by the NIOS II processor via the Avalon bus. The processor can also read the value of the leader (the best individual) and its fitness. The Avalon slave port is used for communication between the evolvable circuit and the NIOS II processor. The selected way of implementation enables a very easy use of this periphery by other systems.

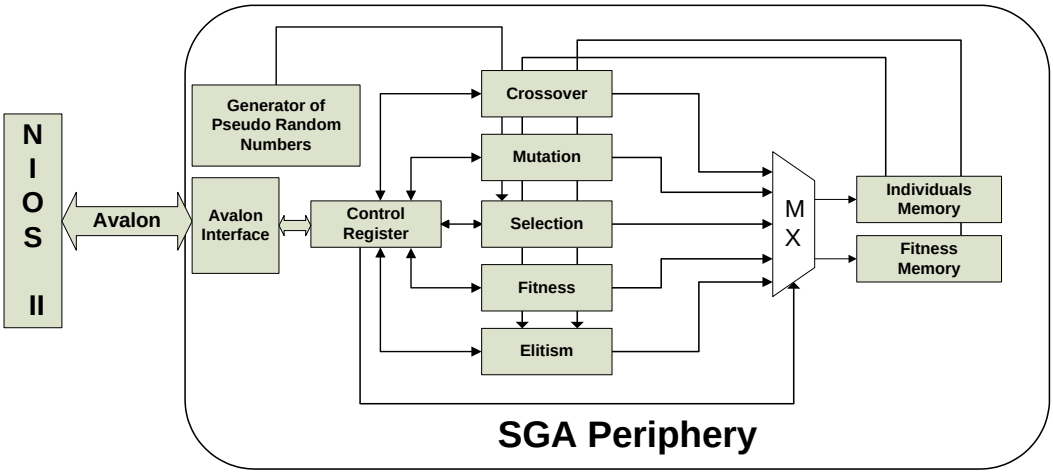

Fig.3: SGA periphery

The genetic algorithm has many parameters and modifications. A decision was made to use a standard modification of the SGA. The individuals of the algorithm are represented by means of a binary 32-bits stream. It corresponds to the configuration stream of a reconfigurable structure. One-point crossover is used. It means that two parents (the individuals) enter into crossover and two offspring are the output. However, the question is whether this crossover is efficient for this type of application. The selection of parents depends on a probability of the crossover (the P_c parameter). For each of the individuals a random number (8 bits) is generated and it is compared to the P_c parameter. If this random number is less than P_c, the individual participates in crossover. Two types of mutation were used. The first type depends on the probability of the mutation (the Pm parameter). This way of mutation ("multi-bits mutation") can change several bits in the individual. The second method always changes only one bit of the individual (single-bit mutation). Moreover, this type of mutation does not depend on the Pm parameter – a generated random number (5 bits) determines directly the position of the bit which is mutated (negated). All individuals always undergo mutation. The key element of every evolutional algorithm is a fitness (valuation) function. The individuals were tested with all input combinations. The fitness expresses the number of lines in the truth table, where the output matches required output. The reconfigurable structure with 4 inputs and 2 outputs was used. For that reason, the maximum of fitness is 32 (16 x 2). The 2-individuals-tournament type of selection is used in the SGA. This type of selection should ensure a sufficient diversity of the population. The individuals for a new population are selected from the offspring, the mutants and from the individuals of the former generation. The tournament selects 64 individuals for the next generation from overall 192 individuals (64 mutants, 64 offspring and 64 individuals of former generation). The selection is complemented by elitism. It means that the best individual (the leader) always gets into the next generation.

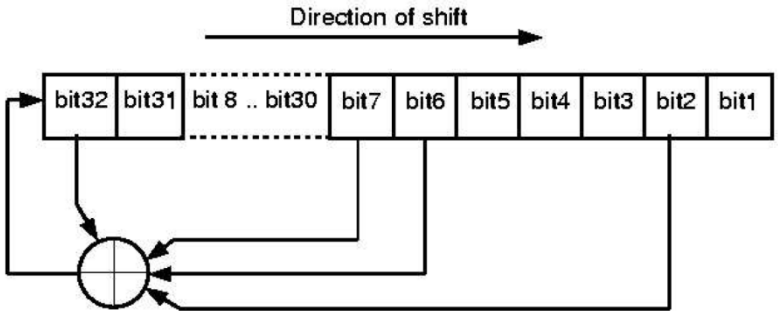

Fig. 4: 32-bits linear feedback shift register [4]

The system also includes a generator of pseudo random numbers. The generator is created by a 32-bits linear feedback shift register (*fig. 4*) with the polynomial $x^{32} + x^7 + x^6 + x^2 + x^0$. This generator is designed in such way that a random number is generated in one cycle [4].

The whole system was successfully implemented by FPGA device. The system needs approximately 1300 LEs (logic elements), 630 registers and over 1 kB of memory. The verification of the operation of the system is not easy. Because each system based on the evolutionary algorithm contains a certain random component part. For that reason, the system testing has to be repeated several times. In such a case, the results have statistical nature and can be considered as relevant.

For the system power measuring, several test truth tables (logic functions) were chosen. Each evolution of every truth table (circuit) was 1000 times repeated in order to

obtain relevant indications. The evolutions ran with various parameters (P_c, P_m, type of mutation) and an optimal set of parameters was analyzed.

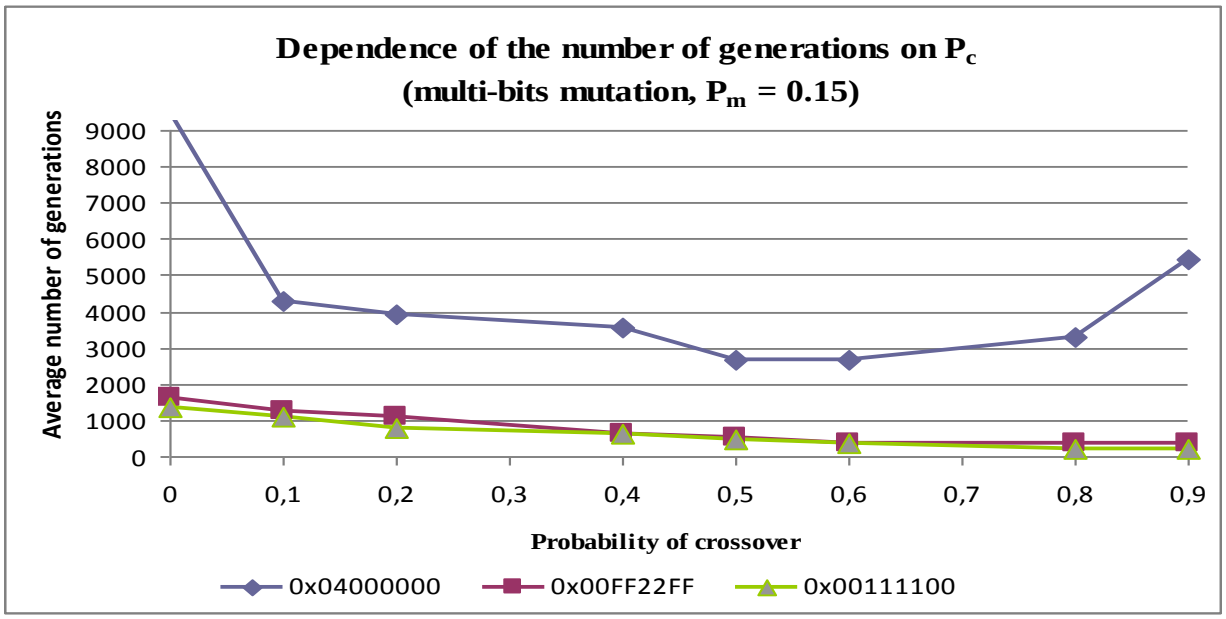

Chart 1: Dependence of evolution speed on P_c parameter

Chart 1 describes the dependence of the average number of the generations that are needed for successful evolution (fitness = 32) on the probability of crossover. The P_m parameter (probability of mutation) was set for fixed value 0.15 (15%); it is a recommended value of mutation. The curves represent particular logic circuits defined by truth tables. In the chart key, the truth is defined in serial format. For example, 0x04000000: the first output of circuit is always in logic zero, the second one is in zero except input "1010". The curves show that with low value of P_c parameter the number of needed generations rises. In this situation, the evolution has to rely on the mutation; it is the only one to bear change of genetic information in the population. The testing pointed that the optimum value of P_c is about 0.6 – 0.7. A very interesting piece of knowledge is the fact that certain regularity in a truth table conduces to faster evolution.

The next chart shows opposite situation, the P_c parameter is set for the value of 0.7 and the probability of mutation is variable. It is obvious that mutation contributes to a significantly better course of the evolution. The optimum value is approximately 0.15. This fact is clearly readable from the chart. High value of the P_m causes very aggressive interventions to individuals, because such mutation can randomly change a significant part of an individual – quality genetic information is lost. For that reason, only low values of mutation are recommended. However, very low values can markedly slow down the convergence of the evolution.

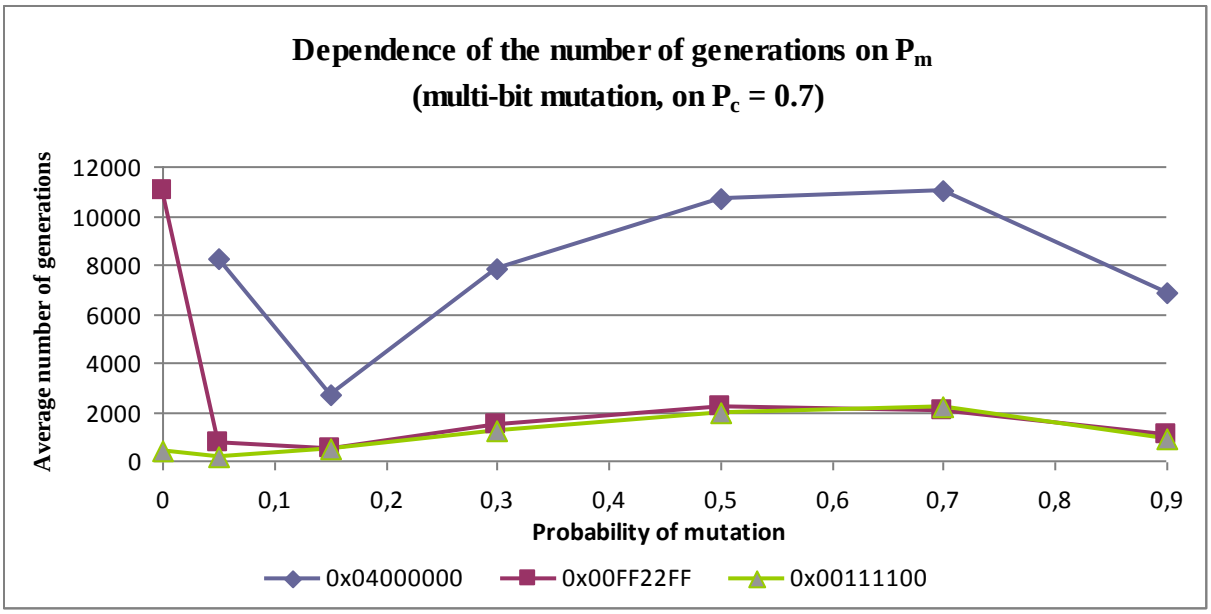

Chart 2: Dependence of evolution speed on P_m parameter

As long as it was noted that too aggressive mutation causes damage of good genetic material, this problem can be solved by a particular method of mutation which changes a limited number of bits in individual. The *chart 3* shows the situation with "single-bit" mutation. This mutation always changes only one bit in the individual. The impact of this mutation on genetic information is very soft.

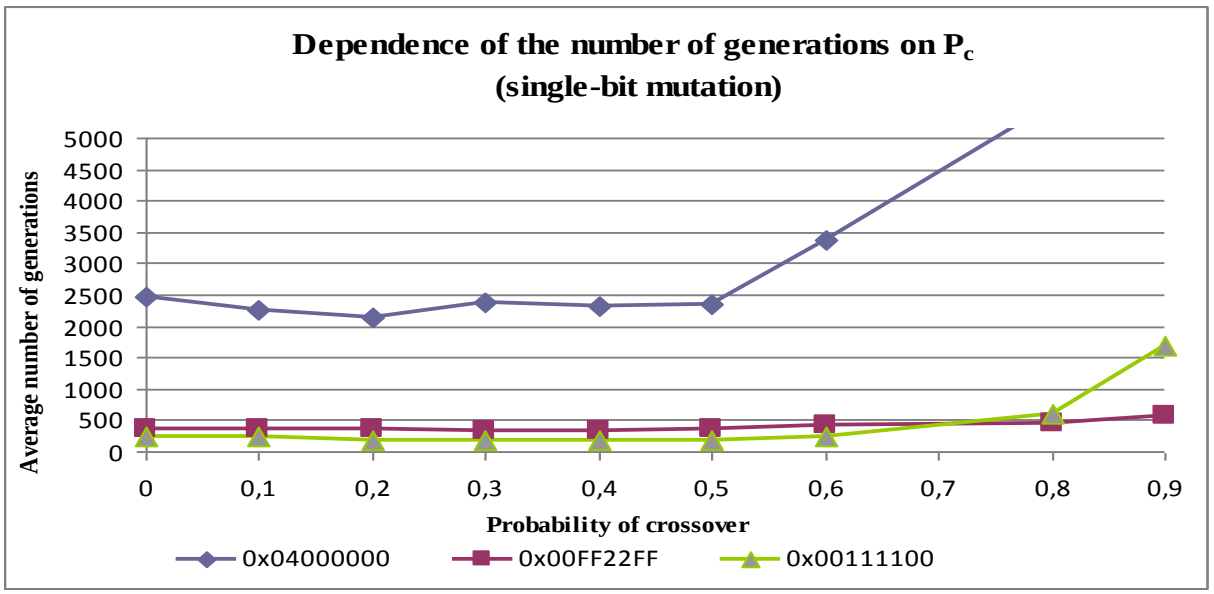

Chart 3: Dependence of evolution speed on P_c parameter (single-bit mutation)

This chart shows that the crossover does not contribute to quicker evolution in any way. The chart acknowledges the fact that the mutation is the only effective recombination operator in Cartesian Genetic Programming domain.[2] We can say that an evolution algorithm with small population and a suitable method mutation is an ideal modification of algorithm for these applications.

All tests ran on the system with 90 MHz system clock. The computing power of the implemented Standard Genetic Algorithm is roughly 14,000 gens/s (multi-bits mutation) and

29,000 gens/s (single-bit mutation). On the basis of these specifications, we can determine the time (see *table 1*) needed for the evolution of the circuit. The single-bit mutation is less time-consuming; hence the evolution based on this type of the mutation is faster.

Table 1: Samples of circuits and times needed for the evolution (with ideal SGA parameters)

Sample	P_c	P_m	P_c (single-bit)	The number of the generations	Time[ms]
0x04000000	0.6	0.15	-	2652.03	164.48
0x04000000	-	-	0.2	2388.18	82.18
0x00111100	0.5	0.05	-	146.34	8.58
0x00111100	-	-	0.4	169.74	6.17
0x00FF22FF	0.6	0.15	-	340.87	20.55
0x00FF22FF	-	-	0.3	334.74	11.59

The time of evolution achieved is obviously not ideal. Note that only small circuits were evaluated. For increase in the evolution speed, the fitness function has to be better implemented. We can use some kind of parallel processing. If the algorithm can evaluate more individuals at the same time, the evolution will be much faster. Note that for some truth tables the evolution did not find a solution. It is a major disadvantage of this method of design. We have no certainty that the solution will be found. The algorithm can stay at local maximum (for example: fitness = 31) and the circuit will not work exactly according to the required truth table. At the end of this chapter, the instances of evaluated circuit are presented (*fig. 5*).

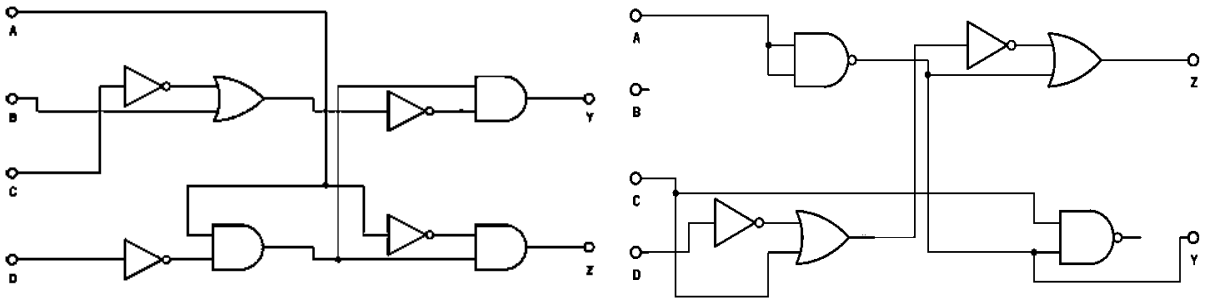

Fig. 5: Evaluated circuits
(truth tables: 0x0400000 - on the left and 0x00FF22FF - on the right)

EVOLUTIONARY FIR FILTER

The second practical demonstration of an evolutionary system which is presented in this paper is the "Evolutionary FIR filter".

The main goal of this project is the design of an evolutionary FIR filter. The parameters (the impulse response) of this filter are obtained by the evolution. It means that no operator intervention is needed for correct function of the filter. The standard FIR filter has one data input and one data output. In addition, the evolutionary filter has a special data input – the input for ideal output samples. The evolution will have to find a suitable impulse response that ensures a correspondence between the output signal and ideal output signal. That is why the filter has an adaptive character.

Fig. 6: Evolutionary FIR filter

The FIR (finite impulse response) filter is one of the basic types of digital filters. Its function is based on the convolution. The FIR filters excel in simple structure and stability. They consist of a shift register, multipliers and adders. The shift register accumulates input data samples that are multiplied by parameters (the impulse response) of the filter. Afterwards, these multiples are summarized and the consequent result creates the filter output. [5] Filter parameters determine its kind and its amplitude characteristic. If the impulse response is changed by the evolution, the features of the filter are changed too. The FIR filter is always stable. For that reason, this type of filter is suitable to determine the impulse response by means of evolutionary techniques.

In this project, the symmetric FIR filter with 29 taps is used. The taps of odd numbers are preferable if we want to generate various kinds of filters. Each parameter is represented by 8 bits in the two's complement. In addition, the FIR filter structures can be implemented very simply by an FPGA circuit. The FIR filter also has to be able to perform a quick reconfiguration if we want to use it for cooperation with the evolutionary algorithm.

With regard to the adaptive character of the filter, the design of this filter is a different task from the case of the evolvable combinational logic circuit. The previous system worked with static fitness function.

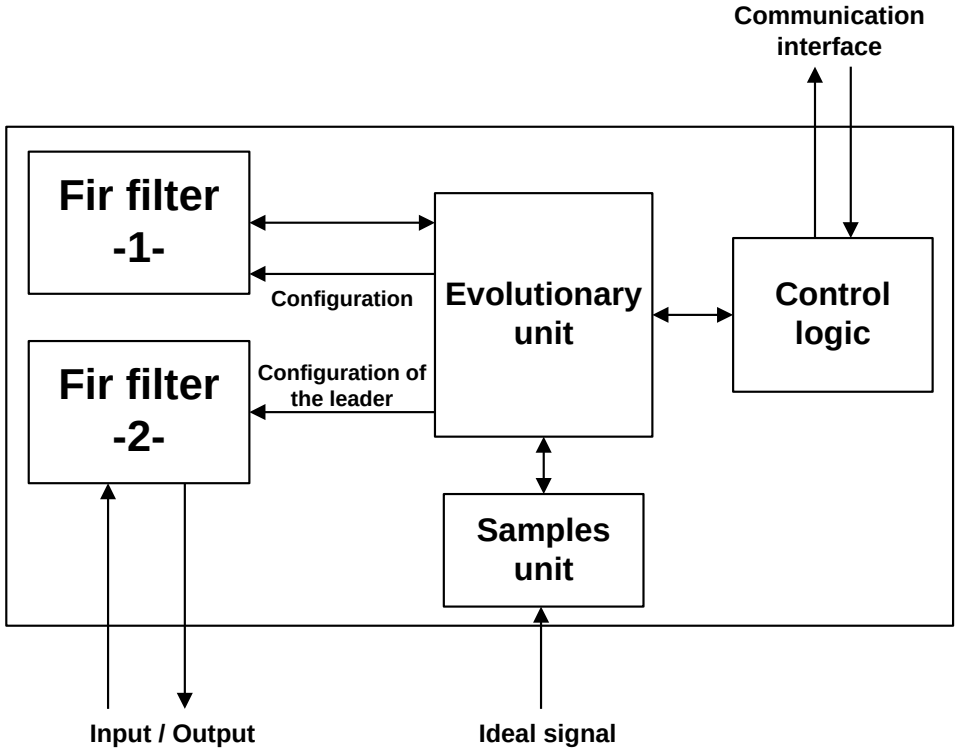

Fig. 7: The diagram of the evolvable system

This fact requires several changes in the evolvable system. The evolvable system working in variable environment has to perform two basic tasks. The first task is the evolution of the system. It is necessary to seek for a suitable set of filter parameters. However, the filter has to process input signal at the same time. It is necessary to reflect these two tasks in the real system (see *fig. 7*). For that reason, an adaptive evolvable system has to be composed of two same reconfigurable structures at the minimum - in this case, composed of two FIR filters. The first is used for fitness function calculation, the second one serves to process the input signal. The latter is configured by the best individual (the leader). Two filters in the system allow the correct function of the evolutionary algorithm and processing of the input signal at the same time.

Nevertheless, the needed system changes are not finished with this. It is also necessary to change the evolutionary algorithm so that it fulfills the requirements of the application. The algorithm uses 16 individuals. Each individual consists of 15 parameters of the FIR filter. It means: 15 x 8 bits = 120 bits, then 4 bits are used for the control of the filter output (precision) and 4 bits for the reserve. On the whole, the individual consists of 128 bits (16 bytes). If an offspring (max. 16), mutants (max. 16) and a former generation (16) are taken into account, a memory for 64 (16 x 4) individuals (1024 bytes) is needed. There are only 15 parameters in individuals. However, as the FIR filter is symmetric, it can be created with 30 parameters (only 29 are used in this project. The parameters are stored in two's complement. The initialization of a population is made by filling the individuals by random values.

The algorithm uses a primitive one-point crossover. The crossover depends on the probability of the crossover – P_c. Two parents participate in the crossover and two children result from it. It is obvious that this type of crossover is probably not optimal for this application. Other methods of crossover will be mentioned in the following text.

The situation with the mutation is very difficult, because the mutation can be implemented in many ways. In our project, the following method of mutation is used: At first, an individual is divided into single bytes (parameters of the FIR filter). Afterwards, the generator of random numbers generates 11-bits random number. First 8 bits determine whether a certain byte will undergo the mutation (according to the probability of the mutation - P_m). Remaining 3 bits determine the position of the bit for the mutation. The mutation of the bit means its negation. All individuals of the population compulsorily undergo the mutation. It is very profitable if the probability of mutation can change value within the run of the algorithm.

The selection of new generation is solved in the same way as in case of the previous evolvable logic circuit. However, the principle of elitism is disturbed on the ground of an adaptive character. In this application, the leader (the best solution/individual) does not need to be better than in previous generation. It is caused by the fact that the environment, when the evolution is running, is variable. It means that the individual that provides a very good solution in the current generation can present an absolutely unsuitable solution in the next generation. This fact requires the calculation of fitness function for all individuals (also former population) in each generation. It ensures us a valid valuation of the individuals. However, it is clear that this change influences the evolution speed negatively. This change also does not solve the basic problem with dynamic fitness function.

The work with dynamic fitness function is solved by means of a special sample memory – the samples unit. The principle of this unit is based on shift register. The samples unit accumulates samples of the input and ideal output signal. If the fitness function is started, last 200 samples (200 input samples and 200 ideal output samples) from the samples unit are

stored in work registers in a fitness module. The valuation of the whole population within one generation proceeds with the same samples of signals; in this way, equal conditions for all members of the population are ensured. The factual fitness function of the evolutionary filter is based on the method of least squares. The filter in the fitness module processes the input signal (from the samples unit). Afterwards, the differences between the filter output and ideal output signal are calculated. The sum of these differences creates the resultant values of fitness. The best individual has the lowest value of fitness.

The rest of the system and its implementation by FPGA device are solved in a way similar to the case of logic circuit. Note that this system obviously needs significantly more hardware recourses. Remember that the previous evolvable logic used individuals with the length of 32 bits. This filter system uses 128 bits. The whole system was implemented by FPGA Cyclone II device. This device contains embedded multipliers which are very useful for the use in DSP domain. All modules of the system are optimized so that the access to the memory is exploited effectively. The pipeline structures are used in the modules. For example: the process of the crossover (the creation of maximum number of the offspring - 16) needs only 21 cycles of system clock, the process of the mutation (the mutation of 16 individuals) needs only 275 cycles of system clock. However, the calculation of the fitness function is very time-consuming. This function needs 11,182 cycles (the valuation of 48 individuals – former population, mutants and offspring). Naturally, it is possible to reduce this time-consumption: we can use fewer samples for fitness function or exploit some parallel structures. One generational cycle needs c. 12,000 system cycles. If system clock is 100 MHz, the performance of the evolution is c. 8,300 generations per second.

For testing and verification of the system function the simulation of the perturbing influence on useful signal was used. The interference signal was superimposed to the useful signal. The useful signal is used as ideal output signal. The task of the evolutionary FIR filter is to eliminate the interference signal. It is surprising that the filter provides good results after only few generational cycles. It is important to note that this implementation is the first prototype of the system, and after a future development better results might be expected.

In the next figures we can see an example of the performance of the filter. One signal with frequency 1 kHz and another with frequency 10 kHz were regarded as the useful signal (*fig. 8*). Other signals with frequency 5 kHz and 15 kHz were superimposed (*fig. 9*) on this signal. The useful signal was termed as an ideal filter output. The sample frequency 48 kHz was used for the testing. The good working of the filter is detectable already in the 6^{th} generation, and in the next generations the results continue improving. In the $1{,}000^{th}$ generation (*fig. 10)* we can see that the attenuation of the "interference signals" is c. 42 dB (5 kHz) and 43 dB (15 kHz). When interpreting the results, let's take into account the possibilities of the FIR filter with 29 taps.

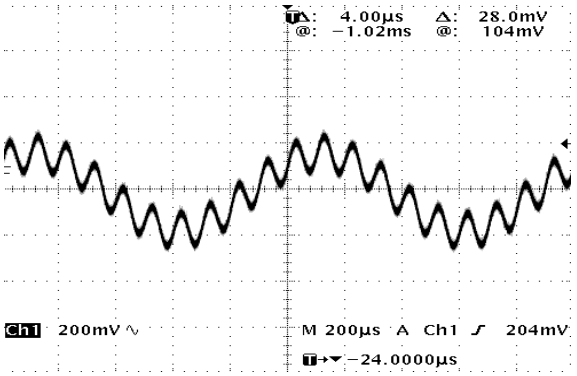

Fig. 8: Useful signal

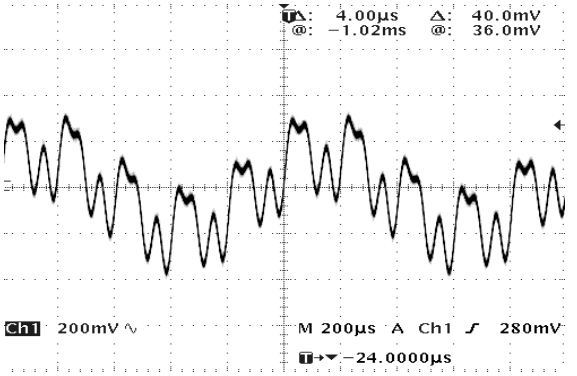

Fig. 9: Filter input signal

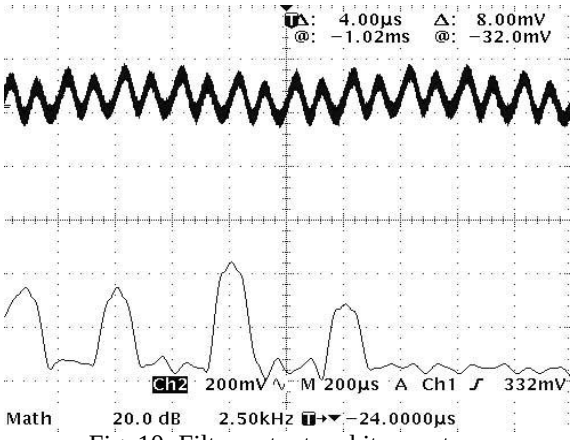

Fig. 10: Filter output and its spectrum (1^{st} gen., fitness = 11,028)

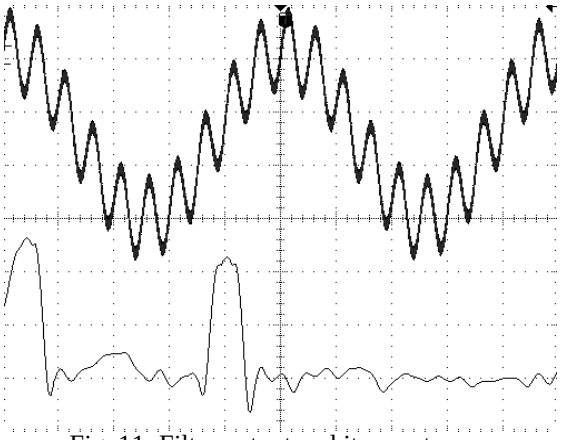

Fig. 11: Filter output and its spectrum ($1{,}000^{th}$ gen., fitness = 1,005; 20 dB/div, 100 mV/div)

The same test application was used to analyze the influence of the parameters of the evolution on its behaviour. By means of quality analysis, we can determine optimal values of parameters of the evolution that could get better time needed for successful evolution. The testing can also point to the pertinence of selected recombination operators. In case of the evolvable logic circuit, it was shown that one-point crossover is not suitable for the evolution in Cartesian Genetic Programming domain. The next chart shows the dependence of speed of the evolution on the P_c parameter. The mutation was fixed set to the value $P_m = 0.4$ (40%). The evolution process was terminated when fitness function achieved the value of 2,000 or when the number of generations achieved 30,000. Each run was repeated 1000 times.

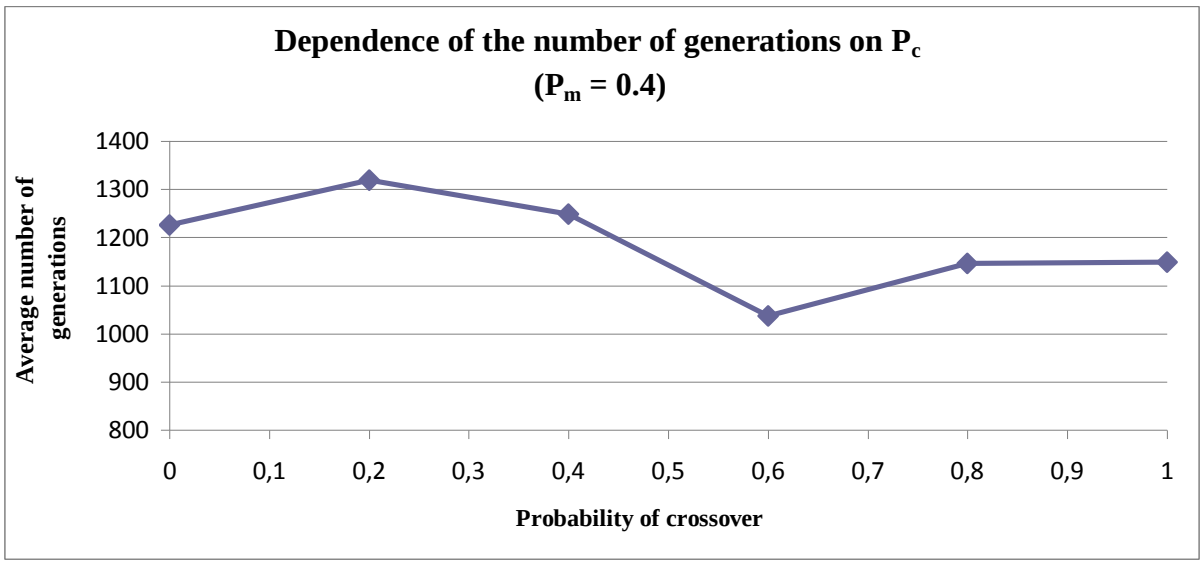

Chart 4: Dependence of the number of the generations on P_c

We can observe in the chart that the crossover does not yield an essential improvement of evolution speed. The process of search for solution is quicker (less generations) when the P_c value is about 0.6, but we have to take into account that higher value of the P_c increases the number of individuals. It means that fitness function for more individuals must be calculated. For that reason, this type of crossover is not benefit.

In the next chart, the dependence on P_m is analyzed. We can see that a change of the P_m parameter causes big differences in the number of generations needed. The optimum is

between 0.3 and 0.4. These values are higher than generally recommended values (c. 0.15). It might be caused by the fact that the mutation of filter parameter does not always have the same meaning. The filter parameter is represented by 8-bits in two's complement. And it is clear that the mutation of the least significant bit causes different change than mutations of other bits. By this, a higher value of mutation can be explained.

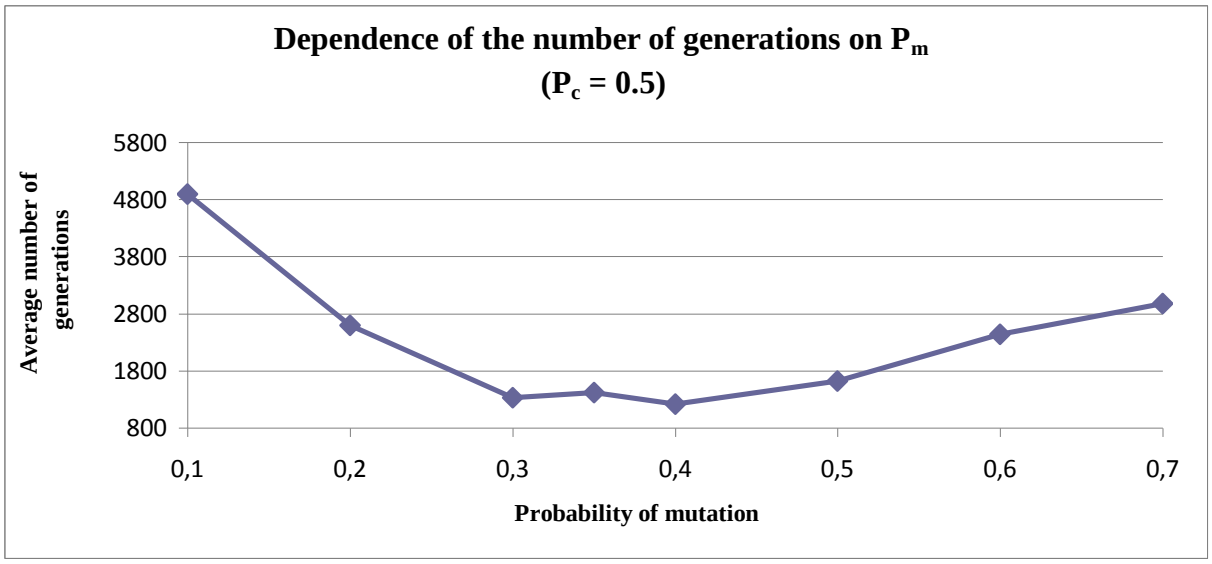

Chart 5: Dependence of the number of the generations on P_m

As one-point crossover does not exhibit good results, another method was sought. The principle of averaging of filter parameters was tested. It is also based on biology, strictly speaking on corporate culture. Animals in herd seek for food and if a member of herd locates good source of food, it informs others. Then, other members go towards this source. The averaging of individuals with the leader simulates this behaviour exactly. [6] This principle was implemented and it provides markedly better results than the previous version of the crossover. It is shown in the *chart 6*.

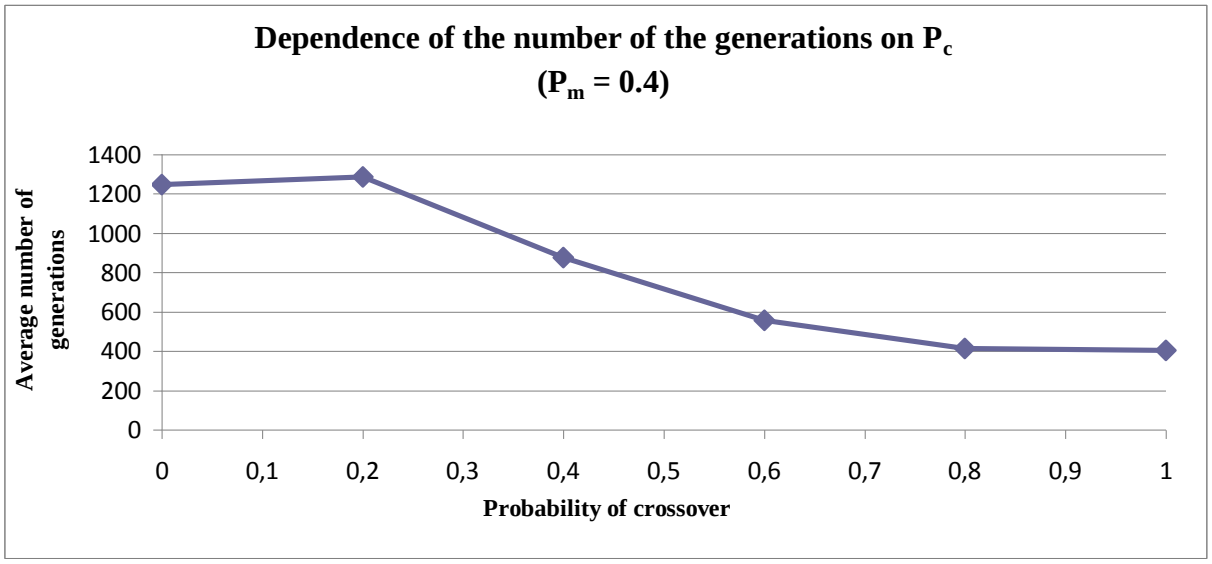

Chart 6: Dependence of the number of generations on P_c

The results of this crossover look like very efficient. If the value of P_c rises, the number of needed generations falls. However, it is necessary to note that averaging of

individuals with the leader can depresses the population diversity. This fact could negatively affect especially dynamic characteristics of the evolutionary FIR filter – the adaptability competence could be restricted. Hence, further research in this field is needed.

CONCLUSION

The evolutionary algorithms and reconfigurable structures implemented show that the design of the evolvable system can be solved by means of FPGA devices very effectively. Naturally, the system with long individual (chromosome) can require sizable hardware recourses (logic elements, registers, embedded memory). The principle of the evolutionary algorithm is altogether simple. However, we have to search for a suitable version of algorithm for each particular application. On the basis of the research executed and practical experience, it is possible to define several primary recommendations:

- Use of simple algorithms
- Small populations
- Crossover is not efficient
- Pay attention to fitness function optimizing

REFERENCES

[1] Lažanský, Mařík, Štěpánková: *Umělá inteligence 3.* Academie, 2003. ISBN 80-200-0472-6.

[2] Sekanina L.: *Evolvable components*. Springer, Natural Computing series, 2004. ISBN 3540403779.

[3] Sekanina Lukas, Mikusek Petr: *Analysis of Reconfigurable Logic Blocks for Evolvable Digital Architectures,* In: Lecture Notes in Computer Science, 2008, no. 4974, DE, p. 144-153, ISSN 0302-9743.

[4] Martin, P.: *A Hardware Implementation of a Genetic Programming System Using FPGAs and Handel-C,* In: Genetic Programming and Evolvable Machines, 2008, p. 317 – 343, ISSN 1389-2576.

[5] Alan V. Oppenheim, Ronald W. Schafer, John R. Buck: *Discrete-time Signal Processing*. Prentice Hall, 1999. ISBN 0137549202, 9780137549207

[6] Zelinka I.: *Umělá inteligence v problémech globální optimalizace*. BEN, 2002. ISBN 80-7300-069-5