FACULTY OF ELECTRICAL
ENGINEERING
UNIVERSITY
OF WEST BOHEMIA

DOCTORAL THESIS

to obtain the academic title Ph.D.
with specialization in Electronics

Petr Burian

Implementation of Selected Bio-Inspired Techniques by Programmable Logic Devices

Director: *prof. Ing. Jiří Pinker, CSc.*
Date of doctoral state examination: *22nd March 2010*
Date of thesis submission: *11th October 2013*

Implementation of Selected Bio-Inspired Techniques by Programmable Logic Devices

ABSTRACT

This thesis deals with the analyses and the implementation of selected unconventional bio-inspired techniques by programmable logic devices. The main objective of the thesis is to design and implement three selected bio-inspired techniques.

In the first part, the focus is given to Cartesian Genetic Programming (CGP). The emphasis is put on the utilization in the domain of the evolvable hardware and the evolutionary design of digital circuits. The thesis presents modifications of CGP which cause that the wasted fitness calculations are omitted. After the analyses, the implementation of CGP with introduced modifications by an FPGA device is discussed. The author of the thesis introduces a special component detecting active genes in genotypes/chromosomes.

The second part deals with the design of the FIR filter whose parameters are obtained using a Standard Genetic Algorithm. The author explores the usage problems of evolutionary algorithms in adaptive systems domain, then discusses the adaptive FIR filter implemented by an FPGA device, and continues by the discussion of advantages and disadvantages of such implementation. The thesis also compares the use of different crossover operators.

The third part of the thesis describes the implementation of the fast image recognition based on n-tuple neural networks. It explores an n-tuple methodology using node 'grouping' and the possible advantages offered by this little-known technique. A novel approach to the organization of the neural networks data in the n-tuple memory is introduced. The author performs tests on a real-world recognition task – the recognition of road signs. Then the test results are presented, discussed and compared with conventional methods and other implementations.

The thesis yields these main contributions: 1) hardware implementation (by an FPGA device) of CGP reducing the number of performed fitness calculations; 2) the software tool for evolutionary design with the support of the generation of VHDL source codes; 3) a novel approach to the memory organization of the neural networks data in the n-tuple domain.

KEYWORDS

CGP; Cartesian Genetic Programming; FPGA; image recognition; n-tuple; neural networks; evolutionary design; evolvable hardware

Implementace vybraných biologií inspirovaných technik v programovatelných logických obvodech

ANOTACE

Tato disertační práce se zabývá vybranými nekonvenčními technikami, které nalézají svoji inspiraci v biologii. Hlavní cíl práce je analyzovat tyto techniky, a tři vybrané implementovat v obvodech programovatelné logiky.

První část práce se věnuje Kartézskému genetickému programování (KGP), důraz je kladen na jeho využití v oblasti evolučního návrhu a v oblasti vyvíjejících se obvodů. Autor představuje modifikaci tohoto algoritmu, která omezuje počet volání hodnotící (účelové) funkce. Práce se následně věnuje implementaci tohoto algoritmu v obvodu FPGA. Autor představuje speciální komponentu detekující aktivní geny v genotypech jedinců.

Druhá část práce demonstruje návrh FIR filtru, jehož parametry jsou získávány pomocí Standardního genetického algoritmu. Autor se věnuje problematice evolučních algoritmů v oblasti adaptivních systémů, následně diskutuje vlastní implementaci filtru. Práce také porovnává různé varianty rekombinačního operátoru.

Závěrečné část práce popisuje implementaci systému pro rozpoznávání obrazu, který je založen na n-tuple neuronových sítích. Autor seznamuje čtenáře s metodou seskupování a možnými výhodami, které nabízejí n-tuple neuronové sítě. Následně autor představuje nový přístup k organizaci dat neuronové sítě. Navržený a implementovaný systém pro rozpoznávání obrazu je testován na aplikaci rozpoznávání dopravních značek; výsledky testů jsou porovnávány s ostatními metodami a implementacemi.

Hlavní přínosy práce jsou následující: 1) hardwarová implementace (v obvodu FPGA) algoritmu KGP omezující počet volání hodnotící funkce; 2) softwarový nástroj pro evoluční návrh s automatickým generováním VHDL zdrojových kódů; 3) nový přístup k paměťové organizaci v oblasti n-tuple neuronových sítí.

KLÍČOVÁ SLOVA

KGP; Kartézské genetické programování; FPGA; rozpoznávání obrazu; n-tuple; neuronové sítě; evoluční návrh; vyvíjející se obvody

DECLARATION

I hereby declare that this doctoral thesis is completely my own work and that I used only the cited sources.

Pilsen, 11th October 2013 ……..……

ACKNOWLEDGMENTS

The completion of this thesis was possible thanks to the support of several people. I would like to express my sincere gratitude to all of them.

I am glad to express heartfelt appreciation to my thesis director prof. Ing. Jiří Pinker, CSc. In my mind, he is a person who gives meaning to the title of professor.

I am very thankful to doc. Ing. Martin Poupa, Ph.D. for his advice related to an FPGA design domain.

I am grateful to my dear colleague Ing. Radek Holota, Ph.D. for the cooperation that was inspiring and very fruitful.

I would like to thank Mgr. Kateřina Petrošová for her selfless help in the English grammar domain.

Special thanks go to my parents, who encouraged and helped me at every stage of my personal and academic life.

Finally, I would like to express gratitude to my dear girlfriend Kateřina. She always supports me in good and bad times.

GLOSSARY

List of Symbols

#	Number of
<CR>	Carriage Return
floor	Floor Rounding Function
$\mathbf{f_{mo}}$	Memory Organization Factor
$\mathbf{f_s}$	Frequency of system clock
$\mathbf{F_v}$	Factor of Valuation
$\mathbf{G_i}$	Gene Index
$\mathbf{ir_{ji}}$	Ideal response of circuit to training data
$\mathbf{M_l}$	Latency of reading from memory
$\mathbf{M_r}$	Memory requirements
$\mathbf{n_c}$ / $\mathbf{N_c}$	Number of column (in CGP)
$\mathbf{N_d}$	Number of discriminators
$\mathbf{N_g}$	Number of generations
$\mathbf{N_i}$	Number of individuals
$\mathbf{n_i}$	Number of program/primary inputs (in CGP)
$\mathbf{N_n}$	Number of n-tuples
$\mathbf{n_o}$ / $\mathbf{N_o}$	Number of program/primary outputs (in CGP)
$\mathbf{N_p}$	Number of selected pixels (for n-tuple processing)
$\mathbf{n_r}$ / $\mathbf{N_r}$	Number of rows (in CGP)
$\mathbf{P_c}$	Probability of Crossover
$\mathbf{P_m}$	Probability of Mutation
$\mathbf{P_{mag}}$	Probability of mutation of active genes (in CGP)
$\mathbf{P_{mig}}$	Probability of mutation of inactive genes (in CGP)
$\mathbf{rr_{ji}}$	Real response of circuit to training data
$\mathbf{T_e}$	Time of evolution (algorithm run)
$\mathbf{T_f}$	Time needed for the evaluation of an individual
$\mathbf{T_p}$	Time of production of population
VR	Valuation Reduction
$\mathbf{W_b}$	Width of memory data bus
$\mathbf{W_d}$	Width of data signal
$\mathbf{\Lambda}$	Length of a Chromosome/Genotype (in CGP)

λ	Number of Offspring
μ_g	Mutation Rate
μ_r	Mutation Rate (percentage)

GLOSSARY

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
CA	Coefficient of Acceleration
CCGP	Compact Cartesian Genetic Programming
CGP	Cartesian Genetic Programming
CMR	Coefficient Memory Requirements
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CSV	Comma-Separated Values
EA	Evolutionary Algorithm
ECGP	Embedded Cartesian Genetic Programming
ES	Evolutionary Strategy
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
fps	Frames Per Second
FR	False Rate
GA	Genetic Algorithm
HDL	Hardware Description Language
JTAG	Joint Test Action Group
l-back	levels-back parameter
LCD	Liquid-Crystal Display
LE	Logic Element
LEs	Logic Elements
LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit
LUT	Look-Up Table
MSB	Most Significant Bit
NFL	No Free Lunch
PLA	Programmable Logic Array
POE	Phylogeny - Ontogeny - Epigenesis
px	Pixel
RAM	Random Access Memory

RGB	Red - Green - Blue
SDRAM	Synchronous Dynamic Random Access Memory
SGA	Standard Genetic Algorithm
SLN	Single Layer Networks
SMCGP	Self-Modifying Cartesian Genetic Programming
SoPC	System on a Programmable Chip
SPICE	Simulation Program with Integrated Circuit Emphasis
SRAM	Static Random Access Memory
TNT	'Trixel' n-tuple
TTL	Transistor-Transistor Logic
UART	Universal Asynchronous Receiver and Transmitter
VBAI	National Instruments Vision Builder for Automated Inspections
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuits
VRC	Virtual Reconfigurable Circuit

GLOSSARY

List of Figures

GLOSSARY

List of Tables

CONTENTS

1 INTRODUCTION

The Industrial Revolution has caused the transition to new manufacturing processes. Since then, human work has been gradually replaced by machines. Inventions as steam engine, sewing machine or telegraph meant important changes in human thinking. Suddenly, people's power and qualifications were not needed for a broad range of activities.

At the present 'transistor' time, electronic systems occupy an important position in our lives. The use of cell phones, computers or modern measuring technology is our daily bread. It also caused the formation of the branch of artificial intelligence, among others. The progressive part of this branch is created by *evolutionary algorithms* (or evolutionary computational techniques in general). These algorithms are inspired by the biological evolution (Darwin's theory) and are successfully used in design domain. A lot of products (optical systems, mechanical systems, electronic circuits, antennas, etc.) were designed by means of these methods. Several of these products were patented and provide better behaviour than solutions developed by qualified and creative human workers. Thus these algorithms gradually replace human creative thinking in the same way as steam power replaced human work in the past.

In the domain of new electronic components, the emphasis is placed on their flexibility. This feature is represented by the possibility of reconfiguration. The process of development and research can be significantly accelerated if the component is able to change its functionality dynamically in time. Nowadays, such components exist in digital and analogue domain of electronics. However, the modern reconfigurable devices are represented mainly by an FPGA device. These devices may implement large and complex digital systems.

The intersection of these introduced approaches helped to establish a novel scientific field that is called *evolvable hardware*. The primary aim of this branch is the implementation of adaptive systems interacting with the environment; eventually, the systems that are designed automatically.

However, there are also other bio-inspired techniques that can be utilised in the electronics; the evolutionary algorithms represent only one of them. This thesis deals with the implementation of some selected unconventional bio-inspired techniques by FPGA devices.

1.1 Motivation

The primary motivation for writing this thesis was an encounter with the Thompson's publication [1]. Thompson implemented and performed a very interesting experiment which often serves as task for the understanding of evolvable hardware domain. In this case-study, Thompson intended to design a circuit that discriminates between 1 kHz and 10 kHz input square signals by the setting of an output signal. He chose the Xilinx XC6216 FPGA device as a target platform. This device supports a partial reconfiguration and the format of its configuration data was known. Note that no configuration can damage the device – unfortunately it does not apply to today's FPGA devices. An FPGA design is usually developed by means of Hardware Description Language (HDL). However, Thompson did not use it; the configuration of the FPGA device was created by the evolutionary algorithm. This algorithm was performed by a personal computer connected with the FPGA device and generated candidate solutions – candidate configurations. Each of these configurations was programmed into the FPGA configuration memory and the ability of discrimination was evaluated by an analogue integrator and a tone generator (a discriminator response to the input signal was tested). The output of the evaluation process was sent back to the PC. According to obtained responses, the evolutionary algorithm continues. The evolution process took roughly two weeks. However, after 3,500 generation cycles, the discriminator worked almost perfectly – only infrequent glitches were detectable, but they were eliminated after another 600 generations. It is necessary to note that no clock signal was used. The results of the experiment were impressive. However, the result of the evolution did not work perfectly in other FPGA devices. It was always needed to run the evolutionary algorithm on each FPGA device. Even when the element with no path to output was removed, the function failed. Further investigations showed that the correct functionality of the discriminator depended also on the power supply voltage and the ambient temperature. This behaviour causes that the solution generated by the evolution is not portable. The evolutionary algorithm produced a circuit where asynchronous feedbacks and the delays of paths are utilised for the correct function. For that reason it is obvious that the functionality is not caused only by the configuration data, but it depends also on the material and the manufacturing process. Furthermore, it is fascinating that the evolutionary algorithms make it possible to exploit physical properties of materials. This fact is called *evolution in materio* [2].

Another very interesting experiment was performed by Harding and Miller and published in [3]; the authors used an LCD display as reconfigurable structure. They fed defined places of the LCD by various values of voltage. Indeed, the places and voltage values were obtained by an evolutionary algorithm. The authors showed that it was possible to configure the LCD so that it worked as a filter, a circuit closer (a transistor), etc.

The author of this thesis found out that the project published in [4] is considered as the first *evolvable hardware*. This project is based on the use of GAL16Z8 device.

There are a lot of inspiring projects. For example, the prosthetic hand controller based on evolvable chip is published in [5]. It provides the digital PLA circuits with a 16-bit processor. The PLA circuits implement the controller according to the evolutionary algorithm performed by the processor. This approach yields improvement compared to other methods. The same hardware was used for the design of a robot navigation system [6]. The evolutionary algorithm was also used by NASA for the design of a satellite antenna within "Space Technology 5" mission [7]. The above-mentioned and other projects are described in [8]. The monographs [9][10][11] deal with the domain of evolvable hardware in detail.

However, as has already been noted, the evolutionary algorithms represent only one group of the bio-inspired hardware domain. The kind of biological inspiration is classified by the means of the so-called *POE* model. This model represents the three-dimensional space with the axes *P (Phylogeny)*, *O (Ontogeny)* and *E (Epigenesis)* [12]. The position in this space expresses the degree and kind of biological inspiration. The evolutionary algorithms belong to the *P* axe. The *Ontogeny* deals with the development of multicellular organisms. The *cellular automata* are a typical representative of this approach. The *Epigenesis* serves as inspiration for *immunological electronics* [13] and primary *neural electronics (neural networks)*. *Neural networks* represent the second main topic of this thesis; concretely, *n-tuple* neural networks were the subject of research.

Often, one of the main tasks of neural networks is to recognize the object within the neighbourhood and to opt for the best action based upon image understanding. This objective is usually very difficult and time-consuming. For this reason, specialised hardware systems are designed. The *n-tuple* methodology was chosen because it is very fast in hardware and comparable with other conventional methods in performance. A detailed comparison in different applications is published by Morciniec and Rohwer [14].

The original *n-tuple* methodology of Bledsoe and Browning from 1959 [15] has not been realised until later when it could be implemented by means of 'deterministic' logic nodes. Hence this may be considered to be one of the oldest pattern recognition techniques based on logic node neural networks. This method was later popularised by Aleksander [16][17] and realised in hardware [18][19][20]. In the early 1990's, several software simulations were created, which led to the design of an improved software system [21]. It included the image pre-processing stage [22] and the image recognition stage [23]. The system supported the recognition of binary, greyscale, and colour images. The novel derivative for this colour image recognition was published in [24][25].

The *n-tuple* methodology was originated (by Bledsoe and Browning) for the purpose of recognising printed characters and, subsequently, hand-written characters. Nevertheless, the use of the *n-tuple* method is not limited only to this purpose. In the past, systems utilising *n-tuple* nodes were used in many different applications, e.g. face recognition [26], texture recognition [27], control and automation [28][29], or medicine [30].

1.2 Goals of Thesis

The previous sections indicate the fundamental topics of the thesis that are evolutionary design of a digital circuit, evolvable hardware and *n-tuple* neural networks. Indeed, it is obvious that the subjects are very extensive. For that reason it is necessary to specify several main goals of the thesis.

- The author of the thesis will analyse *Cartesian Genetic Programming (CGP)* as the tool for evolutionary design and evolvable hardware. The focus will be also given to the possibilities of the reduction of fitness function calculations. Furthermore, the author will design components for a suitable implementation of this algorithm by an FPGA device. The design and implementation process should be discussed and documented in detail for easy re-use. The author also considers the implementation of a design tool to support the use of *CGP*.
- The second main goal is to demonstrate adaptive evolvable hardware implemented by an FPGA device.
- In the *n-tuple* neural networks domain, the objective is defined as the design of general architecture/system for image recognition based on this bio-inspired method. This system will be also implemented by an FPGA device.

1.3 Thesis Organization

The rest of the manuscript is organized as follows.

Chapter 2 describes the basic principles of evolutionary algorithms and *Standard Genetic Algorithm*, as the essential member of this algorithm group, is discussed.

Chapter 3 deals with the utilization of these algorithms in an evolvable hardware and an evolutionary design domain. The chapter defines the differences between these two terms and also sets the basic terminology.

Chapter 4 is focused on *Cartesian Genetic Programming*. The subchapter 4.1 explains the basic aspects of this algorithm needed for understanding of the other subchapters. The three above-mentioned theoretical sections are relatively brief, because the author of the thesis puts emphasis mainly on his contribution. Subchapter 4.3 analyses the possibilities to accelerate the process of evolutionary design. In this subchapter, the author presents modifications of *CGP* which cause that the wasted fitness calculations are omitted. These modifications were implemented and verified by the experiments and the results are presented in this subchapter. The author outlines the design of the compact version of *CGP* reducing the number of performed fitness calculations. Subchapter 4.4 discusses the implementation of *CGP* by an FPGA device in great detail. The subchapter is divided into several subsections. Each of them deals with individual component of the implementation – the mutation, the search algorithm, the configuration memory, the active genes detector and the fitness function calculation. The author of the thesis supposes that mainly the subchapter 4.4.1 is crucial; it describes the design and the implementation of a special component which detects active genes in the genotype/chromosome. The chapter also presents the benchmark results and FPGA resources utilization. Note that some of the subchapters are very comprehensive. The author realizes that this may be an object of criticism. However, the author believes that a detailed description contributes to easier understanding and reusing.

Chapter 5 deals with the design of an adaptive evolvable hardware. The chapter presents the evolvable *FIR* filter whose parameters are obtained using *Standard Genetic Algorithm*. Mainly, the conception of adaptive system is discussed. The chapter also presents the use of different crossover operators and their influence on the evolutionary process.

Chapter 6 describes the design and the implementation of the system for fast image recognition based on the *n-tuple* neural networks. At the beginning of the chapter, the basic

methodology and terminology in relation to *n-tuple* neural networks is presented. Further, the author describes the designed implementation of the system and discusses its individual parts. Mainly, a novel approach to the organization of the neural networks data in the *n-tuple* memory is introduced. The author performs tests on a real-world recognition task – the recognition of road signs. Then the test results are presented, discussed and compared with conventional methods and other implementations.

Finally, chapter 7 concludes the manuscript and summarizes the results and the contributions of the thesis.

2 EVOLUTIONARY ALGORITHMS

The evolutionary algorithms (EA) form a wide group of profitable tools for the solving optimization problems. They are based on the Darwin's theory of evolution and survival of the fittest, and make it possible to solve optimization tasks that can be defined as a search for the global/local maximum/minimum of a function.

The evolutionary algorithms belong to the category of stochastic search algorithms. The search space is a space that contains all possible considered solutions of the problem. In comparison to other search methods (hill-climbing algorithm, random search, etc.), the evolutionary algorithms use more candidate solutions. These candidate solutions are called *individuals*. A group of individuals forms a *population*. The EAs are iterative algorithms; it means that the searching is carried out in cycles. These cycles are called *generations*. The key element of each evolutionary algorithm is the so-called *fitness function*. This function produces a *fitness value* that expresses the quality of a found solution in the search space. In other words, the fitness value indicates how well the solution meets the problem requirements. In each generation, a new population or its part is created by the means of using genetically inspired operators (mutation and crossover are commonly used). The individuals forming the new generation are selected by a selection process. Usually, this selection is based on the current fitness of the individuals. A better fitness implies greater changes that an individual will live in the next generation. Likewise, they are more likely to produce the offspring by genetic operators. These operators produce new individuals with novel genetic information (offspring inherit part of the parental genetic information); it means that they represent novel solutions of the problem. The fitness value of the individuals controls the evolution towards better areas of the search space. The principle of the EAs is presented in the Figure 1. The evolutionary algorithm is terminated if a terminal conditional is satisfied. Generally, two conditions are defined; the algorithm can be terminated if the required solution (fitness value) is found, or if certain number of generations is achieved. However, other conditions may also be defined. [10][9][31][32]

The search functionality of the evolutionary algorithms can be divided into two approaches [31][9]:

- *Exploiting* – the algorithm looks around the best solution which has been found up to now.
- *Exploring* – the algorithm explores unknown places in the search space.

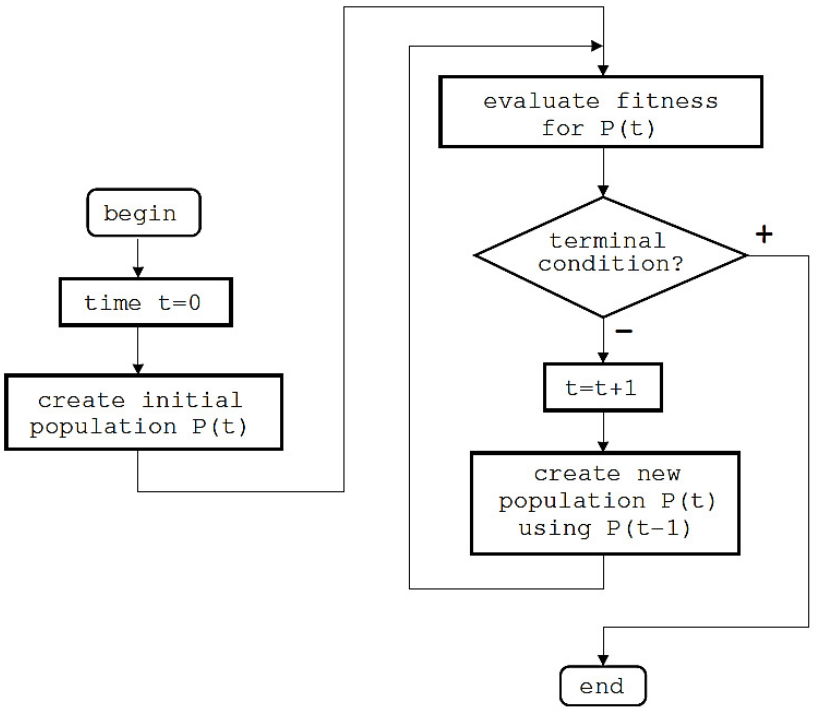


Figure 1. Control-flow diagram of an evolutionary algorithm [31]

These approaches should be in balance to achieve the best results. For example, if the exploiting is significantly dominant, the algorithm may have a tendency to converge too early. It may reach the peak of a local maximum and become stuck. Similar solutions have a low fitness while the actual solution is located in different part of the search space. On the other hand, if the algorithm performs only 'exploring', it passes to a random search algorithm – and the neighbourhood of promising solutions is ignored. [9]

The individual can encode various objects in the search space. For example, the representation of an individual can be based on a binary stream, floats, integers, etc. This forms a *genotype space*. The space of mapped actual solutions is called a *phenotype space*. An *encoding function* (see the Figure 2) defines the relationship between these spaces. The fitness function is applied to evaluate phenotypes. While the fitness function works with phenotypes, the genetic operators (e.g. crossover, mutation) are defined over genotypes. [31]

The group of evolutionary algorithms is wide. The fundamental evolutionary algorithms are the following: Genetic Algorithm, Evolutionary Strategy, Evolutionary Programming, and Genetic Programming. [31][10][9]

Unfortunately, a general-purpose universal optimization strategy does not exist. This fact is expressed by the so-called *no free lunch theorem* (NFL theorem). This theorem states: *"If*

a sufficiently inclusive class of problems is considered, no search method (never resampling points of the search space) can outperform an enumeration". [10][33]

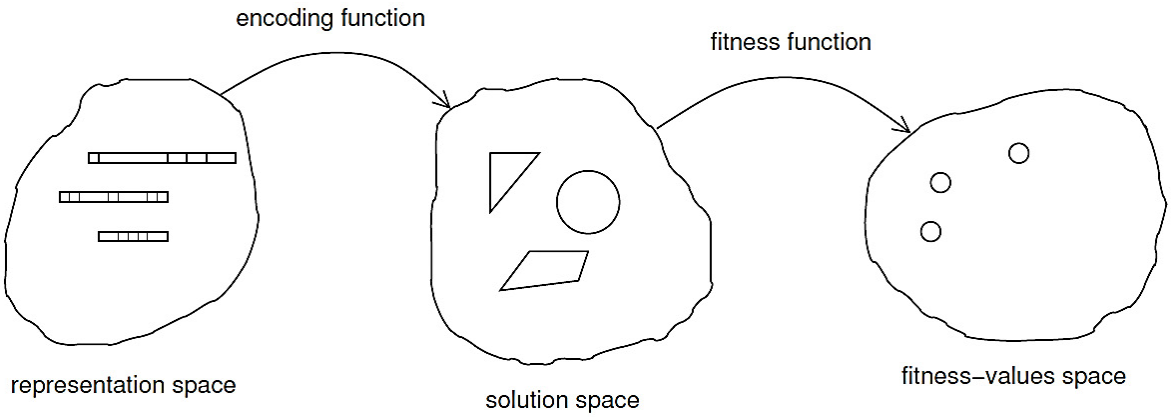


Figure 2. Demonstration of the genotype-phenotype-fitness mapping [31]

2.1 Standard Genetic Algorithm

Genetic algorithms (GA) represent the popular group of algorithms inspired by the evolution. They were created by John Holland in the 1970's [34]. The fundamental member of this group is the *Standard Genetic Algorithm* (*SGA*; this algorithm can also be called Simple GA), but there are other versions of GA, such as Messy GA, Compact GA, or Extended Compact GA.[35]

The following paragraphs describe the features of SGA. In publications [36][37][38][32], the genetic algorithms are described and discussed in detail.

Models

A Genetic Algorithm can be based on two models:

- *Generational*
  - The new population is formed only by the products of crossover and mutation.
  - All individuals of a population are of the same age.
- *Steady-state*
  - The new population is composed of new individuals (offspring) and members of the old population.
  - The individuals within one population are of different age.

Most GAs described in the literature is considered to be 'generational'. It means that the population consists entirely of the offspring formed by parents in the previous generation.

Indeed, some of the offspring can be identical to their parents. In a steady-state genetic algorithm, the offspring replaces only part of the parents.

Representation

An individual of the SGA is defined as a binary chromosome fixed length. This binary representation can map various objects. The binary chromosome represents the *genotype*. The decoded values from this chromosome form a *phenotype*. For example, the binary string "1111" represents the *genotype*. If the chromosomes encode integer (as a two's complement), the *phenotype* takes the value of -1. [10][38][32]

Usually, the initial values of the individuals/chromosomes are filled by random values. However, a knowledge base, if available, can be used to generate initial individuals.

Genetic Operators

There are two genetic operators defined in the SGA – the crossover and the mutation. The crossover creates two offspring from two parents. One-point crossover is the simplest genetic recombination operator. The principle of this operator is shown in the Figure 3. The crossover point is randomly generated. According to this point, two children are created. There are also other versions of crossover operator – a two-point crossover, an uniform crossover, etc.

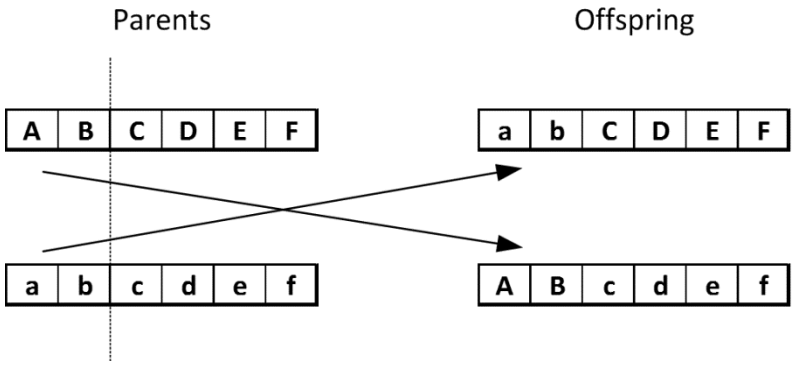


Figure 3. One-point crossover

The second genetic operator – *mutation* – generally changes the value of one gene (see the Figure 4). If the bit representation is considered, the mutation performs the bit negation of a gene. In comparison to the crossover, the mutation produces only one child (mutant).

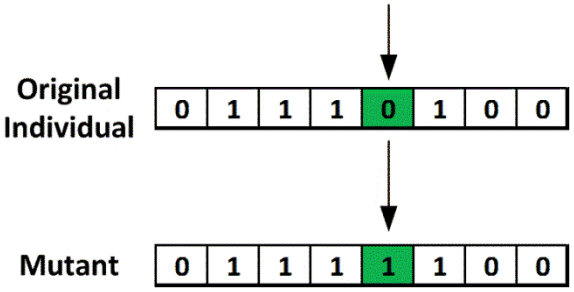


Figure 4. Mutation

The genetic operators are controlled by two parameters. The *probability of crossover* (P_c) says how often the crossover will be performed. Usually, this parameter takes value range from 0.6 to 1.

The *probability of mutation* (P_m) defines how often a gene of an individual will be mutated. The P_m can be in relation to a single gene of the population or to the whole chromosome. The value of this parameter is often very low (e.g. 0.1%). In *SGA*, the mutation is considered to be an auxiliary operator. [10][38]

Selection

Most often, the selection process is based on fitness values of the individuals. Two basic selection operators are discussed in the literature – *roulette wheel selection* and *tournament selection*.

The *roulette wheel* uses the relative fitness value which determines the selection probability of an individual. This probability is defined by the formula [10][38]:

$$p(a_i) = \frac{f(a_i)}{\sum_{j=1}^{N} f(a_j)} \quad (1);$$

where $p(x)$ *.....selection probability of the individual,*

a_i, $a_{i,}$ *....individual (chromosome),*

N*.....population size,*

$f(x)$*.....fitness function.*

Each value of selection probability determines the sector size in a virtual roulette wheel. A randomly generated number specifies the position in the roulette wheel; thereby an individual is selected. This process is repeated until the new population is filled.

The *tournament selection* is a high-frequently implemented method. This selection takes a random uniform sample of a certain size $q > 1$ from the population. The best individual of these q individuals will survive for the next generation. [10][38]

The selection techniques are commonly complemented by *elitism*. *Elitism* is a technique which ensures moving of the best individual (the leader) into the new population. [10]

3 EVOLVABLE HARDWARE

3.1 Basic Concept

The concept of the *evolvable hardware* (and the *evolutionary circuit design*) is based on the cooperation of an evolutionary algorithm and an appropriate reconfigurable system/circuit/structure. The algorithm searches for the configuration of the circuit in order to achieve the required circuit behaviour. Thus the task needs to define a suitable representation of the chromosome/genotype and a fitness function. The algorithm generates new individuals/chromosomes that are evaluated by the fitness function and the input stimuli. According to the calculated fitness value, the algorithm continues searching for the required functionality. The basic concept is shown in the Figure 5. [9][10]

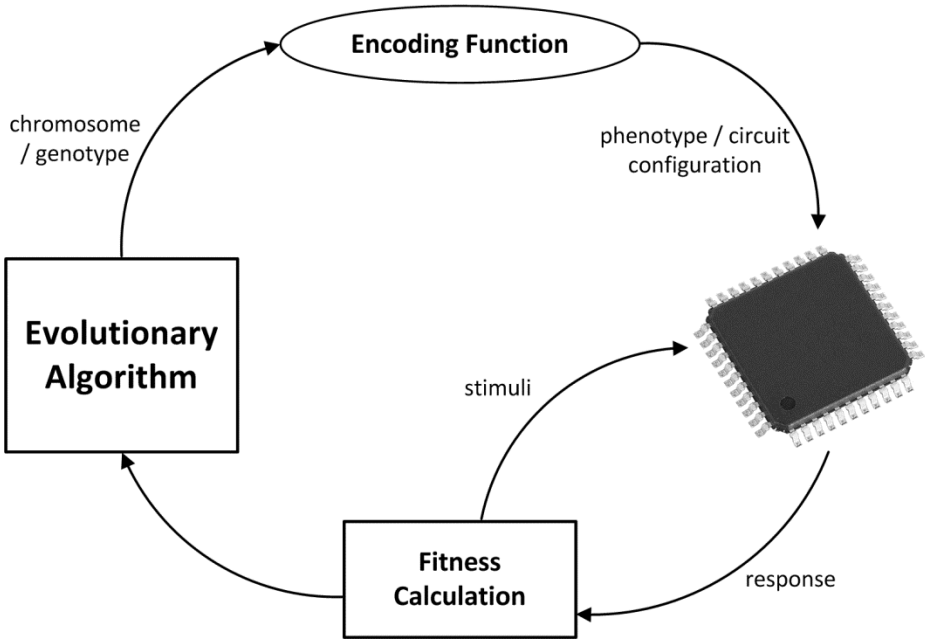


Figure 5. Basic concept of the Evolvable Hardware

The fitness function is based on the correspondence between the response of the candidate circuit/solution and the required response. The case of a combinational circuit is the simplest; the input stimuli (vectors) are applied to the inputs of the candidate circuit. The obtained responses form the truth table of the circuit that is compared with the correct (required) truth table.

Indeed, the evolutionary algorithm can also be used to optimize an already designed solution – this approach is called the *evolutionary optimization*.

3.2 Evolvable Hardware vs. Evolutionary Circuit Design

The cooperation of the evolutionary algorithms with the reconfigurable structures (systems) can be used in two similar domains – the *Evolvable Hardware* and the *Evolutionary Circuit Design*. Nevertheless, these terms should not be confused.

The *evolutionary design* deals with the design of a circuit by the means of evolutionary techniques. The evolution is used only for the design phase. This method of circuit design makes use of the mathematical model of the reconfigurable structure or of the system with variable parameters. This model is used for the fitness calculation. In other words, the fitness function is calculated by the means of a circuit simulator. Generally, the special software is designed and implemented for this purpose. A new and innovative solution to the circuits can be found by these techniques. The resulting circuit can be implemented as real hardware. The final real hardware does not use an evolutionary algorithm in any way. The evolutionary design is not limited by the circuit design; for example the publications [39][40] discuss the evolutionary design of antenna. [10]

In the case of the *evolvable hardware*, both parts – the reconfigurable circuit/structure and the evolutionary algorithm – have to be really implemented. The evolution process can run whenever. It means that the circuit/hardware can change its function dynamically in time. A circuit performing various functions can be obtained by this approach. To change its function, it is necessary to define the new required functionality first and then the evolution can be started. If the evolution finds a suitable configuration/solution, the system performs the new 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 the candidate solutions (individuals) can be very difficult – time-consuming. For that reason, the speed of evolution (searching for configuration) depends especially on the implementation of the fitness function. In the *evolvable hardware* domain, 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 point of view. [9][10]

The fitness function of the *evolvable hardware* is not static; it is modified when the superior system changes the required functionality or when the environment of the

evolving system changes. From this point of view, the system may be classified into several categories [10]:

- *Embedded evolutionary circuit design*

  The evolution of this system is started rarely. Despite of this, the evolutionary process is required. The system works for a long time with unchanged functionality; but from time to time, the request of functionality change arises. After this request, the evolution starts and searches for a solution corresponding with the new specification. If this solution is found, the evolution is terminated and the system uses the result of the evolutionary process. A significant feature of these systems is the fact that their function can be interrupted when it is necessary to perform the evolution. An example of this system is published in [5] and it describes the controller of a prosthetic hand. [10]

- *Self-adaptive systems*

  These systems perform the evolution continually. They have to react to the changes of the environment and the functional specification. The demonstration of this system is discussed in detail and implemented in the chapter 5. The self-adaptive system implements usually two reconfigurable circuits/structures. The first circuit is used for the processing of useful data. The second one makes it possible to calculate the fitness function so that the processing of useful data is not interrupted or affected. [10]

- *Self-triggered evolution*

  This category lies between the two previous categories. The system detects an event when the evolution has to be started. In general, these events occur periodically. [41]

- *Online evolution*

  This kind of evolution is mentioned in relation to robot controllers published in [6].

3.3 Evalution of Candidate Solutions

The quality of the candidate solutions is given by the *fitness value* which is a product of the *fitness function*. Indeed, there is no a universal fitness function and its definition is task-dependent. In spite of this, in the digital circuit domain, two basic cases are defined. If a circuit can be described by the truth table, the fitness function calculates the

correspondence between the real response of the candidate circuit (solution) and the ideal (required) truth table. In other cases (e.g. design of filters), the fitness function based on the *Method of Least Squares* can be used.

Except the definition of the fitness function, it is necessary to determine the approach to the evaluation. The following approaches have been introduced:

- *Extrinsic evolution*

  The candidate circuits are evaluated using a circuit simulator; it means the software. The results of the evolution are not tied to any particular hardware. For example, the system evaluates the candidate solution by the means of the SPICE simulator. The evolved circuit is defined by the netlist/diagram and it may be implemented by an arbitrary real hardware (TTL circuits, FPGA, CPLD, etc.). Generally, the evaluation using a simulator is slower. [10]

- *Intrinsic evolution*

  All candidate solutions are evaluated in real hardware (reconfigurable circuit). It should accelerate the evaluation process. [10]

- *Mixtrinsic evolution*

  This approach was developed by Stoica [42] to overcome the portability problem. A part of the individuals is evaluated extrinsic in target hardware, and some individuals in the software simulator within the same population. [10]

The next two categories deal with the portability and implementation of reconfigurable circuits:

- *Unconstrained evolution*

  In the chapter 1.1, the Thompson's experiment has been introduced [1]. This experiment is a perfect demonstration of an *unconstrained evolution*. The evolution algorithm can use the selected cells of the FPGA device in any way. The final evolved circuit used asynchronous feedback and delays of data paths. For the sake of these techniques, the final configuration bitstream was not portable to other FPGA devices (of the same type). [10]

  In other words, the candidate solution (circuit) may take different fitness values if it is evaluated using different reconfigurable circuits (of the same kind). This

behaviour is known as the *portability problem* [42]. It can be stated that the *unconstrained evolution* exploits physical properties of the chip/device. [10]

- *Constrained evolution*

  In comparison to the *unconstrained evolution*, this approach ensures that the candidate circuits take just one fitness value even when it is evaluated by different reconfigurable circuits. Hence, the portability problem is eliminated. [10]

3.4 Reconfigurable Circuits

The choice of a suitable reconfigurable circuit is strictly task-dependent. It is also necessary to define the fit level of abstraction. This level determines the fundamental elements that are represented by a chromosome. The representation of a chromosome usually reflects the architecture of a given reconfigurable circuit. For example, a circuit can be based on logic gates, discrete components (transistor, resistor, etc.), higher functional elements (multipliers, adders, multiplexers, etc.), nanostructures, etc.

Generally, the following features are required for the reconfigurable circuit/structure to be used in the evolvable hardware domain [10]:

- *Fast reconfiguration*

  The reconfiguration of target circuit has to be performed as fast as possible. The possibility of partial reconfiguration is welcome.

- *Unlimited number of reconfigurations*

  The circuit can be reconfigured an unlimited number of times. For example, the old GAL devices do not meet this requirement.

- *Safe configuration*

  It must be ensured that any configuration does not destroy the circuit.

- *Documented bitstream*

  It is necessary to know the data format of configuration bitstream. It has to be available and open to the user.

- *Controllability and observability*

  The inputs and outputs of the used circuit can be easily available. The system feeds the inputs of the circuit by the stimuli and reads its output response.

Nowadays, the main emphasis is put on the FPGA devices. However, all FPGA devices do not support partial reconfiguration; and above all, the format of their configuration bitstream is not usually documented and opened. Further, an uncorrected configuration (random mistakes in the bitstream are protected by a CRC) of an FPGA also may destroy a chip. Despite these facts, there are projects that modify the FPGA bitstream directly. Except the above-mentioned Thompson's project, the FPGA devices from Xilinx Corp. were used in projects focused on the adaptive image processing [43][44]. The new promising FPGA devices contain hard processor cores that can perform genetic operators and dynamic partial reconfiguration is supported [45]. It is also possible to use the reverse-engineering for obtaining the configuration format of an FPGA device [46].

To avoid using the native reconfiguration of an FPGA device, the concept of the *Virtual Reconfigurable Circuit (VRC)* has been established in [47]. This concept represents an implementation of a domain-specific reconfigurable circuit on the top of an FPGA device. The designer can implement such structure that is fit for the defined task.

4 CARTESIAN GENETIC PROGRAMMING

4.1 Background

This section describes the algorithm that was introduced by Miller and Thompson in 1999/2000 in the publications [48][49]. It grew from a method of evolving digital circuits developed by Miller et al. in 1997 [50].

Cartesian Genetic Programming (CGP) is a powerful tool suitable for the evolutionary design of combinational logic circuits. However, it can be also used in other tasks. In *CGP*, programs are represented in the form of directed acyclic graphs. This graph forms a reconfigurable structure which is used for evolving circuit. The Figure 6 shows an example of a *CGP* graph representing a digital circuit. The reconfigurable structure is modelled as a set of computational nodes in a grid/matrix organization. The nodes can represent gates or other elements of a digital circuit. The author of this thesis also uses the term *cell* instead of *node*. The size and topology of a structure is defined as n_c (columns) x n_r (rows). Each node (cell) has n_n inputs and one output; it can implement one node/cell function out of the group of defined functions. It can be imagined that the node implements a look-up table function. Only one selected function of this table is performed. Other parameters of *CGP* are the number of primary inputs (n_i) and program/primary outputs (n_o); these two parameters define the number of inputs and outputs of the designed logic circuit. The configuration information of the structure (chromosome) determines the function of particular nodes/cells, the interconnection among them, the interconnection between the primary inputs and the cells, and the interconnection between the program/primary outputs and the cells. For the interconnection in the structure, there are several rules. The feedback is not allowed; its implementation results in a difficult evaluation of the structure. The primary input can be connected as an input to all cells. The connectivity among cells is limited by the so-called *levels-back parameter* (*l-back* parameter), which determines the columns whose outputs can be connected to the current cell. For example, if *l-back* = 1, only the cell outputs of the immediately previous column may be used as input for the current cell. If the *l-back* parameter is set to maximum (n_c), there are no limits for the connectivity among cells. A higher value of the *l-back* parameter enables a greater possibility of connectivity; thereby a more complicated circuit may be generated. However, we must take into account that a high value of the *l-back* increases the space where the algorithm searches for suitable solutions. *CGP* uses an integer representation

(one gene of a chromosome is represented by one integer); each cell is defined by n_n+1 integer values/genes. The first n_n genes determine the input nodes of the cell; these genes are called *connection genes*. The last integer/gene is called *function gene* and it determines/addresses certain function which is performed by a node. The end of a chromosome is constructed by genes which define the nodes/cells representing primary outputs. The total length of the chromosome is defined as: [51][52]

$$\Lambda = n_r n_c (n_n + 1) + n_o \quad (2);$$

Note to the formula (2): The total length of the chromosome is represented by the symbol Λ or Λ_{CGP} in publications [52] and [9]. This convention is used in the rest of this thesis. However, Miller [53] uses the symbol L_g.

The values of the genes (i.e. alleles) are highly constrained. The *function gene* must take a valid address of the allowed cell/node function. The *connection genes* take values that point to achievable nodes. It means that the valid gene values depend on a type of the gene and its position within a chromosome.

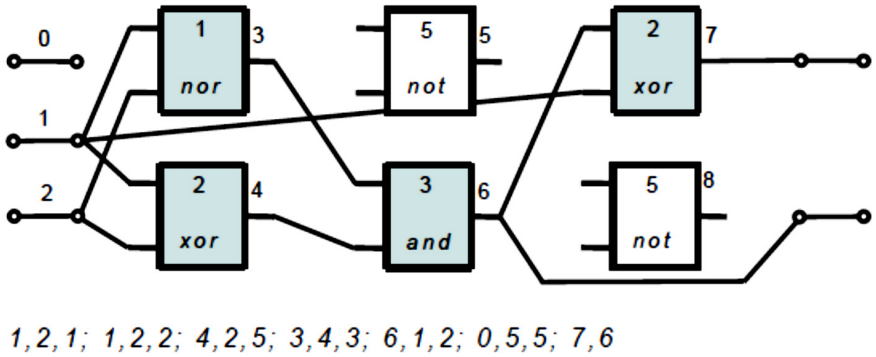


Figure 6. Example of a *CGP* graph [52]

The whole chromosome represents a genotype. The decoding of a genotype → a phenotype creates final program/circuit. It is obvious that if any node output is not utilised in the calculation of the output response, the size of the phenotype is smaller than the genotype size. In other words, the genotype represents all nodes/cells in a grid graph/structure; the phenotype contains only the cells that are needed for the output data calculation. In evolutionary circuit design domain, it means that the phenotype is formed by gates/components implementing the digital circuit. The cells/nodes which are not used in a phenotype are called 'inactive', the used cells called 'active'.

4.1.1 Evolutionary Algorithm

The algorithm that is used for the search for solutions is very similar to the *Evolutionary Strategy – (1 + λ)-ES*, where the character λ defines the number of mutants/offspring and the constant number 1 represents the fact that the algorithm uses only one parent. It is necessary to ensure that, so that the genes take correct values only.

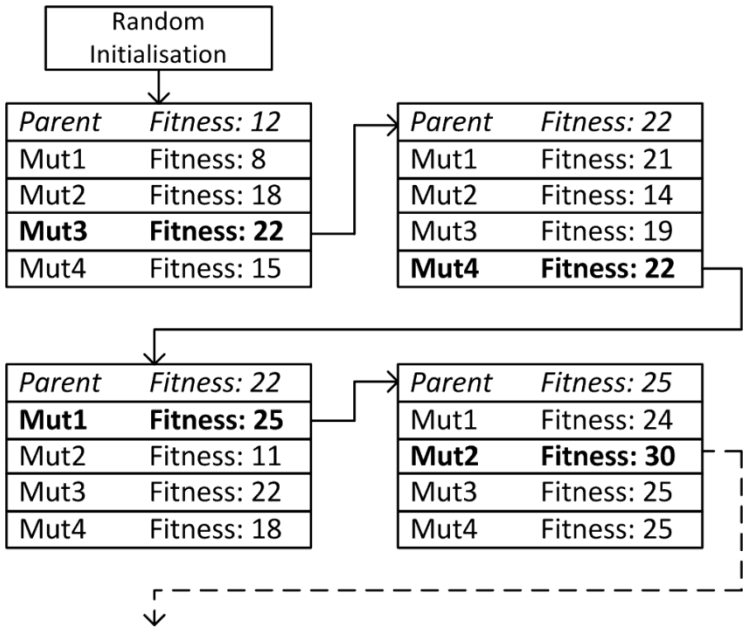


Figure 7. Principle of the search algorithm

The principle of the search algorithm is shown in the Figure 7. At first, the initial population is generated randomly. The process of mutation produces (by a point mutation operator) λ mutants (offspring) by random changing of a few genes. After the calculation of the fitness function, the selection is performed. It selects the best individual which will represent the parent in the next cycle of the evolution. A special case can occur when there are mutants (or one mutant) with the same fitness value as the parent and there is no offspring better than the parent. In this case the algorithm prefers the mutant (products of mutation) to its parent. This step is very important for *CGP*; it makes good use of redundancy in *CGP* genotypes.

This genetic redundancy is needful for the good progress of the evolution. In *CGP* there can be genes having no influence on the phenotype. This phenomenon is called neutrality and it is beneficial to the efficiency of the evolutionary process. [49][53][54][55][56]

If the fitness function is performed, only active nodes/cells have to be calculated; inactive nodes can be neglected. Listing 1 shows the procedure of determining active nodes. The symbol *M* denotes the maximum number of addresses in the *CGP* graph; *G* is an array of

genotype; the boolean array *NU* determines the used/active nodes. The number of used nodes is given by the symbol n_u. The *NP* denotes array where the addresses of the used nodes are stored.

The other algorithm's procedures created by Miller are published in [53].

```
1: NodesToProcess(G,NP) // return the number of nodes to process
2: for all i such that 0 ≤ i < M do
3:         NU[i] = FALSE
4: end for
5: for all i such that Lg −no ≤ i < Lg do
6:         NU[G[i]] ← TRUE
7: end for
8: for all i such that M −1 ≥ i ≥ ni do // Find active nodes
9:         if NU[i] ← TRUE then
10:                index ← nn(i −ni)
11:                for all j such that 0 ≤ j < nn do // store node genes in NG
12:                        NG[ j] ← G[index+ j]
13:                end for
14:                for all j such that 0 ≤ j < Arity(NG[nn −1]) do
15:                        NU[NG[ j]] ← TRUE
16:                end for
17:        end if
18: end for
19: nu = 0
20: for all j such that ni ≤ j < M do // store active node addresses in NP
21:        if NU[ j] = TRUE then
22:                NP[nu]← j
23:                nu ← nu +1
24:        end if
25: end for
26: return nu
```

Listing 1. Determining active nodes [36]

If the digital circuits are designed by *CGP*, a minimal solution/circuit is often required. The following fitness function takes into account the circuit functionality and the number of the used cells [52][9].

$$f = \begin{cases} b & when\ b < n_o 2^{n_i}, \\ b + (n_c n_r - z) & otherwise; \end{cases} \quad (3);$$

where b represents the circuit functionality (according to the required truth table) – it is number of correct output bits obtained as response for all possible assignments to the inputs; and z denotes the number of used/utilised cells. Usually, the circuit is optimized to decrease the number of used cells. The algorithm takes into account the number of used cells only in cases when the full functionality is found. This procedure is very important. The alternative function was introduced by Gajda and Sekanina in [52].

4.1.2 Mutation Process

As mentioned above, the mutation used in *CGP* is a point mutation operator. The mutation process randomly chooses the gene position in a chromosome. The gene on this location is replaced by a valid random value.

The actual number of genes mutated within one mutation process is given by the parameter called *mutation rate*. It uses the symbol μ_g. The mutation rate can also be expressed relatively as a percentage of the total number of genes in the genotype/chromosome – the symbol μ_r is used for it. It implies $\mu_g = \mu_r \Lambda$. Note that in this thesis, the absolute value (it means μ_g) of the mutation rate is used.

The Figure 8 shows an example of the mutation. The program output gene O_A was mutated; the gene value changes from 6 to 7. This mutation causes a significant modification of the phenotype. The active cells (gates in this case) are represented by a solid line; the inactive ones by a dashed line. [53]

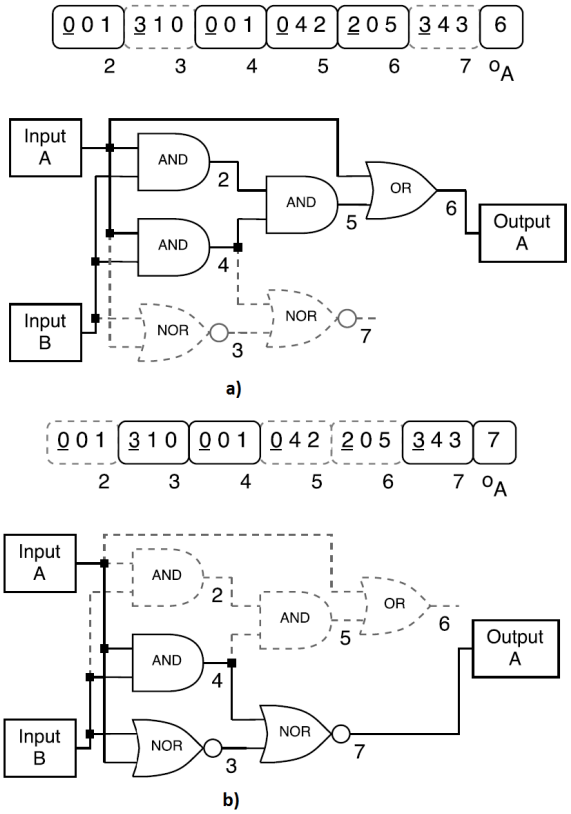


Figure 8. Example of the point mutation operator [36]; a) before the mutation, b) after the mutation

4.1.3 Other versions of CGP

In recent years, other versions of the algorithm were derived from *CGP*. Some authors removed the restriction related to the use of the feedback. By this, the *Cyclic CGP* was created. [53][56]

The *Embedded CGP (ECGP)* is a *CGP* which incorporates sub modules/subprograms. In the *ECGP*, modules are created and destroyed during the evolutionary process. The *Self-Modifying CGP (SMCGP)* is also established, it implements self-modification functions/instructions that cause alteration of the code itself. Both algorithms are described in [53].

4.2 Software Design Tool

4.2.1 Proposed Design System

The design of a system based on *CGP* can be divided into several main parts which need to be implemented. Naturally, the Evolutionary Algorithm (Evolutionary Strategy) represents only the core of the system; the other parts are very significant as well. The designer also has to implement an appropriate reconfigurable structure and a suitable fitness function. These tasks may be highly time-consuming. It must be taken into account that the use of evolutionary algorithms is represented by a large number of experiments. Usually it is necessary to change parameters during the evolutionary design process. It is clear that the design of a comprehensive design tool (system) based on an evolutionary design may be very useful. This tool could significantly decrease the amount of time required for the design of a system based on the EA.

The benefit of a design tool can be more important if the tool supports the export of results to an embedded system – it means an automated generation of source code for a processor or an FPGA device. Such design tool can represent a rapid-prototyping system. From the user's point of view, it enables a simple simulation and an implementation of issues. Unfortunately, the scientific community offers only few projects which implement the evolutionary design by means of *CGP*. Most of them are provided in source codes, e.g.[57] refers to *CGP* programs (in the C programming language) that can be compiled into programs that can handle three kinds of data with *CGP*, namely Boolean, integer and floating point. These source codes were created by Julian Miller (the *CGP* architect). Other known implementations are available in Java programming language or in the codes for Matlab [58][59]. However, the focus on the design of digital circuits is given in the tools described in [60]. These tools produced by Vašíček and Sekanina make it possible to design any combinational logic circuit by the means of *CGP*. Their tool called *Cgpviewer* enables to display the *CGP* chromosome (a result circuit), simulate it and convert it to a VHDL file. Despite these facts, the available tools do not provide a user-friendly environment and the user always has to modify the source codes to set the desired parameters of *CGP*.

In this thesis, the author's goal was to design a tool with a suitable user-friendly environment, which requires only a minor understanding of the evolutionary design and the *CGP* domain. This design tool is mainly focused on the utilization of *CGP* for the

evolutionary design and the implementation of evolvable hardware. The support of FPGA devices as a target platform is assumed. The proposed system should enable fundamental design steps – the definition of the problem, the simulation (sensitivity analysis) and the implementation of the result of the evolutionary design. It was also assumed that the tool enables the generation of source code for particular parts of evolvable system (structure for the *CPG* primarily). The Figure 9 describes the design flow used with the proposed system.

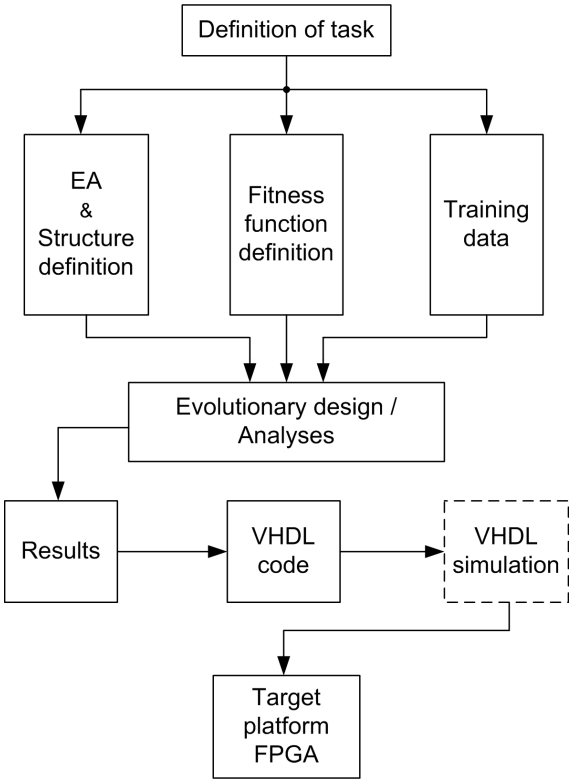


Figure 9. Design flow of the proposed system

After the general definition of a task, the user has to determine the parameters of the search algorithm and the reconfigurable structure. The system should offer several predefined kinds of fitness function. However, it is obvious that the possibilities of the predefined fitness function are limited, because it isn't feasible to cover all requirements of real-world applications. The preparation of training data is the part of design whose implementation isn't assumed within the proposed design tool. This data depends strongly on the application. The tool has to offer only an appropriate interface for its import.

Of course, the core of the design tool is the performance of the evolutionary design. This functionality should be supplemented by the possibility to perform analysis, observing influence of algorithm parameters, etc. The results of the evolutionary design must be well interpretable – the automatic generation of final designed circuit's diagram is essential.

However, the results could be described by other ways – a logical expression, a SPICE netlist or hardware description language (for example: Verilog or VHDL). The author assumed the use of a circuit diagram and a VHDL description. The design result expressed by this description can be directly implemented by an FPGA device. Nevertheless, a simulation after the implementation is recommended.

In the following subchapters, the successfully implemented parts of the system and its functionality are introduced.

4.2.2 Implementation of Tool

The design tool was named as *Evolutionary Designer*; it was designed by the means of the C# programming language and it is available for Windows platform.

4.2.2.1 Search Algorithm

The designed tool implements the search algorithm coming out of *CGP*. That is derived from the Evolutionary Strategy algorithm. The algorithm is determined by the expression $(n + \lambda)$; where n represents the number of parents and λ determines the number of mutants. This implies that the implemented search algorithm enables various variants of Evolutionary Strategy; not only the $(1 + \lambda)$ type which is supposed to occur in the *CGP* domain. The principle of the search algorithm is shown in the Figure 7 in the chapter 4.1 and it is also discussed and described in this chapter. Note the fact that the implemented algorithm always selects an offspring (a mutant) if one or more mutants take the value of fitness which is equal to a parents' fitness.

In the Figure 10, the dialog window for algorithm setup is shown. The user may set several parameters of the search algorithm – the number of parents and mutants, the mutation rate, the maximum of generational cycles (the maximal number of performed generations) and the maximum of fitness.

The mutation rate determines directly the number (an absolute value) of genes which will be mutated within the mutation process. The item *Cycles Limit* represents the termination condition. The algorithm is terminated if the set value of generation cycles is achieved. The user can turn this condition off if this *Cycles Limit* is set to zero. The second termination condition is determined by the item *Max Fitness*. This value doesn't mean only the maximum of fitness value. It can be used even when the algorithm looks for the minimum of fitness value. In other words, the algorithm is terminated if this fitness value is achieved.

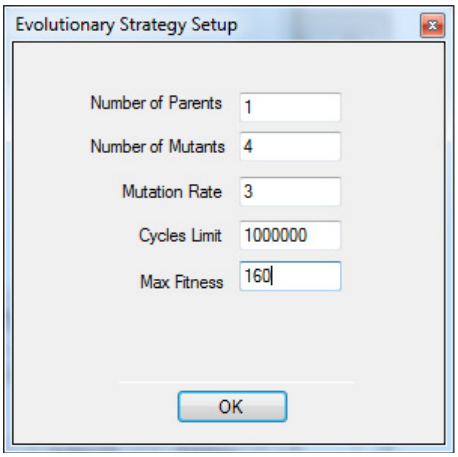


Figure 10. Dialog window of algorithm setup

4.2.2.2 Definition of Reconfigurable Structure

The designed tool offers a detailed setup of possibilities of a structure for *CGP*. At the beginning of the setup procedure, the user defines sets of cell functions (only two-input cells are supported; the user may define one or more of these sets). A particular set contains functions which are available in a single cell/node of *CPG* structure. The structure can utilise only one set of cell/node functions – cells form homogenous structure. It can also use various sets of functions – various cells form a heterogeneous structure. The user can compile his/her own set of cell functions. This process is very easy; the user uses a simple dialog window (see the Figure 11) – *Cell Sets Manager*.

By the means of this dialog, the user creates a new set – the ID of this set and the number of functions must be defined. The function sets can contain a different number of functions. The design tool supports from 2 up to 64 functions for each set. If the set is created, it may be edited by the dialog *Cell Functions Setup* depicted in the Figure 12. In this dialog, the user adds and removes functions to form the required set; the figure shows the definition of the function set called "basic_gates" containing 4 cell functions – AND, OR, XOR and NOR.

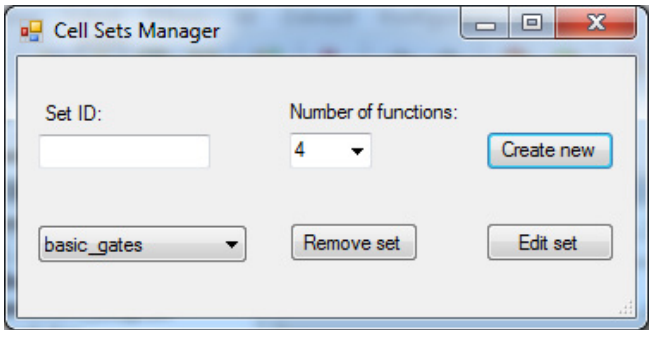


Figure 11. Cell Sets Manager dialog

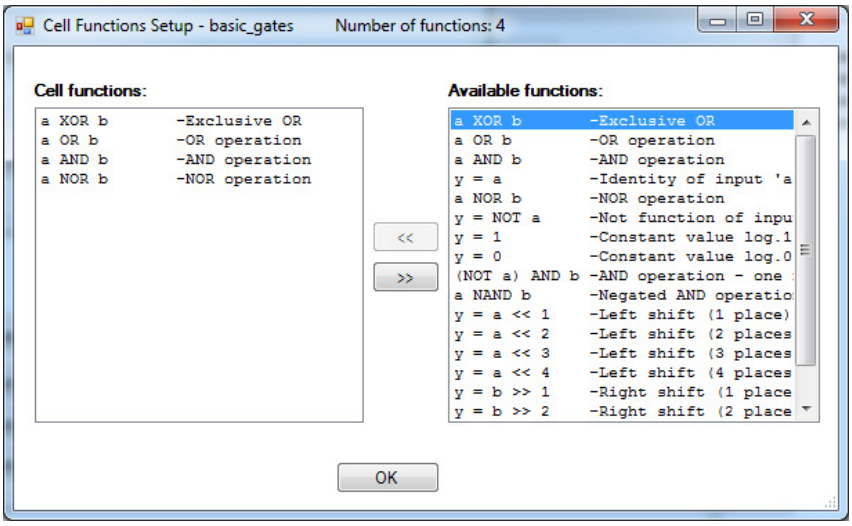


Figure 12. Cell Functions Setup dialog

For each available function, the tool defines four parameters. Three of them are easily deducible – the function name, the function description and the function source code. The name and the description are shown in the *Cell Functions Setup* dialog; the source codes of the functions are used when the tool calculates a fitness function. The last parameter is the reference to VHDL file describing the function. This VHDL model is in unified form and it can be used for the implementation of the function in an FPGA device (this features will be discussed in the following sections). In the Listing 2, the VHDL model of the XOR logical function is shown.

```
library ieee;
use ieee.std_logic_1164.all;

entity func_xor is
   generic
      (
      data_width : positive := 8
      );
   port
      (
      input_data_A : in  std_logic_vector(data_width-1 downto 0);
      input_data_B : in  std_logic_vector(data_width-1 downto 0);
      output_data  : out std_logic_vector(data_width-1 downto 0)
      );
end;
```

Listing 2. VHDL model

The name of the entity is formed by the merger of the string “func_” and the name of the function. The entity contains one generic parameter – *data_width* – that determines the bit width of the ports. In the port declaration, there are three obligatory items – the inputs ports *input_data_A, input_data_B* and the output port *output_data*. The architecture (which is very simple in this case) has to be implemented so that the VHDL model is correct for arbitrary valid (positive) value of the generic parameter *data_width*.

If at least one set of cell functions is established, the other *CGP* structure parameters can be set by means of the dialog shown in the Figure 13. The user can define the following basic items: the number of rows and columns, the number of primary inputs and outputs, the *l-back* parameter and a default set of cell functions. The item *Cell Function Set* determines the used default set of cell functions and is valid if the user does not use the "Advanced Structure…" choice. This choice makes it possible to define the structure using more sets of cell functions. The user may determine a function set for each cell. This feature is controlled by the dialog *Advanced Structure Setup* – see the Figure 14. The user can specify whether the selected function set belongs to an individual column, row or cell/node. By using these choices, a heterogeneous structure can be defined – the cells implement different function sets. This described functionality is not usual, but it can serve as a tool to analyse irregular structures.

Note that the tool does not allow direct connecting a primary input to a primary output! CGP definition enables this type of connection; however, the author considers that it is needless (this type of connection could be beneficial only in trivial tasks).

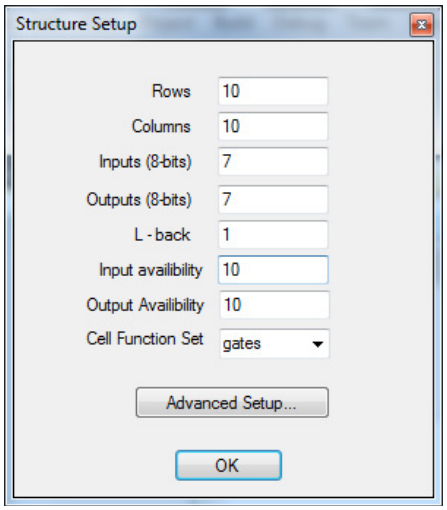


Figure 13. Structure Setup dialog

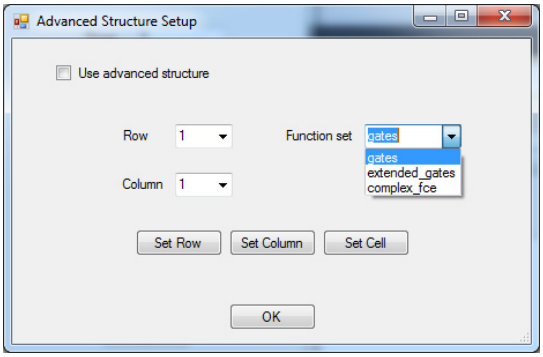


Figure 14. Advanced Structure Setup

In addition, there are other two 'unusual' parameters – the *Input Availability* and the *Output Availability*. These parameters affect the connectivity of a structure. The *Input Availability* specifies the columns whose cells can be connected to primary inputs. For example: if this item takes the value of two, only the first two columns of the grid structure can be fed by the primary inputs. Usually, this type of connectivity is not limited. The latter parameter – *Output Availability* – reduces the connectivity of the primary outputs conversely. This means, if it takes the value of 1, only the last column can represent the primary outputs of the structure. It follows the maximum of both parameters is equal to the number of columns. The author established these new parameters because they reduce the search space of the design process; their effect may appear more interesting when the implementation of a structure by an FPGA device is supposed.

All signals in the designed software tool are represented by 8-bit vectors. This fact does not limit the bit-oriented functions in any way.

4.2.2.3 Fitness Function

The tool contains several options to determine the kind of a fitness function. Two basic tasks are assumed – the signal of the structure represents a one-bit signal or an unsigned/signed vector. According to the signal meaning, the kind of fitness function can be defined. The fitness function with one-bit signals is based on the correspondence between the real and the required truth table. This correspondence is expressed by means of *Hamming Distance* – the algorithm searches for minimum *Hamming Distance* between the real and the required response of a circuit. If the signal of the *CPG* describes an unsigned/signed vector, the system can utilise the fitness function using the principle of *Least Square Method*. Both kinds of fitness function can optimize the size of the result circuit – this feature is enabled or disabled by the user. It means that the evolution can search for a circuit with a correct function and the minimal hardware requirements.

The fitness function for the tasks using bit signal is defined by the formula (3) mentioned in the chapter 4.1. If the full functionality of the circuit is found, the number of unused cells/nodes is added to the resulting final value. For example, assume that the *CGP* structure is 7x7 cells and the 5-bit majority circuit is desired. The full functionality of the circuit implies the fitness value equal to 32. If the circuit is constructed of 25 cells, the number of unused cells is 24 (= 49 – 25); and the result fitness value is equal to 56. The algorithm takes into account the number of used cells only in the cases when the full functionality is found. This is very important. The tool also enables to mask certain bits of

primary outputs. It means that the user defines bit's positions that won't be included into the fitness calculation.

For the design circuit as the image filter, etc., the signal of the *CGP* structure represents an unsigned/signed number. In these cases the system uses another fitness function. It is based on the difference between the real and the ideal response circuit to the training/test data; it can be defined as:

$$f = \sum_{j=0}^{n_0-1} \sum_{i=0}^{N-1} \left| ir_{ji} - rr_{ji} \right| \quad (4);$$

where ir_{ji} denotes the ideal and rr_{ji} the real response of the circuit to training data; N is the number of training vectors; n_0 defines the number of primary outputs. From (4) it is obvious that the evolutionary design performs the minimization of the fitness function, by contrast to (3) where an algorithm searches for a circuit with the maximum value of fitness. The designed tool enables to include the size of candidate circuit into the fitness function even when a fitness function given by (4) is used. In this case, the number of used cells is added to (4), if the full correspondence between ideal response and real response is found.

4.2.2.4 Training Data Import

The well-known format – CSV file – was chosen as the interface for the import of the training data. This format is supported by MS Excel and its structure is very simple. The items in lines are separated by a semicolon; the <CR> char is a final mark of a line.

The first line of the file with the training data must contain the integers notifying the number of inputs and outputs of the circuit. These numbers serve to check the training data. Next lines contain the training vectors; the input stimuli and the corresponding ideal responses. The data in the file must be 8-bit integers; it means values in the range from 0 to 255, -128 to 127 (signed). If a classical truth table is required, it has to be transformed to 8-bit vectors (one line of training covers 8 lines of the classical truth table of the logic circuit).

The Figure 15 shows the transformation truth table of 5-bit majority circuit to the CSV file. This circuit has 5 inputs and 1 output – this is noted in the first two cells in the CSV file. The second line of the file describes the first training vector – the first five integers (MSB is the leftmost) represent input stimuli corresponding with eight lines in the truth table. And the rightmost items define an ideal (required) response of the designed circuit. After

this simple transformation, the training data can be used for the fitness function calculation. In the case of a fitness function that works with signals in terms of integers, no transformation is needed.

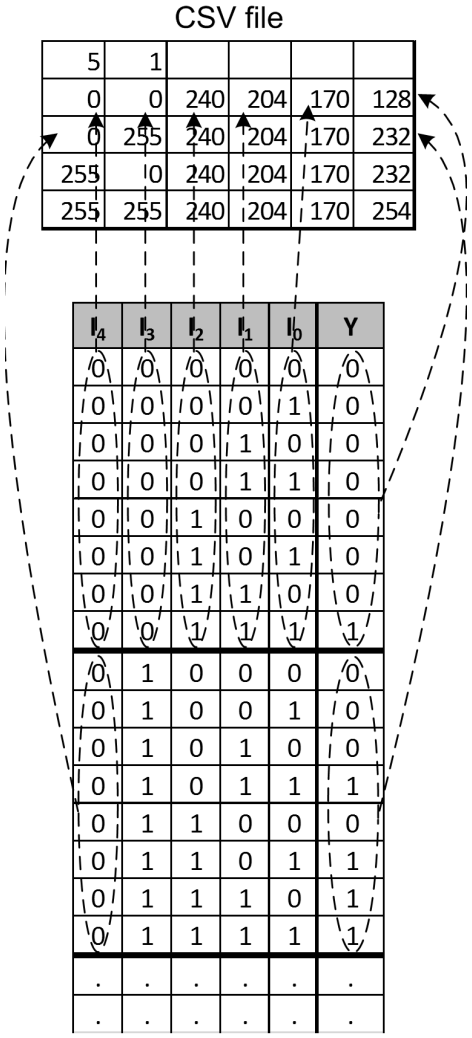


Figure 15. CSV file with the training data

If the cells in the *CGP* structure contain only strictly logic functions (AND, OR, etc.), the tool composes training vectors to 64-bit vectors. It enables to use parallel processing and accelerate the calculation of the fitness function; 64 cell responses can be obtained within one logic operation.

4.2.2.5 Result Viewer

The tool is equipped by a viewer that displays the evolution result in a diagram (see the Figure 16). This viewer can show all cells/nodes and their interconnectivity or it can show only the active cells/nodes. In additional, the user can depict only active cells related to

particular primary outputs. The viewer shows the functions which are implemented by the cells.

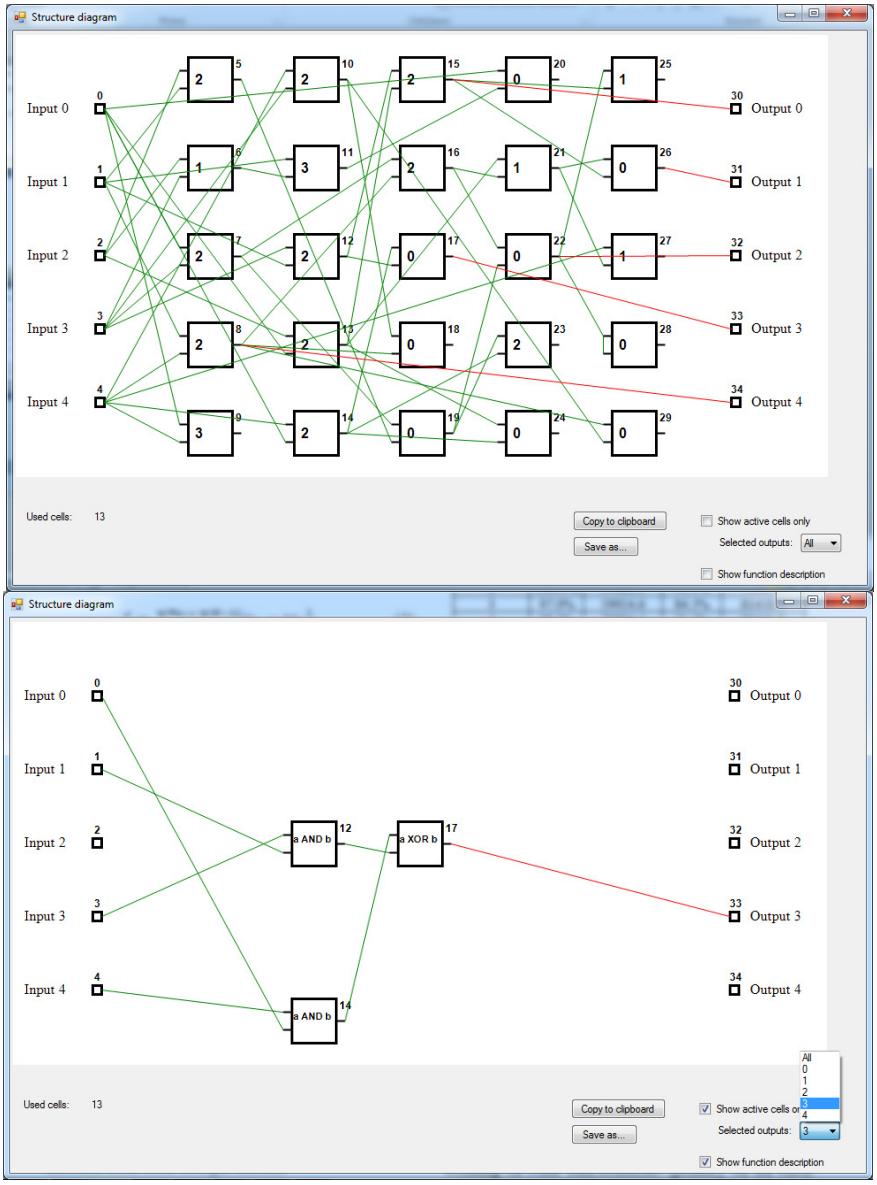


Figure 16. Structure diagram dialog

4.2.2.6 VHDL Output

The designed tool can generate VHDL files that describe the result of the evolutionary design. This feature requires only few settings which are obvious from the following figure.

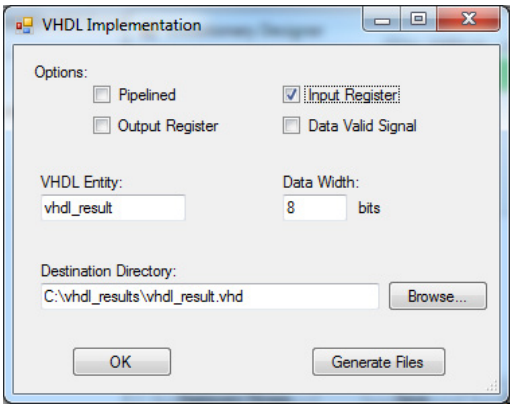


Figure 17. VHDL Implementation dialog

The user selects the destination folder, the name of the file and the name of the top-level entity. The result of the evolution can be implemented either as an asynchronous combinational circuit or as a circuit with input/output register. However, if the circuit contains more cells and these cells implement complex functions, the delay caused by logic elements can be too long. For that reason, the tool also offers pipelined processing of the result. This choice divides the resulting circuit into several pipeline stages by the insertion of the registers amongst particular cells. The *Evolutionary Designer* performs the whole process automatically, but it needs the VHDL models of the used cell functions (see the section 4.2.2.2 and the Listing 2).

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity parity is
    port
        (
        -- clock and reset port
        clk           : in std_logic;
        reset_n       : in std_logic;

        -- input ports
        input_data_0  : in std_logic;
        input_data_1  : in std_logic;
        input_data_2  : in std_logic;
        input_data_3  : in std_logic;
        input_data_4  : in std_logic;

        -- output port
        output_data_0 : out std_logic
        );
end;
```

Listing 3. Automatically generated VHDL code – 5-bit majority

The example of the entity of the automatically generated VHDL source code describing the 5-bit majority circuit is shown in the Listing 4 (full source code is available in the Appendix A).

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity parity is
    port
        (
        -- clock and reset port
        clk           : in std_logic;
        reset_n       : in std_logic;

        -- input ports
        input_data_0  : in std_logic;
        input_data_1  : in std_logic;
        input_data_2  : in std_logic;
        input_data_3  : in std_logic;
        input_data_4  : in std_logic;

        -- output port
        output_data_0 : out std_logic
        );
end;
```

Listing 4. Automatically generated VHDL code – 5-bit majority

4.2.2.7 Generation of Virtual Reconfigurable Circuit

By analogy to the VHDL implementation of the evolution result, the generation of the VHDL source codes implementing *Virtual Reconfigurable Circuit (VRC)* can be performed very easily. The *VRC* can be used for designing an evolvable system in the FPGA devices. It exactly describes the structure which is used for the calculation of the fitness function by *CGP*. The *VRC* represents quite a complex circuit with many parameters. The automatic generation of source codes of the *VRC* can reduce the time needed to implement the evolvable system based on *CGP*.

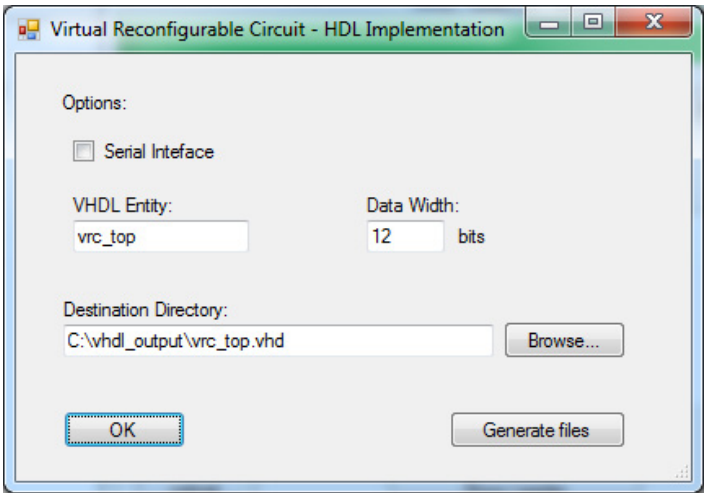


Figure 18. VRC implementation dialog

```
-- Virtual Reconfigurable Circuit
-- Entity name: vrc_top
--
-- VRC parameters:
-- Number of columns: 5
-- Number of rows: 5
-- L-back: 1
-- Input availability degree: 5
-- Output availability degree: 5
-- Config interface: Parallel
--
-- Generated by the Evolutionary Designer - 4.5.2013 23:30:21

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.vhdl_designer.all;

entity vrc_top is
        port
                (
                -- primary input
                primary_input_0 : in std_logic_vector(7 downto 0);
                primary_input_1 : in std_logic_vector(7 downto 0);
                primary_input_2 : in std_logic_vector(7 downto 0);
                primary_input_3 : in std_logic_vector(7 downto 0);
                primary_input_4 : in std_logic_vector(7 downto 0);
                primary_input_valid : in std_logic;

                -- primary outputs
                primary_output_0 : out std_logic_vector(7 downto 0);
                primary_output_1 : out std_logic_vector(7 downto 0);
                primary_output_2 : out std_logic_vector(7 downto 0);
                primary_output_3 : out std_logic_vector(7 downto 0);
                primary_output_4 : out std_logic_vector(7 downto 0);
                primary_output_valid : out std_logic;

                -- config data port
                column_0_config_data : in std_logic_vector(39 downto 0);
                column_1_config_data : in std_logic_vector(49 downto 0);
                column_2_config_data : in std_logic_vector(49 downto 0);
                column_3_config_data : in std_logic_vector(49 downto 0);
                column_4_config_data : in std_logic_vector(49 downto 0);
                output_0_config_data : in std_logic_vector(4 downto 0);
                output_1_config_data : in std_logic_vector(4 downto 0);
                output_2_config_data : in std_logic_vector(4 downto 0);
                output_3_config_data : in std_logic_vector(4 downto 0);
                output_4_config_data : in std_logic_vector(4 downto 0);

                clock_config : in std_logic;

                config_column_0_wr : in std_logic;
                config_column_1_wr : in std_logic;
                config_column_2_wr : in std_logic;
                config_column_3_wr : in std_logic;
                config_column_4_wr : in std_logic;
                config_output_0_wr : in std_logic;
                config_output_1_wr : in std_logic;
                config_output_2_wr : in std_logic;
                config_output_3_wr : in std_logic;
                config_output_4_wr : in std_logic;

                -- other signals
                clk_vrc    : in std_logic;
                clk_vrc_en : in std_logic;
                reset_n    : in std_logic
                );
end;
```

Listing 5. Automatically generated VHDL code – VRC implementation

Due to facts described in the next part of this thesis, there is only one limitation – the *l-back* parameter of the generated *VRC* is reduced to maximal value equal to 2. In other

parameters, the generated source codes precisely match the *CGP* structure defined in the section 4.2.2.2.

From the dialog shown in the Figure 18, it is obvious that the user can specify two main parameters – the *Data Width* of data signals and the type of configuration interface (parallel is set by default; serial interface can be set optionally). The Listing 5 demonstrates an automatically generated top-level entity (only entity's declaration); the complete VHDL source code is available in the Appendix B.

The implementation of the *VRC* is described in the chapter 4.4.3.1 in detail.

4.2.2.8 Test and Validation

The designed tool was used for the experiments performed in the rest of the thesis. The results of these experiments demonstrate the work of the *Evolutionary Designer* tool. For that reason, a special chapter dealing with the examples of functionality and benchmarks has not been written. Nevertheless, the author presents a brief comparison of the result of the designed tool with results obtained with *Tools4CGP* developed by Vašíček and Sekanina [60].

The evolutionary design of the multiplier 3x2 bits was used. The configuration of *CGP* was the following: the generation limit = 500,000; set of the cell functions: AND, OR, XOR, identity; the number of primary inputs and outputs was 5. The evolution looks for the full functionality of the circuit; it means that the required fitness value is equal to 160 (= 5 x 2^5).

There were 100 runs of the algorithm performed. The used topologies and the *l-back* values together with the results are presented in Table 1. From the table it is clear that both tested implementations of *CGP* provide very similar results. However, all runs of *Evolutionary Designer* found solution successfully. Seven runs of the *Tools4CGP* did not find a circuit with the full functionality. If the mean number of generations needed for successful evolution is compared together, it can be observed that the results given by the *Evolutionary Designer* are slightly more favourable. However, the differences are not significant and are caused by the fact that the *Evolutionary Designer* does not support the direct connection of the primary inputs to the primary outputs (see the subchapter 4.2.2.2).

Table 1. Comparison with Tools4CGP

Topology n_c x n_r	*l-back*	Mut. Rate	Tools4CGP [60]		Evolutionary Designer (Burian)	
			Mean # Gen.	Run Successful [%]	Mean # Gen.	Run Successful [%]
5 x 5	1	1	95,815	98	84,080	100
		2	79,233	98	67,572	100
		3	102,059	98	72,511	100
	5	1	71,179	99	54,847	100
		2	59,349	100	43,603	100
		3	70,961	100	55,704	100
10 x 10	10	1	58,351	100	50,940	100
		2	31,966	100	25,951	100
		3	22,693	100	20,873	100

4.3 Reduction of Fitness Calculations

This chapter deals with evaluation issue in the *Cartesian Genetic Programming* domain. It explores the possibilities of reduction of candidate solutions needed to evaluate. This reduction may accelerate the process of the evolution – evolutionary design, etc. The chapter presents the approach that detects changes in the phenotype and, based on that, the algorithm can omit the valuation of the candidate solution. The effectiveness of this approach is presented on evolutionary design of multipliers.

4.3.1 Time-Consumption of Evolutionary Design

Generally, the evolutionary design is relatively time-consuming. It is not possible to predict exactly the time needed for the evolution process. However, it may be expressed by the following formula [10]:

$$T_e = N_g\,(T_p + N_i\,.\,T_f) \tag{5};$$

where
T_e ..... *the whole time of the evolution (algorithm run),*
N_g ..... *the number of generations,*
T_p ..... *the time of production of population,*
N_i ..... *the number of individuals,*
T_f ..... *the time needed for the evaluation of an individual.*

This formula is valid for evolutionary algorithms generally. In the case of *CGP*, the parameter N_i represents the number of mutants produced in each generation of algorithm.

According to this formula, the possibilities of the acceleration of the evolutionary process may be discussed. It is clear that the parameter N_g is very significantly related to the speed of evolution. This parameter depends on the effectiveness of the search (evolutionary) algorithm and the area of the search space. If the common search algorithm of *CGP* is assumed, the number of mutants and topology of re-configurable structure can be affected. These parameters can reduce the number of generations. The parameter – T_p – depends on the implementation of the evolution process on the target platform. It presents the computational time related to the preparation of the population; it means the selection and mutation process. The remaining two parameters are the most significant because the calculation of the fitness function (i.e., the valuation of individual) is the key element of each application based on evolutionary algorithms. A good implementation of this function

could reduce the time needed for the evolution very markedly. If trivial tasks are not considered, this infers that the latter statement is applied.

$$T_p \ll N_i \cdot T_f \quad (6).$$

It is obvious that the evolution time depends mainly on the product $N_i \cdot T_f$. In other words, the evolution time is related to the time of the fitness calculation of the individual and to the number of individuals which have to be evaluated during the whole evolutionary process. This fact can be expressed as:

$$T_e \approx N_g \cdot N_i \cdot T_f \quad (7).$$

4.3.2 Proposed Approach to Reduction of Number of Fitness Calculations

To accelerate the process of evolution (evolutionary design), it is necessary to focus on the formula (7) mentioned in the previous subchapter. The parameter T_f is dependent on the used implementation and the target platform used.

In this thesis, the author assumes that T_f is constant and deals with the remaining possibility of acceleration of *CPG*. It consists in the reduction of the number of individuals requiring the fitness calculation (evaluation).

As mentioned in section 4.1, after the creation of mutants (offspring), it is necessary to calculate the fitness function of these new individuals – each new individual is evaluated (the fitness function is calculated). However, this approach includes certain redundancy.

Initially, it is assumed that the parent's chromosome contains inactive (unused) cells (genes); if the mutation process changes only this inactive part of the chromosome (it means that the phenotype of the individual remains unchanged), the new created individual (mutant/offspring) must take the same value of fitness function. This feature is called *silent mutation* by Miller in [72]. In other words, if the mutant process does not change the phenotype of the individual, the fitness calculation of such an individual is not necessary! The modified algorithm which detects the changes in the phenotype can reduce the number of individuals requiring a fitness calculation. The proposed and designed modified algorithm is described by the following diagram.

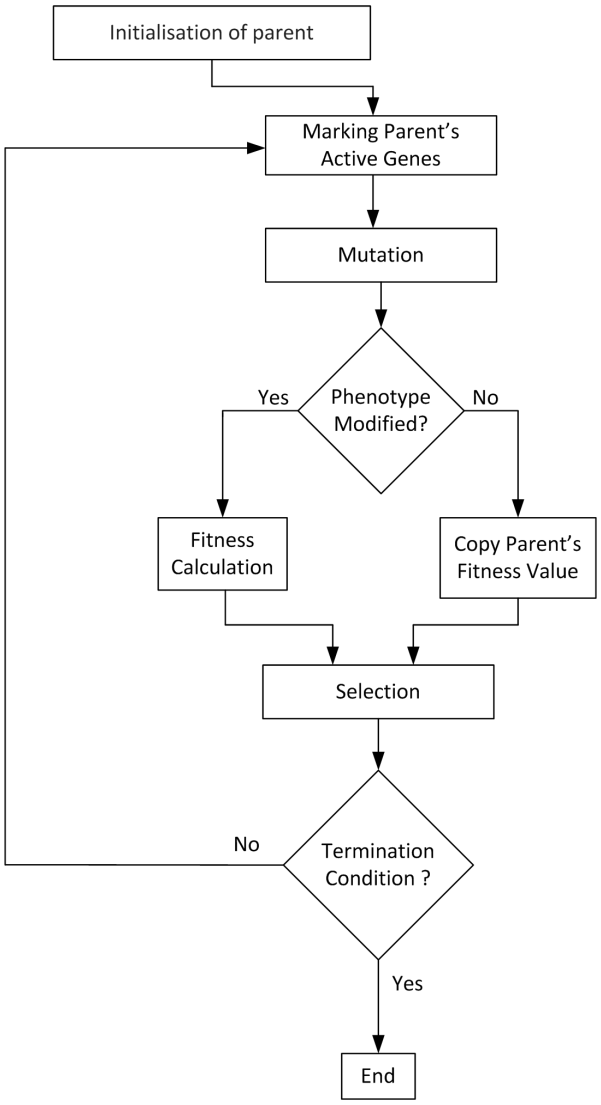


Figure 19. Design flow of proposed modified algorithm

In each generation, the designed algorithm detects active genes of all parent chromosomes (in the case that more than one parent is used; *CGP*, generally, uses one parent only). The mutation process performs changes in the chromosomes. According to the information about active genes, the mutation process can determine whether the phenotype has been modified or not. If this kind of chromosome modification occurs, the fitness calculation of a new individual has to be performed. If the mutation process modifies only inactive genes, the fitness value of the parent is assigned to the mutant (offspring). Note that in the first generation, the fitness calculation of the parent/s is necessary.

Further, it must be discussed whether the detection of active and inactive genes in the chromosome entails a burden. At first glance, it can seem that this detection represents an additional computing time. However, this detection is often part of the fitness calculation. Because it is profitable to evaluate only used cells (used cells are encoded by active genes).

If the fitness calculation uses this feature, the result of the detection is stored and can be used in other parts of the algorithm. For that reason, the detection of active genes does not represent a radical increasing demand factor of processing time.

Indeed, it is not necessary to search for active genes in every generation of evolution. In generations where the mutation process changes only inactive genes, the algorithm doesn't need to investigate any chromosome and can only copy information about active genes from the parent.

If it is assumed that T_p (time needed for production of population) and the overhead of the algorithm are marginal and they can be neglected; the time of algorithm run is given by the formula:

$$T_e \approx N_g . N_i . F_v . T_f \quad (8);$$

where F_v (this parameter has been established by the author of this thesis) means *Factor of Valuation* and takes the value range of (0; 1] and expresses the ratio between the number of individuals which must be evaluated and the total number of produced individuals. In the standard version of *CPG*, F_v is equal to 1; in the designed modified algorithm it can be less than one, this means the number of fitness calculations also can be reduced. The value of F_v depends on a few parameters of *CPG*. In the following experiments, the relationship between the value of F_v and the parameters of *CGP* is shown.

4.3.3 Experiments and Results

4.3.3.1 Experiments Description

Several experiments were performed to verify the proposed modifications of the algorithm. The author chose the design of multiplier circuits (2x2b, 3x2b, 4x3b) as benchmarks. All testing was performed by means of the software tool for the evolutionary design of a digital circuit which was introduced in the section 4.2. The modifications mentioned in the previous chapter have been implemented into this tool. In all tests, *CPG* used two-input cells with one output and four cell functions – AND, OR, XOR and identity (it copies the value from one input into an output directly; this function is represented by 'wire' in a real digital circuit). The mutation rate was variable in the value range from 1 to 7. It was considered that this range of mutation rate covers the optimal mutation rate for the benchmarks tasks used in the experimentation.

For individual tests, various *CGP* topologies *(n_c x n_r)* and maximal value of generations were used. To obtain relevant statistical results, each run of the algorithm was repeated 100 times in the case of multipliers 2x2b and 3x2b and 10 times for the multiplier 4x3b.

In the results of the test the value *Valuation Reduction [%] (VR [%])* is introduced. This coefficient expresses how much the number of fitness calculations (valuations) is reduced. For example, if *VR* takes the value of 25%, the algorithm executed by a quarter less fitness calculations than the standard version of *CGP*. The *VR* has an inverted relationship (*VR = 1 - F_v*) with respect to the *Factor of Valuation (F_v)*.

4.3.3.2 Experiments 1

This section deals with experimental results presented by Table 2, Table 3 and Table 4.

Table 2. Design of multiplier 3x2b (topology 5x5 cells)

Mutation Rate	Mean # Generations	Mean # Valuations	Mean # Individuals	VR [%]	Max. Fitness	Min. Fitness	Mean Fitness	Evolution Successful [%]
1	54,847	127,462	219,387	41.9	160	160	160.00	100.0
2	43,603	142,680	174,413	18.2	160	160	160.00	100.0
3	55,704	204,557	222,818	8.2	160	160	160.00	100.0
4	69,702	268,844	278,809	3.6	160	160	160.00	100.0
5	105,604	415,487	422,416	1.6	160	160	160.00	100.0
6	142,743	566,451	570,972	0.8	160	160	160.00	100.0
7	210,576	839,401	842,305	0.3	160	158	159.82	88.0

The first of these tables (Table 2) describes the results obtained for the evolutionary design of the multiplier 3x2b. The topology of 5x5 cells of *CGP* was used; the maximal number of generations was set to 500,000. The evolution design searched only for the full functionality of the multiplier; it follows that the maximal fitness value is 160. The table shows the mean number of generations which were executed by the algorithm. Further, it presents the mean number of individuals produced in algorithm by the mutation process. The two key items in the tables are the mean number of valuations and *VR [%]* (see previous section). The standard version of *CPG* performs the same number of fitness calculations as in the number of produced individuals (*Mean # Individuals*). The column *Mean # Valuations* shows the number of executions of fitness function in the modified algorithm. It can be observed that for a mutation rate equal to 1 the reduction of valuations is over 40%. The *VR* decreases according to the growing value of the mutation rate.

From the presented values it is obvious that the evolution with a mutation rate equal to 2 needs the lowest number of generations but the variant with a mutation rate set to 1 executes the minimum number of valuations. For the proposed formula (8) a mutation rate equal to 1 appears to be optimal.

The remaining columns of the tables show the maximum, minimum and mean values that were obtained during the evolution runs. The last column *Evolution Successful* indicates how many evolution runs reached the maximal fitness value (i.e., full logic functionality was found).

Table 3 shows results of the similar experiments. However, the design of a multiplier of 2x2b was used. In contrast to the previous table, this table does not contain some columns expressing the successfulness of the evolutions runs. The reason for this is simple; all runs of the experiments were successful. The maximum value of fitness (representing full functionality) was 64 for this benchmark. Each run of evolution performed 250,000 generations at the most.

Table 3. Design of multiplier 2x2b (topology 5x5 cells)

Mutation Rate	Mean # Generations	Mean # Valuations	Mean # Individuals	Valuation Reduction [%]
1	7,068	12,521	28,275	55.7
2	3,871	10,425	15,485	32.7
3	3,205	10,255	12,820	20.0
4	3,499	12,268	13,995	12.3
5	2,942	10,887	11,768	7.5
6	3,759	14,346	15,038	4.6
7	4,639	17,994	18,557	3.0

As in the previous case, an analysis of the obtained results for the multiplier design can be made. From the point of view of the mean number of generations, a mutation rate equal to 5 provides the best results (i.e., the quickest evolution). However, from the point of view of the mean number of valuations, the variation with the mutation rate that takes the value of 3 is better. Indeed, it is necessary to note that the number of generations and valuations are very balanced, and are independent of the value of the maturation rate. Despite this fact, it can be observed that there is a meaningful benefit in the term of the mean number of valuations. The most efficient configuration with a mutation rate equal to 3 reduces the

number of valuations by an average of 20 percent in comparison with the standard version of *CGP* with the same value of mutation rate.

The described experiments were repeated with other settings of topology $(n_c \times n_r)$. The additional design of the multiplier 4x3b was also performed; only the topology 1x100 was tested for this design. The fitness maximum of this benchmark is 896 (= 7 x 2^7); the limit of generations was set to 100 million.

Table 4 summarises the results of all benchmarks for the various topologies. It shows only the cases with the optimal value of the mutation rate.

The item *VR to Min [%] (Valuation Reduction to Minimum)* represents the benefit of the modified algorithm in contrast to the standard *CPG* with the optimal value of mutation rate. The meaning of this can be described by the means of Table 3. The configuration with the mutation rate equal to 3 is optimal from the point of view of the number of valuations. The modified process of valuations yields approximately the reduction of 20% for the given value of the mutation rate. If the standard version of *CGP* is assumed, a mutation rate equal to 5 ensures the best (fastest) runs of evolution. The comparison between these two situations is represented by the value *VR to Min*. In other words – in the case of Table 3 – the algorithm modification yields a reduction of roughly 13% against the standard concept of *CGP* when a range [1; 7] of mutation rate is supposed. If the item *VR to Min* doesn't contain a value, it means that the optimal behaviour for both versions of *CPG* is reached with the same value of mutation rate.

From the results in Table 4, the reader can see that – across various topologies of *CGP* – the minimum reduction of the mean number of fitness calculations (valuations) is 12.9%. Indeed, the optimal value of mutation rate is henceforward very significant.

Table 4. Summary of experiments

Circuit of Multiplier	Topology	Mutation Rate	Mean # Valuations	Valuation Reduction [%]	VR to Min [%]
2x2b	5x5	3	10,255	20.0	12.9
2x2b	1x10	1	25,236	22.2	-
2x2b	10x10	7	6,035	19.4	-
3x2b	5x5	1	127,462	41.9	26.9
3x2b	1x20	1	148,918	30.4	-
3x2b	10x10	5	43,701	23.3	20.7
4x3b (1)	1x100	2	7,656,913	17.1	-

Note to table:

(1) Only 10 runs of algorithm were performed.

4.3.4 Analysis of Reduction

Referring to formula (8) in chapter 4.3.2, and from the experimental results, a more detailed analysis may be made regarding the dependence of the evolution time. The moment which affects the number of fitness calculations occurs after mutation, the individuals with a modified phenotype undergo a fitness calculation, and the other individuals do not. The probability determining (it will be denoted as P_{mig} – *Probability of mutation of inactive genes*) whether phenotype of individual will not be changed can be expressed by:

$$P_{mig} = \left(\frac{G_{iact}}{\Lambda}\right)^{\mu_g} \quad (9);$$

where G_{iact} ..... *the number of inactive genes,*

Λ ..... *the total length of the chromosome (see chapter 4.1),*

μ_g ..... *the mutation rate (the number of genes mutated within the mutation process).*

It follows that the probability of complementary event – *Probability of mutation of active genes* (P_{mag}):

$$P_{mag} = 1 - P_{mig} \quad (10).$$

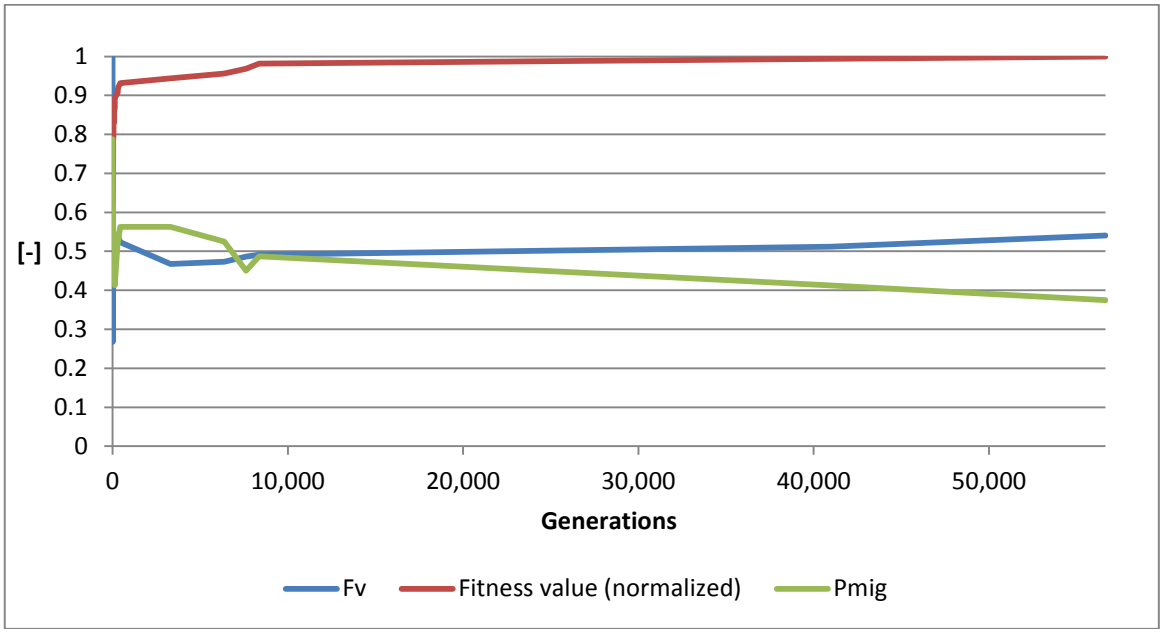


Figure 20. Progression of evolution run (mut. rate = 1)

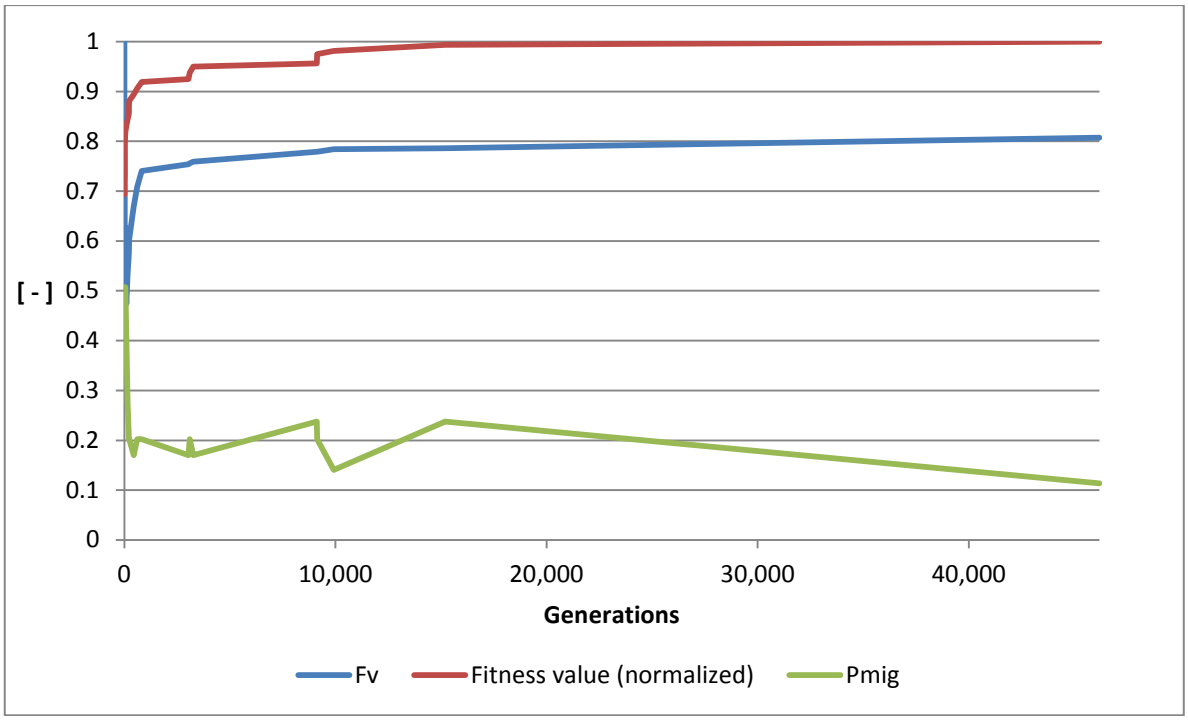


Figure 21. Progression of evolution run (mut. rate = 2)

In other words, the greater the difference between the numbers of used cells (genes) and the total number of cells (genes; chromosome length) the greater the probability that an individual does not need to be evaluated. Indeed, this individual cannot yield any benefit in terms of the fitness value; however, the genotype is changing.

The progressions of the two evolution runs (design of multiplier 3x2; 5x5 topology; maximal value of *l-back*) are shown in the Figure 20 and the Figure 21. The first run was performed with the mutation rate equal to 1, the latter with the mutation rate equal to 2. These two runs were chosen randomly for demonstration purposes. Other evolution runs can proceed differently.

In both runs of the algorithm, the number of active cells was low – 4 and 8; this matches 17 and 29 active genes respectively. During the evolution runs, the number of active genes finally grew up to 50 and 53 active genes. The total length of the chromosome is 80 genes, in the results of which the value of P_{mig} decreased. It can be seen that P_{mig} is lower in the run with the higher value of mut. rate – see formula (9). The progression of the value F_v shows the current reduction of fitness calculation. The behaviour of F_v aids to explain the results in Table 2. Whilst the first run takes more algorithmic generations, it is economical from the point of view of fitness calculations. The higher value of mutation rate also means a higher probability of change of active genes and thus the fitness calculation. In the

presented two runs, the following values of F_v were reached: 0.54 for mut. rate equal to 1 and 0.81 for the mut. rate equal to 2.

The value of F_v is directly related to the value of P_{mag}. However, it is valid only for mean values. If the mean values (weighted average) of parameter P_{mag} are calculated, respective values of values 0.54 and 0.85 are obtained. The parameter F_v (established in section 4.3.2) can be approximately expressed as:

$$F_v \approx Mean\ Value\ (P_{mag}) \quad (11).$$

Note: Mean Value () expresses calculation of the weighted average over the whole evolution run.

4.3.4.1 Experiments 2

In the previous experiments, only the finding of full functionality was performed. The minimal solution from the point of view of the used cells/gates is often desired in the evolutionary design domain.

The subsequent experiments executed the evolution of the 3x2b multiplier with a topology of 1x20 using a maximal value of *l-back* parameter and a limit of 1,000,000 generations. In additional, the algorithm searches for full functionality of the circuit and such a solution which, at most, is composed from 13 cells (see formula (3)).The results of the experiments are shown in Table 5.

Table 5. Design of multiplier 3x2b (topology 1x20 cells) – 7 cells maximal

Mutation Rate	Mean # Generations	Mean # Valuations	Valuation Reduction [%]	Evolution Successful [%]
1	385,976	1,127,302	26.98	71.0
2	175,488	647,443	7.77	95.0
3	164,750	642,683	2.48	97.0
4	281,979	1,119,463	0.75	92.0
5	384,035	1,532,196	0.26	84.0
6	487,082	1,946,490	0.09	82.0
7	720,415	2,880,476	0.04	58.0

The best behaviour shows that the mutation rate is equal to 3. Unfortunately, in this configuration the modified algorithm does not bring any significant benefit. The topology 1x20 was chosen purposely. It is expected that the value P_{mig} will be low; thereby F_v will be greater.

Table 6 illustrates the progression of one run (worse than average) of evolution with a mut. rate equal to 3. The value of P_{mig} decreases very quickly. It can be seen that the last four rows describe the optimization of circuit size (the circuit already has reached full functionality). If focus is given to the last two rows, it is clear that the pass from 14 cells to a circuit with 13 active cells took a very long time – roughly 250,000 generations. During this searching, the value of P_{mig} is minor; because of this, a great number of fitness calculations is performed. When optimization of the circuit size is performed this behaviour is usual.

Table 6. Example of evolutionary progression of design of multiplier 3x2b

Generation	Valuation	Fitness Value	Used Cells	P_{mig}	F_v
1	6	101	8	0.170	0.67
2	10	103	9	0.131	0.77
4	18	111	9	0.131	0.86
7	28	119	9	0.131	0.85
11	43	121	10	0.098	0.88
16	60	125	11	0.072	0.87
23	85	133	6	0.270	0.88
40	136	137	9	0.131	0.82
317	1,143	139	11	0.072	0.90
679	2,495	141	12	0.050	0.92
751	2,761	143	10	0.098	0.92
815	2,996	145	12	0.050	0.92
1,313	4,922	147	15	0.012	0.94
2,764	10,530	149	12	0.050	0.95
3,049	11,607	151	12	0.050	0.95
3,870	14,651	153	13	0.034	0.95
12,002	46,213	155	12	0.050	0.96
14,147	54,289	157	12	0.050	0.96
21,549	82,422	158	14	0.021	0.96
132,666	516,875	159	16	0.006	0.97
132,744	517,181	164	16	0.006	0.97
132,923	517,894	165	15	0.012	0.97
133,656	520,785	166	14	0.021	0.97
387,205	1,513,339	167	13	0.034	0.98

4.3.5 Compact Version of Cartesian Genetic Programming

Because of the results of tests obtained in previous sections, it may lead to change of understanding of the whole conception of *CGP*. Assume the *(1+1)* version of *CGP*. The process of algorithm can be very simplified because only few simple operations are performed. If the modifications introduced in previous sections are inserted, it can result in slightly different flow diagram of the algorithm – the working title '*Compact Cartesian Genetic Programming*' with abbreviation *CCGP* was established. The diagram of this algorithm is shown in the Figure 22.

The initialisation of the parent (in this case, there is the only parent) is performed in the usual way; it means that the parent can be initialized randomly or by some seed. During the calculation of the fitness function of the initialized parent, the algorithm detects active genes in the genotype – in other words, the genes forming the phenotype are marked. After this evaluation and the marking of the genes, the mutation is performed.

The number of mutated genes depends on the mutation rate value, similarly to the standard version of *CGP*. However, the algorithm can detect the mutation of genes forming parent phenotype in this phase. In other words, there is a list of active genes generated during fitness calculation of the current parent. Because of this list, the mutation process can immediately determine whether active genes of the chromosome will be modified or not. Of course, the mutation of active gene itself doesn't mean that the phenotype of the individual was changed. Specific situations can occur when a new value of the gene is equal to a previous value, or when the next gene mutation neutralizes preceding modification in chromosome. The algorithm may implement processes that check these situations and avoid them. In this moment – for simplicity – these protections are not taken into account. The next steps of algorithm are derived from the result of the mutation process. If active genes (phenotype) of parent were changed, it is necessary to evaluate the newly formed individual. If only inactive gene/s was/were mutated, the current parent is replaced by a mutant (result of last mutation process) and this individual undergoes the next cycle of mutation. If the mutation produces an individual with new phenotype, the fitness calculation is performed for one; in this phase, the algorithm also detects active genes of an individual. If the obtained fitness value of the evaluated individual is better than or equal to the fitness found until now, the individual takes a position of parent for the next generation cycle and 'the list of active genes' is updated according to this individual. If the obtained fitness value is worse, the formed mutant is dropped and the next generation

continues with the current parent. However, the test of terminal conditions is necessary before the next generation cycle when better fitness was achieved.

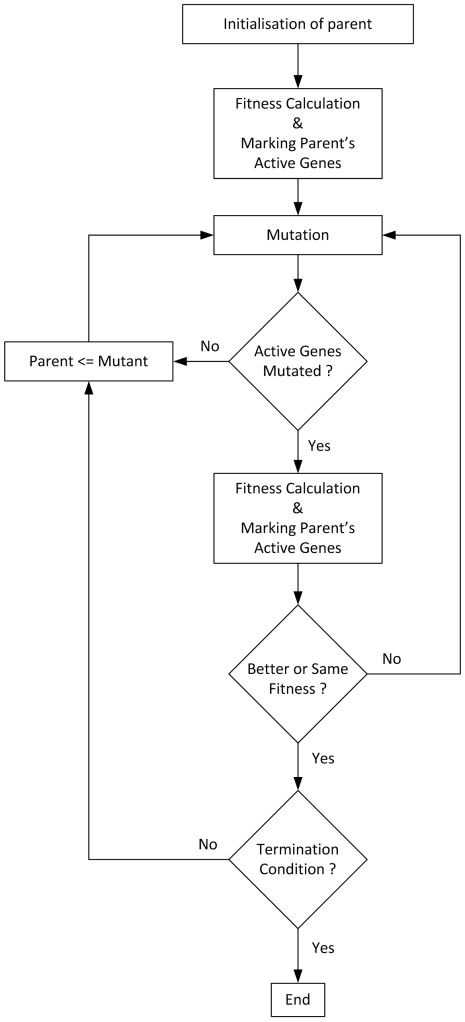


Figure 22. Design flow of the *CCGP*

At the first glance, the presented algorithm is simpler from the point of view of the implementation, memory requirements, etc. On the contrary, the algorithm loses ability of diversity which is essential in the Evolutionary algorithms domain. Only the wide research of one may indicate possible benefits. The next section of this work shows several experiments and tests with this designed algorithm.

4.3.5.1 Testing of CCGP

For basic testing of the *CCGP*, the benchmarks introduced in section 4.3.3 were used. Recall that the tests from this section performed search algorithm with $\lambda = 4$; it means *(1 + 4)* type of algorithm. This value of λ is used very often. For that reason, mainly algorithm runs with this value of λ will represent fundamental subjects to comparison.

First, the test multipliers 3x2 and 2x2 with the topology of 5x5 cells were performed. The terminal condition of maximum number of generation cycles was increased to 2 million generations in the case of the multiplier 3x2 and to 1 million generations for multiplier 2x2, because the λ value is equal to 1. By this modification of the terminal condition is ensured that both version of algorithm can generate the same number of individuals (by means of mutation process).

The test results of these two benchmarks are shown in Table 7 and Table 8. The former table describes test for multiplier 3x2. It is clear that the best results were obtained with the mutation rate equal to 1; roughly 166 thousand individuals were generated in average and approximately for 40 percent of them the fitness calculation was performed. Now, the focus should be given on Table 2 describing the same benchmark for $\lambda = 4$. If the results presented by this table are compared with the results obtained for *CCGP*, it is clear that *CCGP* needs markedly more generations. However, this behaviour was expected, because *CCGP* generates only one mutant within one generation. The mean number of generated individuals is the same order of magnitude for both algorithm variants. Even the *VR* is almost identical. Regardless of these facts, the *CCGP* with mutation rates equal to 1 and 2 performs less evaluations than the *CGP* with $\lambda = 4$ with optimal mutation rate. *CCGP* required about 24 percent fewer evaluations.

The Table 8 describes the same benchmark for the multiplier 2x2b. This table has to be compared with Table 3. It can be seen that *CCGP* provides 'better' results again. In comparison with the multiplier 3x2b, the mean numbers of generations and generated individuals don't increase constantly; however, the *CCGP* with mutation rate equal to 1 performs about 39% less fitness calculations.

Table 7. Test of CCGP – Design of multiplier 3x2b (topology 5x5 cells)

Mutation Rate	Mean # Generations	Mean # Valuations	Mean # Individuals	VR [%]	Max. Fitness	Min. Fitness	Mean Fitness	Evolution Successful [%]
1	166,303	96,836	166,304	41.8	160	160	160.00	100.0
2	137,033	111,922	137,034	18.3	160	160	160.00	100.0
3	179,654	165,451	179,655	7.9	160	160	160.00	100.0
4	222,244	214,287	222,245	3.6	160	160	160.00	100.0
5	356,551	350,685	356,552	1.6	160	160	160.00	100.0
6	602,020	597,857	602,021	0.7	160	158	159.98	99.0
7	867,326	864,265	867,327	0.4	160	158	159.87	90.0

Table 8. Test of CCGP – Design of multiplier 2x2b (topology 5x5 cells)

Mutation Rate	Mean # Generations	Mean # Valuations	Mean # Individuals	VR [%]	Max. Fitness	Min. Fitness	Mean Fitness	Evolution Successful [%]
1	14,473	6,206	14,474	57.1	64	64	64.00	100.0
2	12,146	8,111	12,147	33.2	64	64	64.00	100.0
3	10,513	8,501	10,514	19.1	64	64	64.00	100.0
4	13,284	11,725	13,285	11.7	64	64	64.00	100.0
5	13,271	12,306	13,272	7.3	64	64	64.00	100.0
6	13,566	12,942	13,567	4.6	64	64	64.00	100.0
7	13,917	13,540	13,918	2.7	64	64	64.00	100.0

The presented tests of two benchmarks show promising results. For greater relevance, the next tests/experiments were carried out and are summarized in Table 9 (table shows only the mean number of valuations, the number of generations is not presented for better lucidity). The experiments were extended to other values of λ. The experimental designs were focused only on searching for full functionality of design circuits (a minimal solution isn't required). Note that each run of evolution was repeated 100 times in the case of multipliers 2x2b and 3x2b and 10 times for the multiplier 4x3b. All runs of design multiplier 4x3b were limited by 100 million of generations.

Table 9. Test of CCGP – Design of multipliers with variable number of mutants

			λ = 1 / CCGP		λ = 4		λ = 10		λ = 14		CCGP Mean Red. [%]
Circuit	***l-back***	n_c x n_r	***Mut. Rate***	***Mean # Valuations***	***Mut. Rate***	***Mean # Valuations***	***Mut. Rate***	***Mean # Valuations***	***Mut. Rate***	***Mean # Valuations***	
Mult. 2x2b	max.	5x5	1	6,206	3	10,255	4	14,592	2	16,975	39.5
Mult. 2x2b	1	5x5	3	11,861	1	16,078	2	25,458	4	29,506	26.2
Mult. 2x2b	max.	1x10	1	20,611	1	25,236	2	35,730	2	37,893	18.3
Mult. 2x2b	max.	10x10	1	3,630	7	6,035	6	9,178	4	11,611	39.8
Mult. 3x2b	max.	5x5	1	96,836	1	127,462	2	203,573	2	219,732	24.0
Mult. 3x2b	1	5x5	1	159,671	1	208,729	2	280,543	2	347,049	23.5
Mult. 3x2b	max.	1x20	1	124,069	1	148,918	2	206,369	2	245,894	16.7
Mult. 3x2b	max.	10x10	3	26,248	5	43,701	7	55,779	7	74,004	39.9
Mult. 4x3b	max.	1x100	1	5,057,139	2	7,656,913	3	21,516,024	3	19,460,123	34.0

If obtained results are analysed, an interesting behaviour can be observed. In all cases, *CCGP* was more effective (on the average) in terms of fitness calculations than classical

CGP generating 4, 10 or 14 mutants in one generation. Almost in all cases, the number of fitness calculations needed for evolution run increases linearly with the increasing value of λ. The item called *CCGP Mean Reduction [%]* expresses the reduction of fitness calculations performed by *CCGP* compared with the 'best situation' of the algorithm with λ greater than 1.

Indeed, the great number of mutants produced in each generation causes the decreasing of the number of performed generations. This trend is readable from the Figure 23 which shows the mean number of generations in multiplier 3x2b benchmark. The growth of the number of generations when λ is equal to 1 is markedly. However, it has to be taken into account that only one mutant is produced in this case (compared with $\lambda=14$ => 14 times more mutants are generated within one generation).

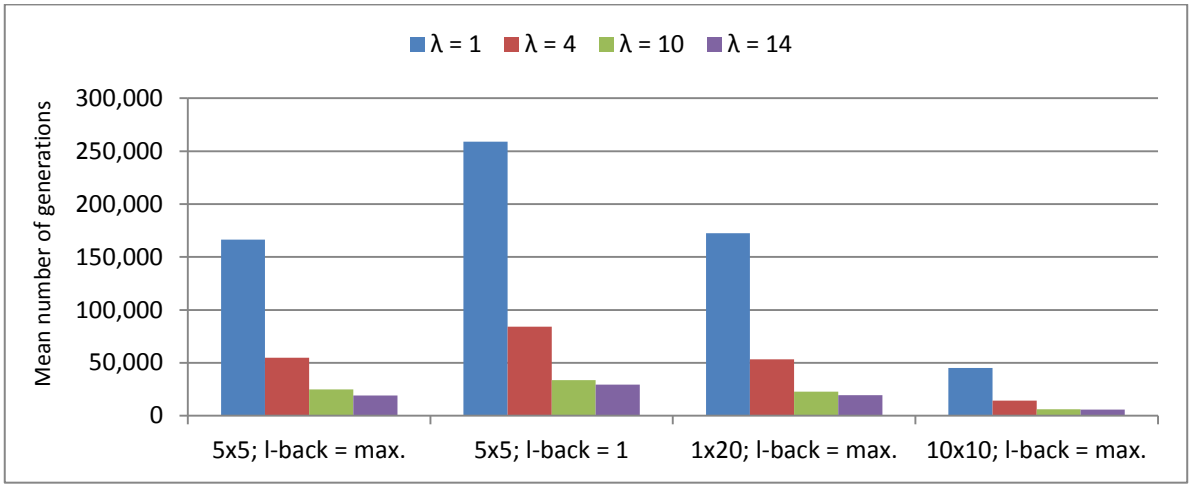


Figure 23. Trend of number of generation with variable λ (multiplier 3x2b)

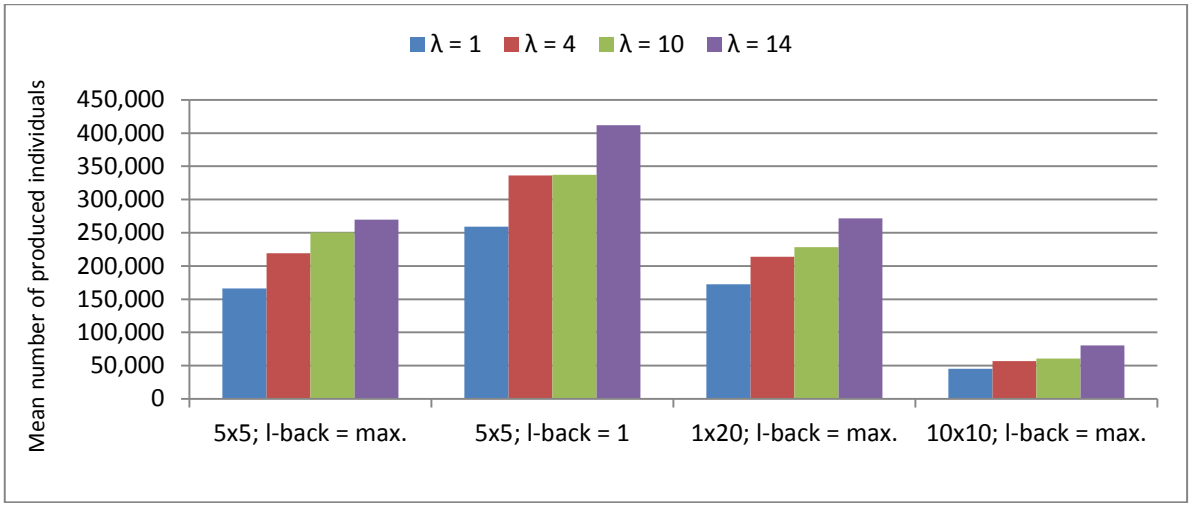


Figure 24. Number of produced individuals (multiplier 3x2b)

The Figure 24 shows the mean number of produced individuals. From this chart is clear that *CCGP* generates least individuals (mutants). Because of introduced modifications of

CGP, some mutants do not need evaluation. By this, it follows the number of fitness calculations described in Table 9. This table also tells us that the minimal benefit of the *CCGP* is approximately 16 percent in case of multiplier 3x2b with topology 1x20. In other cases, the benefit is more significant.

4.3.5.2 Analysis of CCGP

The result in previous section shows that the version of the *CGP* with λ can be very effective. Now, this kind of algorithm will be elaborated.

As mentioned earlier, the *CGP* with search algorithm *(1 + 1)* works with one parent and produces one mutant during generation cycle. It means that the algorithm doesn't form population of candidate solutions and its behaviour approaches certain kind of random searching or Hill-climbing algorithm. The parent in *CCGP* travels in the search space and the fitness calculation is executed only when the phenotype is changed. It is thus questionable, whether *CCGP* is still an evolutionary algorithm in the true sense of the word.

It is necessary to note that tests and experiments introduced in previous sections will focus on the number of executions of fitness calculation and it is not ensured that the designed kind of *CGP* will be effective for other types of tasks.

In the end, the T_e for *CCGP* will be derived by means of formulas mentioned in section 4.3.4. The first, the mean value of P_{mag}, will be expressed by the formula:

$$P_{mag_{av}} = \frac{\sum_{i=1}^{N_g} P_{mag_i}}{N_g} \quad (12);$$

where $P_{mag_{av}}$ ..... *the mean value of Probability of mutation of active genes,*

P_{mag_i} ..... *the value of* P_{mag} *valid for given generation,*

i. ..... *index of evolutionary generation,* $i \in [1; N_g]$.

Further, in the section 4.3.4 it was derived $F_v \approx Mean\ Value\ (P_{mag})$ applies. If the formula (12) is substituted into formula (8) and it is assumed that N_i is equal to 1, it follows:

$$T_e \approx \sum_{i=1}^{N_g} P_{mag_i} \cdot T_f \quad (13).$$

Finally, the P_{mag} can be expressed by means of complementary event P_{mig} – see formula (9). After substitution applies:

$$T_e \approx \sum_{i=1}^{N_g} \left(\frac{\Lambda^{\mu_g} - {G_{iact_i}}^{\mu_g}}{\Lambda^{\mu_g}} \right) \cdot T_f \qquad (14);$$

where G_{iact_i} ..... *the value of* G_{iact} *valid for given generation.*

These formulas partly explain the behaviour and relations of the *CCGP* in terms of its reduction of performed fitness calculations (evaluations). Referring to formula (14) is clear that the value of N_g remains unpredictable. For that reason, a lot of experiments will still be required. However, the techniques introduced in this chapter can accelerate the evolutionary design process very significantly.

4.4 Implementation

This subchapter discusses the implementation of *CGP* by an FPGA device in great detail. It is divided into several subsections. Each of them deals with individual component of the implementation.

4.4.1 Fast Detection of Active Genes

The previous sections show that the detection of active genes in a *CGP* chromosome could yield certain benefits. The process of detection expresses which genes form a phenotype. Only the active cells (represented by active genes) must be considered to get the output response of a *CGP* structure and this feature can accelerate the calculation of the fitness function. In the evolutionary design domain, if the active genes are known, the number of used cells in the designed digital circuit can be derived. Hence, this is a very significant and frequently used feature.

As mentioned above, the general algorithm for detecting active nodes/cells (active genes are determined by active cells) in a *CGP* structure was published in [53]. This general approach can be used by implementations of a software performing *CGP*. It is sequentially defined and, for that reason, it can be implemented, for instance, by a C code very easily. If the implementation in FPGA/ASIC is proposed, this sequence approach of active genes detection is not profitable. Indeed, the detection can be performed by a state machine and can run similarly to the C code implementation. In the case of need, the detection can be implemented by a soft-core processor in the FPGA design. Unfortunately, these mentioned possibilities of the active genes detection are relatively very slow. No implementation of fast detection has been published until these days (when the author wrote this text). Thus, the method of the fast hardware detection of active genes in the *CGP* chromosome has been developed by the author.

The following few obvious requirements for the proposed fast hardware detection were established:

- Determination of active genes in a chromosome
- Derivation of the number of used cells/nodes
- High-speed processing
- Low latency of the output
- Assumed use of the *l-back* parameter taking the value of 1 or 2

- Modular structure

Because of the requirement on high-speed processing of a *CGP* chromosome, the author hasn't put main emphasis on the amount of hardware resources needed for the implementation of the detector.

4.4.1.1 Principle of Detector

In this section, the keynote of the proposed detector will be introduced.

The following figure shows a simple *CGP* structure configured by a certain chromosome. There are three primary inputs and two primary outputs.

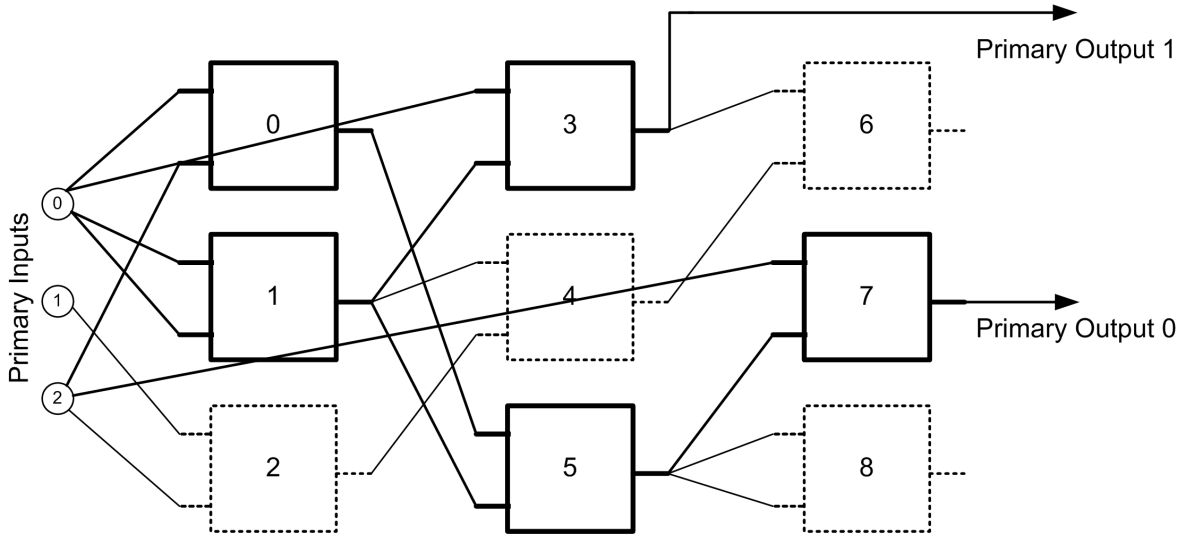


Figure 25. Example of *CGP* structure

It is simply possible to determine the used cells from the figure. The *Primary Output 0* depends on cells 0, 1, 5 and 7. The latter primary output – *Primary output 1* – exploits the cells 1 and 3. By the means of simple reasoning, two basis rules can be inferred:

1) The cell is active/used if it feeds primary output.
2) The cell is active/used if it feeds active/used cell.

The principle of the designed detector is based on these two rules and the propagation/generation of the state describing active cells in the direction from primary outputs to primary inputs. The state is propagated and generated 'column-by-column' starting with the rightmost column. The principle of this method is demonstrated by means of the Figure 26, describing the detection of used cells related to the *Primary Output 0*. There are three tables belonging to particular columns in the *CGP* structure. The item called *AC (Active Cell)* indicates the usage ('1' => used, '0' => unused) of a cell. The

second item – *RC (Route Carry)* – defines cells in the following (direction is shown by arrow) column feeding cell in the current column. First of all, the output node (cell) is determined. In our case, the *Primary Output 0* is connected to the output of the cell no. 7; hence this cell is marked as used. One input of the cell no. 7 is fed by the cell no. 5 and for that reason there is *RC* = '1' in the item on the bottom position. The second input of the cell no. 7 is connected to one of the *Primary Inputs*; this input value does not depend on any cell. In other words, the state describing active cells is carried (propagated) only to the cell no. 5 in the next column. This cell is only used in the middle column of the structure. The inputs of this cell are connected to the cells no. 0 and no. 1 in the first column. It corresponds to the vector "1 1 0" in the *RC* item. This step marks active cells in the last (the first within the meaning of the *CGP* structure) column.

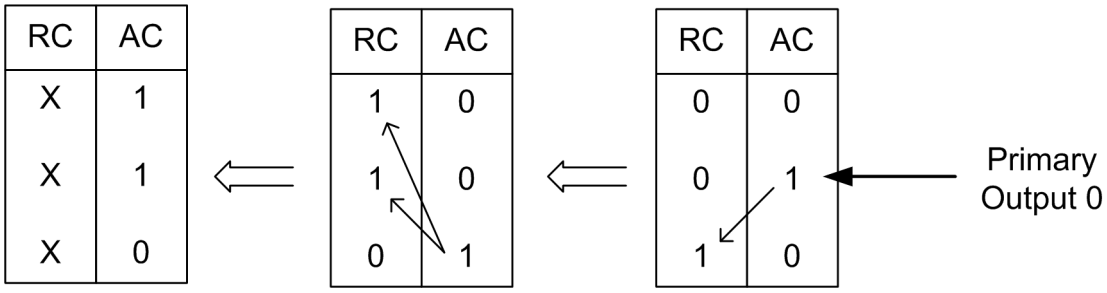


Figure 26. Principle of used cells detection – Primary Output 0

The detection of the used/active cells related to the *Primary Output 1* is performed analogously. This primary output is drawn from the cell no. 3; thus this cell and the cell no. 1 define an output response of the *Primary Output 1*.

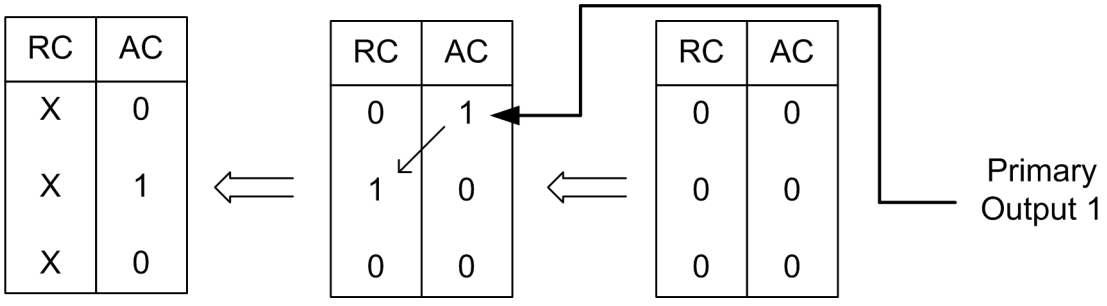


Figure 27. Principle of used cells detection – Primary Output 1

The Figure 26 and Figure 27 show the detection of used cells related to each primary output separately. This feature is useful if it is necessary to know which cells define the behaviour of particular outputs. However, the total count of used cells is demanded more often. In this case, the detection for all primary outputs can be performed at the same

process as shown in the Figure 28. The total list of used cells is given by the merger between the used cells related to particular primary outputs.

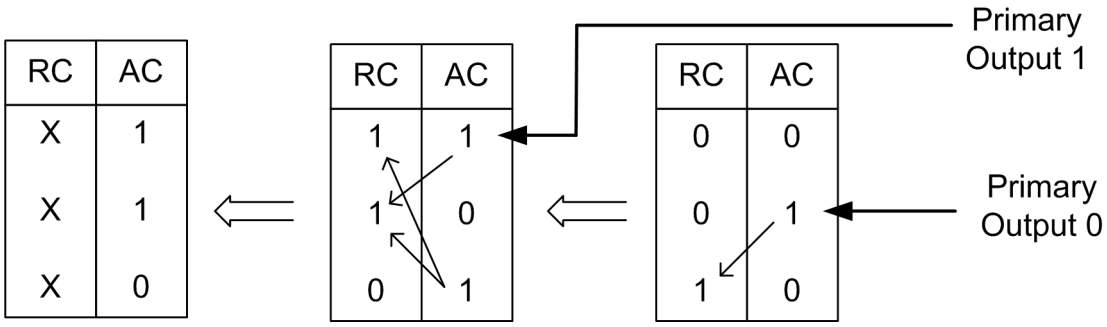


Figure 28. Principle of used cells detection – Primary Outputs

4.4.1.2 Implementation

The principle introduced in the previous section can be represented by a logic combinational circuit. The complexity and the area of this circuit depend on the defined parameters of a *CGP* structure. However, the designed implementation of the detector described in this section is highly modular, composed of several structural elements. This concept makes it possible to compose the detector of active genes for various parameters (the *l-back* parameter still being limited) of the structure. Of course, the parameters of a *CGP* structure affect the area and hardware resources needed to implement the detector. Thereby timing parameters (max. frequency, data delay, etc.) are also affected.

Note: This section mainly focuses on the situation when CGP l-back parameter is equal to 1.

The basic structural element is called a *Detection Cell.* To each cell of a *CGP* structure belongs just one *Detection Cell.* This element determines whether a corresponding cell of *CGP* is active/used or not. The second functionality of a *Detection Cell* observes the *connection genes* (configuration information defining nodes connected to cell inputs) of cell and defines the propagation of a signal indicating an active cell for the next column. In essence, if focus is given to the Figure 28, a *Detection Cell* produces logic values for one row in the table, one *AC* and *RC* value.

The implementation used in an FPGA device uses slightly modified index of nodes within the structure. The relativity addressing nodes is used. If the rightmost column of the structure depicted in the Figure 25 is assumed, valid addresses are in the range [0; 5]. The values vary from 0 up to 2 address nodes of primary inputs. The rest of address range expresses the outputs of cells 3, 4 and 5.

The rules for the detection of active cell have been declared above and are easy to understand. Note that in the general version of *CGP*, a primary output can be connected to any node/cell. If this 'output connectivity' is limited so that the cell cannot feed primary output, only one rule is valid. The cell is active only when it feeds another active cell.

The diagram of the *Detection Cell* is shown in the Figure 29. Note that the diagram explains the behaviour of the designed component; the final HDL implementation can be different. At first, the logic driving signal (port) *act_cell_flag* will be described. The current value of this port is given by the logic sum of input signals *prev_act_cell* and *output_sel*. The former signal indicates that the current cell feeds active cell/s. The latter signal tells us that this cell feeds the primary output. An output value of this OR-gate determines whether the related *CGP* cell is used. The source of the *output_sel* signal is described hereafter.

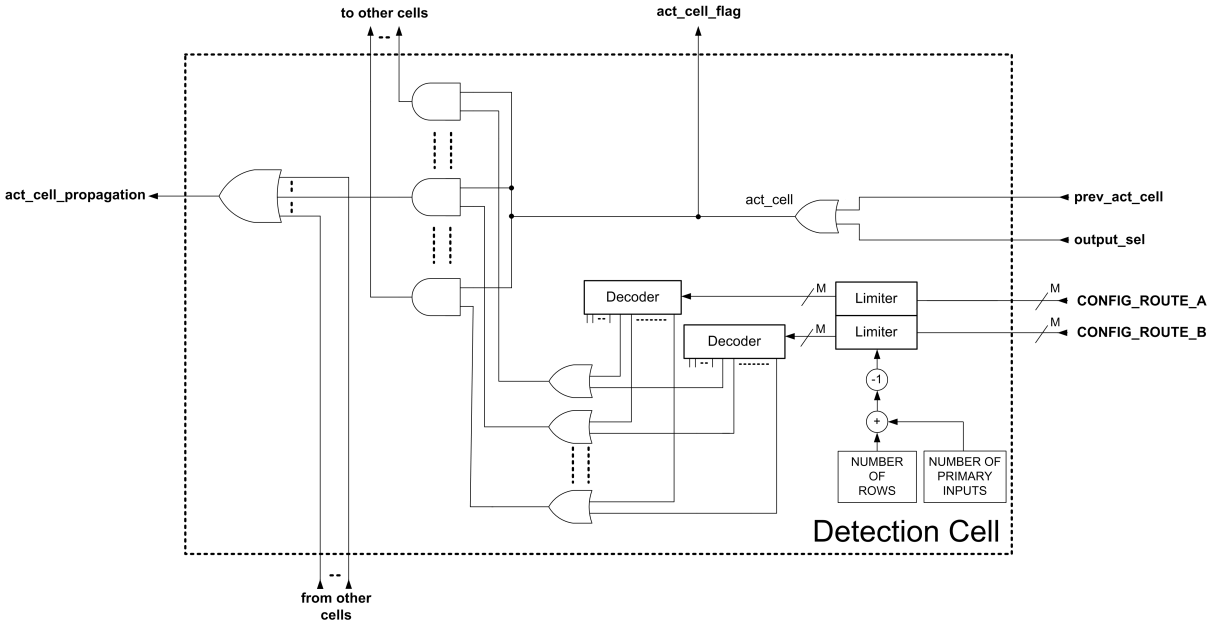


Figure 29. Diagram of the Detection Cell

Note to the Figure 29: The 'M' denotes the width (in bits) of route configuration (connection gene). Other signals are single-bit.

The next part of the detection cell determines the value of the *act_cell_propagation* signal. This signal informs the cell in the next column that some active cell is fed by it. At the beginning, the route configuration vectors of the cell represented by *CONFIG_ROUTE_A/B* are limited to the maximum meaningful value (= the number of rows + the number of primary inputs – 1). Such modified values input to a pair of binary decoders. These decoders produce logic high on just one of its outputs. The outputs

implying the primary inputs – it means the value of *CONFIG_ROUTE_A/B* is less than the number of primary inputs – are unused/unconnected. The other outputs of decoders control the propagation of the *act_cell* signal which is routed to appropriate cell/s.

For example, assume the *CGP* structure 3x3 cells (see the Figure 25) and the *l-back* parameter equal to 1. Place emphasis on the middle cell in the last column; this cell feeds the primary output and is fed by the primary input and the bottom cell in the previous column. One of the decoders produces logic high on unused output and can be ignored. The second decoder produces logic high on the third of the used outputs. By the means of the AND gate, the signal *act_cell* is routed to the bottom detection cell (in the current column). This bottom detection cell drives the *act_cell_propagation* signal to high via the output OR gate.

The *act_cell_propagation* signal can be set if one or more detection cells route *act_cell* into the current cell. Indeed, if the *CONFIG_ROUTE_A/B* points on a cell in the same row, the signal *act_cell_propagation* is driven by the signal *act_cell* generated in the same detection cell. Each detection cell provides the signals that are connected to all other detection cells (in the column) and can drive the *act_cell_propagation* signal to high in these cells. The number of these signals is given by the number of other cells in the column. Also each detection cell accepts the same number of signals from the other cells; these signals represent the inputs of the 'output OR gate' and can drive the *act_cell_propagation* signal of the current cell to high.

The group of detection cells forms a *Column Detector*. It interconnects individual detection cells and provides the needed route configuration for cells detection. It also derives the number of used cells in a column. The Figure 30 describes the *Column Detector* that can be used for columns of a *CGP* structure except the first one. According to this diagram, it is clear that the column detector represents almost only 'glue logic'. It implements only conversion of flags (vector *act_cell_flag*) indicating active cells in the column to an information numeral. This functionality can be ensured by the encoder or by *N* one-bit adders. Of course, the need of the number of active/used cells determination in single column may be debatable. The common applications use only the total number. However, this functionality contributes to the analysis of candidate circuits in large.

It is necessary to note that for higher numbers of cells in a column, the implemented interconnection among fundamental detection cells can be demanding of an FPGA's area and resources.

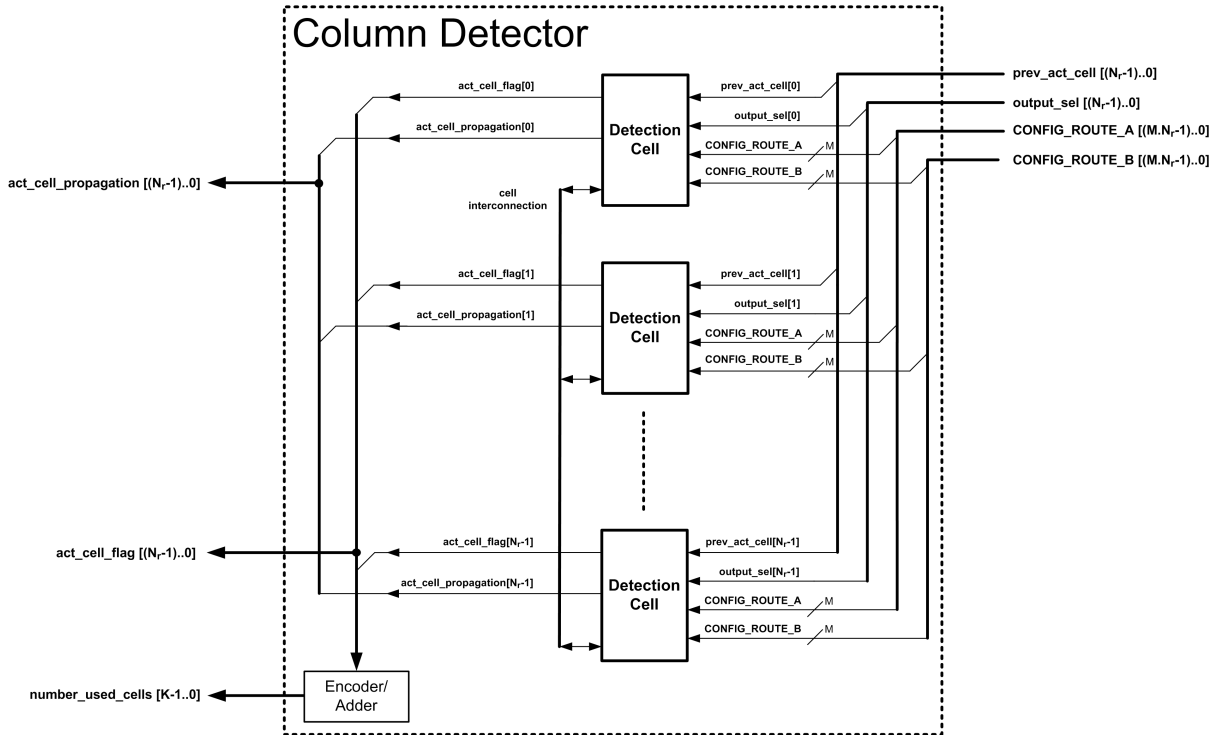


Figure 30. Diagram of Column Detector

Note to the Figure 30: The character 'M' denotes the width (in bits) of route configuration; the 'N_r', the number of cells in a column. The 'K' means the width (in bits) of the vector expressing the number of used cells.

As mentioned above, the column detector depicted in the Figure 30 doesn't have to be used for the first (in terms of the *CGP* structure) column. The signal indicating active cells finished its propagation in this column. For that reason in this column, the interconnection amongst fundamental cells is not required. Hence, even the route configurations of the cells are not useful as well. The detection cells are degraded to the implementation of the simple logic sum of two inputs. The following two figures show the diagrams of the *Column Detector* and the *Detection Cell* needed for the first column. From these diagrams it is evident that the whole *Column Detector* usage for the first column can be implemented by several OR gates and an encoder/adder.

The presented implementations of column detectors were described in VHDL and the source codes are available in the Appendix C of this thesis. The entity describing the column detector contains several generic parameters so that its re-use is very easy and straightaway. The following two tables show elements of the entity called "column_detector" which corresponds to the implementation described in the Figure 30. The Table 10 and the Table 11 show generic parameters of entity and items of the port declaration, respectively.

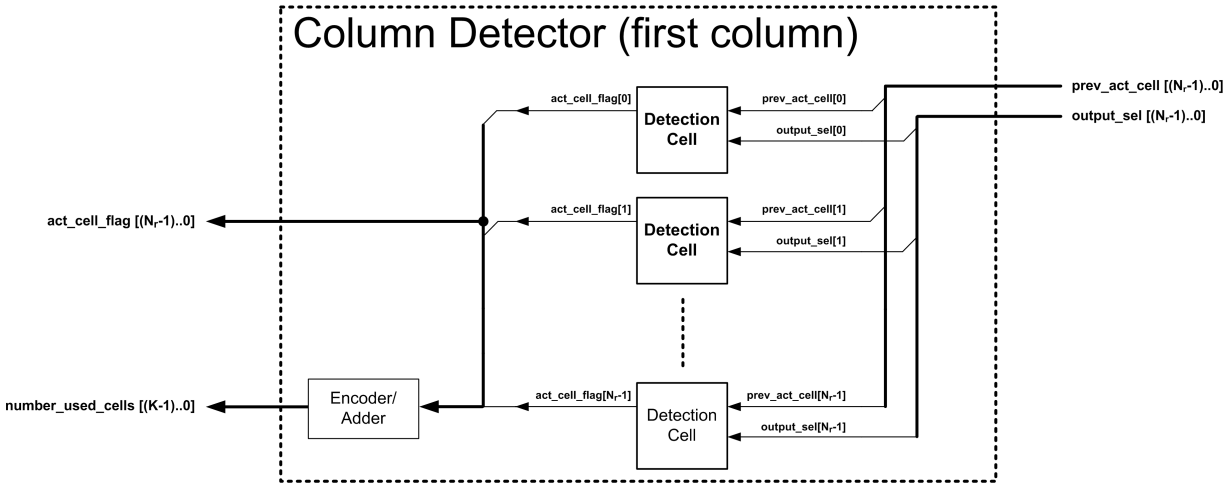


Figure 31. Diagram of Column Detector (for the first column)

Note to the Figure 31: The character 'N_r' denotes the number of cells in a column. The character 'K' means the width (in bits) of the vector expressing the number of used cells.

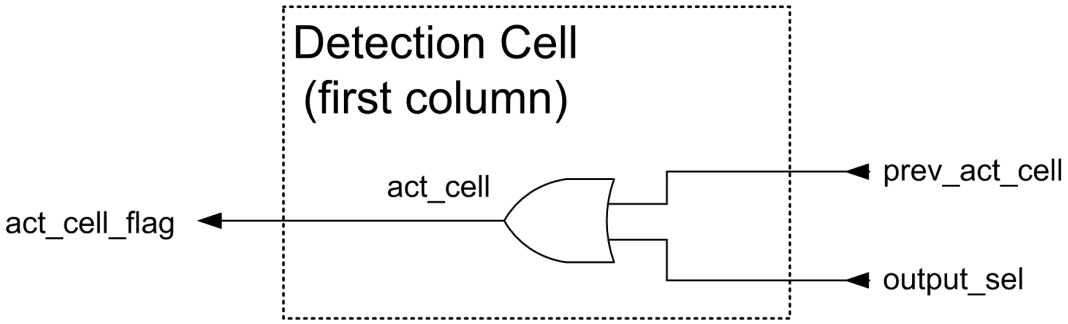


Figure 32. Diagram of Detection Cell (for the first column)

Table 10. Generic parameters of "column_detector" entity

Generic Parameter	Type	Description
NUMBER_CELL	POSITIVE	Specifies the number of cells in a column.
COLUMN_OUTPUT_EN	BOOLEAN	Defines whether the cells of the column can feed primary output/s. If this parameter is set to 'true', the cells can feed the outputs.
COLUMN_PRIM_INPUT_EN	BOOLEAN	Defines whether the cells of the column can be fed by primary inputs. If this parameter is set to 'true', the cells can be fed by primary inputs.
NUMBER_PRIM_INPUT	POSITIVE	Expresses the number of primary inputs.
ROUTE_CONFIG_WIDTH	POSITIVE	Specifies the bit width of a route configuration (connection genes).
FUNCT_CONFIG_WIDTH	POSITIVE	Specifies the bit width of a function configuration (function gene).
LAST_COLUMN	BOOLEAN	Specifies whether the column detector belongs to the last column of a *CGP* structure.

Table 11. Port declaration of "column_detector" entity

Port	Type of Signal	D	Description
column_config[J(1)-1..0]	std_logic_vector	I	The configuration of the whole column.
output_selected[NUMBER_CELL-1..0]	std_logic_vector	I	The input port for flags indicating that the cell feeds primary outputs.
previous_act_cell_propagation [NUMBER_CELL-1..0]	std_logic_vector	I	The input of active cells state.
act_cell_propagation[NUMBER_CELL-1..0]	std_logic_vector	O	The output of active cells state.
act_cell_flag[NUMBER_CELL-1..0]	std_logic_vector	O	The output port indicating flags of active/used cells in the column.
number_act_cell[K(2)-1..0]	std_logic_vector	O	This port expresses the number of active/used cells in the column.

Note to the table:

*(1) J = (ROUTE_CONFIG_WIDTH*2*NUMBER_CELL)+(FUNCT_CONFIG_WIDTH*NUMBER_CELL)*
(2) K = natural(ceil(log2(real(NUMBER_CELL))))

The bit width of the input port “column_config” is defined so that it accepts the configuration data of the whole column. The meaning of other ports and generic parameters is obvious. The entity implements just a combinational logic structure; this is why the port declaration does not contain clock input. It may appear that this approach is not the most suitable one. A synthesis and time analysis of asynchronous components is more difficult.

However, in this case, the presented structural elements can form various high level modules. The developer has to design a suitable placement of registers to a logic structure so that the timing requirements are met.

The structure of the final detector of active/used cells in the *CGP* chromosome is shown in the Figure 33. The figure describes the cascade coupling of *Column Detectors*. The rightmost detector does not use the *act_prev_cell* port; this port is fed by a null vector.

In the next stages of the cascade, the *act_prev_cell* ports are fed by the *act_cell_propagation* port of the previous column detector. Indeed, the first column detector feeds no column. Each used column detector provides fundamental data ports (vectors) indicating the active/used cells and their number. These vectors are merged into two vectors that represent the information about all used cells and the total number of used cells. In the diagram, there is also shown the *Output Config Decoder*. This unit produces the vector of flags that indicates which cell/node (cells/nodes) of the structure feeds the primary output/s. If the detection of active cells for a single primary output is required, the

Output Config Decoder can be equipped with a mask circuit. By the means of this functionality, the decoder produces only flags enabled by the mask register.

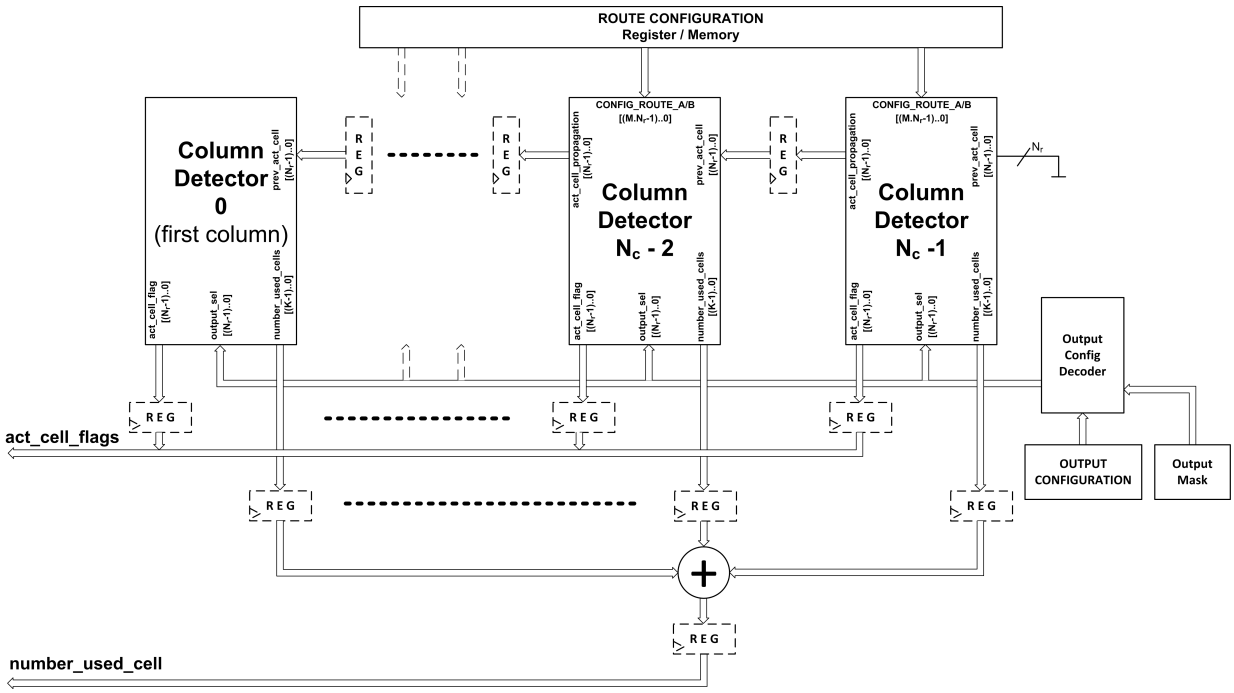


Figure 33. Final active cell detector

Note to the Figure 33: The abbreviation 'N_c' expresses the number of columns. The character 'N_r' denotes the number of cells in a column. The character 'K' means the width (in bits) of the vector expressing the number of used cells. The character 'M' denotes the width (in bits) of route configuration.

The whole detector can be designed just as combinational logic structure (circuit). However, it is clear that this approach does not correspond with the design recommendations. For that reason, the diagram suggests the placement of the registers (depicted by a dashed line) separating combinational data paths. The number and the placement of registers always depend on the current *CGP* structure and the timing and latency requirements.

The Figure 34 describes the *Output Config Decoder* with the mask register. It is a classical form of digital decoder. The fundamental primary outputs are masked by the *Output Mask* bits. The output port's width of the whole decoder is equal to the number of cells that are able to feed primary output/s. To each primary output belongs just one decoder with a group of mask AND gates. If the masking of primary outputs is not required, the AND gates and the mask register are omitted.

Note to the Figure 34: The abbreviation 'N_{cell}' expresses the number of cells able to feed a primary output. The 'N_o' denotes the number of primary outputs.

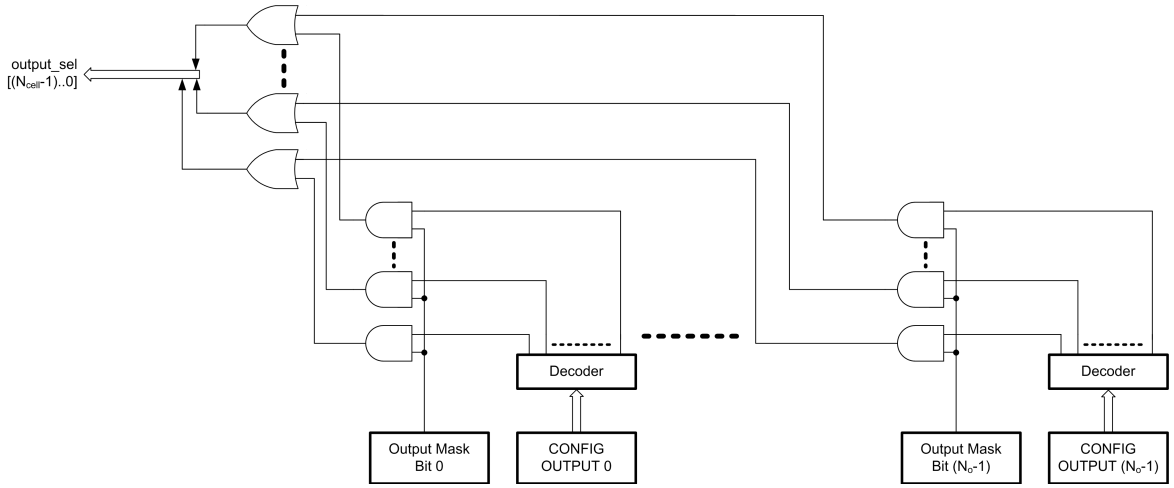


Figure 34. Diagram of the Output Config Decoder

4.4.1.3 Structures with Higher Value of l-back Parameter

In the previous chapters, the structures with *l-back* parameter equal to 1 were discussed. However, the designed detector also makes it possible to detect active genes in the structures where the *l-back* parameter takes the value of 2. Higher value than 2 is not considered, because these structures are not commonly implemented by the FPGA devices. However, on principle, the introduced keynote of the detection is not limited to this *l-back* value.

To perform the detection in the structure with *l-back* taking the value of 2, it is necessary to make marginal modifications in the detector circuit. Firstly, the focus will be given on the *Detection Cell* (see the diagram in the Figure 29). The cell has to be modified in order to provide the additional output indicating that the current cell is fed by the more distant column. This modification needs to extend the binary decoders to double width. The *Limiters* also have to limit to higher value (= (the number of rows x 2) + the number of primary inputs – 1). Finally, the whole AND/OR output logic is implemented twice. Such modified cell provides two outputs, the first output (*act_cell_propagation_L1*) indicates the interconnection with the closer column, and the latter (called *act_cell_propagation_L2*) implies the relationship with the more distant column.

The modifications of the *Detection Cells* affect the changes in the *Column Detector*. The following figure shows the *Column Detector – L2* which is a detector designed for structures with the *l-back* equal to 2.

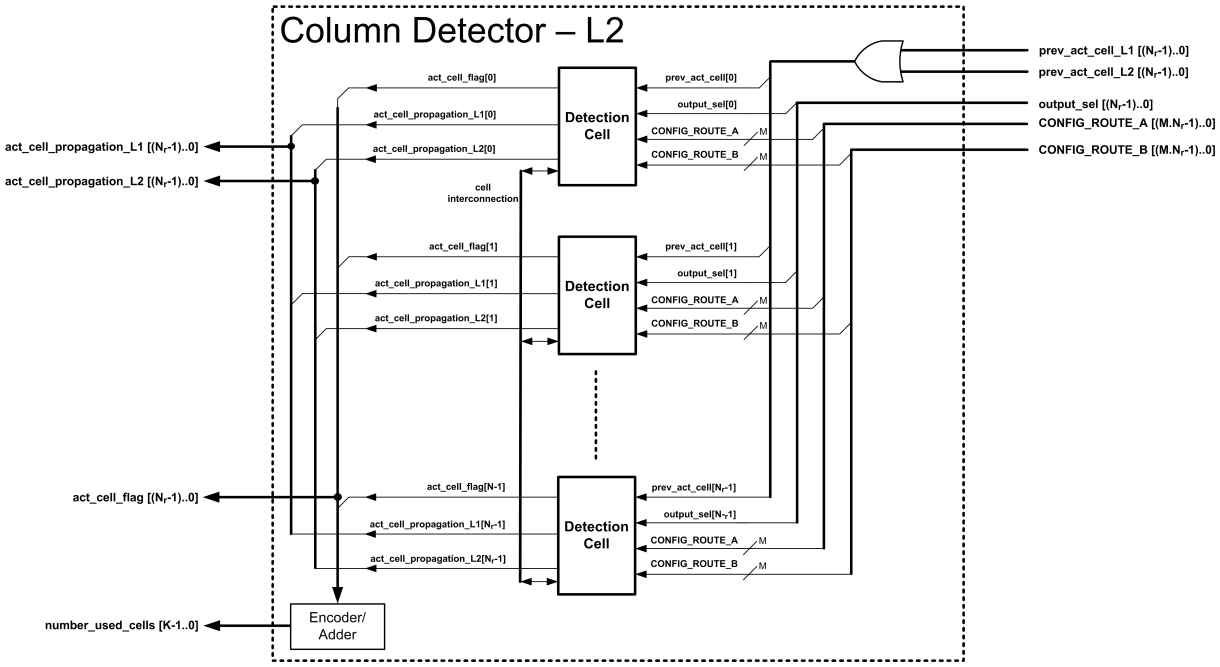


Figure 35. Diagram of Column Detector – L2

It is clear that the column produces the new vector *act_cell_propagation_L2* that is composed of particular *Detection Cells* (as it was explained above). The new input port is added as well – there are the *prev_act_cell_L1* and *prev_act_cell_L2* ports. The latter accepts the state about active cells from the more distant column. Both signals are logically summed and the result is connected to particular cells. The rest of the column stays unchanged.

These *Detection Columns – L2* can be interconnected; thereby, they form the appropriate circuit to detect active cells in the structure where the *l-back* is equal to 2. Such detector (for the structure with 5 columns) is shown in the Figure 36. Note that only the main changed signals are depicted.

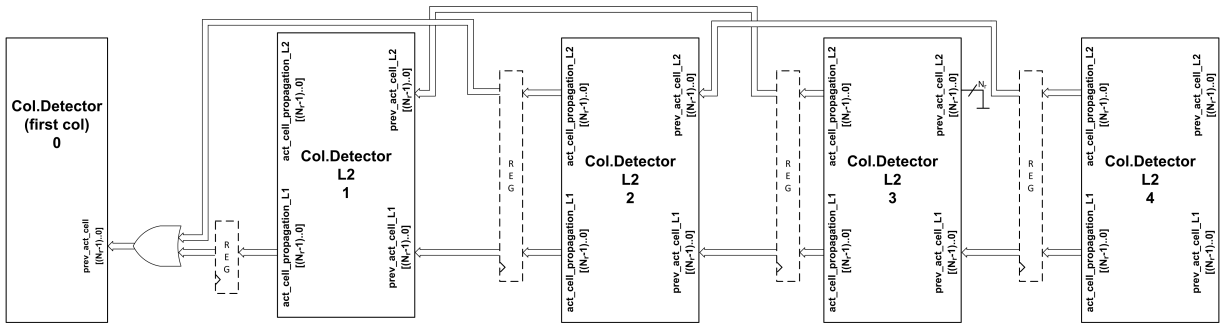


Figure 36. The final active cell detector for *l-back* = 2

The *Column Detector 0* is the detector described in the Figure 31, but its input is fed by the result of the logical sum of the *act_cell_propagation_L1* signal from column 1 and the *act_cell_propagation_L1* signal produced by the column 2.

The *Column Detector – L2* is implemented by the VHDL code (see Appendix D) as the "column_detect_l2*"* entity. The declaration of the entity is very similar to the "column_detector" entity which is described by the Table 10 and the Table 11. Only the ports *prev_act_cell_L2* and *act_cell_propagation_L2* are added.

4.4.1.4 Timing Requirements

The detector of active cells composed of structural elements, described in the previous chapters, was implemented and tested on the FPGA Cyclone IV (speed grade 7) device from Altera, Corp.

The detector was composed according to the *CGP* structure of the following parameters: the values n_r and n_c were set to 5, the numbers of primary inputs and outputs were set in the same way to the value of 5. The *l-back* parameter was equal to 1. All cells can feed the primary output and all cells can be connected to (fed by) primary inputs. The mask register of the *Output Config Decoder* is neglected. It is supposed that the detection of active cells related to all primary outputs is required.

The Figure 33 suggests the placement of registers in the detection structure. At first, these suggested registers are assumed. It means that the data paths between the *act_prev_cell* and *act_cell_propagation* ports are separated by registers. The registers are also placed on *act_cell_flag* and *number_used_cells* outputs. In other words, all outputs of a *Column Detector* are registered. This option of the detector (hereinafter marked as option 1) produces valid outputs after 7 clock cycles (it means the latency is 7 clock cycles). The detector can operate at the maximum clock frequency at approximately 171MHz, which is a very good value.

Further, the next option (option 2) was composed. The register was inserted past the *Output Config Decoder* (see Figure 34). This step reduces the combinational data path between the storage of output configurations and the outputs of the *Column Detectors*. Unfortunately, it also increases the output latency to 8 clock cycles. On the other hand, it can operate at a higher clock frequency – roughly at 209 MHz.

If focus is given to a *Detection Cell* (see Figure 29), it is obvious that a couple of decoders implements combinational logic and could produce a long data path. It has been tried to

place registers between OR gates (connected with decoders) and a group of AND gates. If the configuration of the whole column is stored simultaneously, all decoders can operate at the same time. Such registers in the option 4 were inserted and the registers among *Column Detectors* were omitted. The *Output Config Decoder* was used as unregistered. This tested option operates at lower maximal frequency 100 MHz. However, because of the omitted registers among columns, the output latency drops to 4 clock cycles.

The option 5 is based on the option 4, but the *Output Config Decoder* is registered. Thereby, the maximum operational frequency was improved to 124 MHz.

The registers used in options 4 and 5 were added to the solution represented by the option 2. This variant is marked as the option 3 and achieves the best results in terms of the maximum operational frequency – approximately at 240 MHz. This option was analysed in detail and the critical data path was found between the columns with indexes 3 and 2. Nevertheless, only 3 logic levels implement this data path (see Figure 37). This fact results in obtaining a high maximum frequency.

The results of all options are summarized in the Table 12.

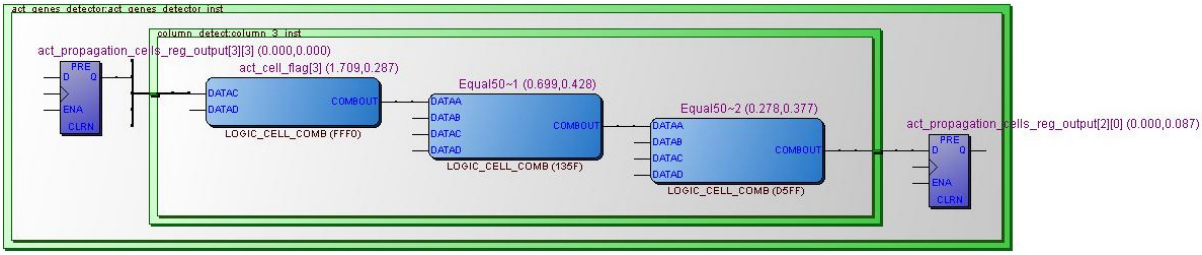


Figure 37. Critical data path of the option 3

Table 12. Options of register placement

	Registered Output				
Option	**Output Config Decoder**	**Column Detectors**	**Route Decoders**	**Max. Freq. [MHz]**	**Latency [clock cycles]**
1	×	✓	×	171	7
2	✓	✓	×	209	8
3	✓	✓	✓	240	8
4	×	×	✓	100	4
5	✓	×	✓	124	4

It is necessary to note that the presented values of the maximum frequency are approximate. The real value of the maximum frequency depends on a few factors, for example: the setting of a complier, the available area of an FPGA device, the partly random character of a route and placement process.

The presented implementations require approximately 605 LEs (logic elements) in the FPGA device Altera Cyclone IV. The registers keeping the configuration of columns and configurations of output nodes are included in this number (the option 3 uses 275 registers in total). The compilations of all options were performed and the optimization of speed was preferred. If the user prefers the optimization of used resources and area and accepts a lower operational clock frequency, he/she can use a different setting of the complier. The complier can perform a register packing process, which joins registers and functions using only LUT (look-up table) into a shared element. These steps can save a few logic elements (573 LEs was achieved with the use of an area optimization technique). The higher amount of used FPGA resources is balanced by the speed of the designed logic. If the designed structure is completed by auxiliary shift registers, it can continuously process a pipelined data stream (*CGP* chromosomes/individuals) at a very high clock frequency.

4.4.2 Mutation Unit

The mutation is the only genetic operator performed by *CGP*. The evolutionary circuit design is a very hard task from the point of view of the evolutionary algorithms because the fitness landscape is not usually smooth. Experiments presented in the publications [61][62] indicate that the standard crossover operators do not yield any benefit in the search process.

The process of the mutation was described in the chapter 4.1. Note that it is necessary to produce only valid values of genes. From the point of view of the software implementation of *CGP*, the mutation process is very simple and it does not present a significant issue. There are requirements: the allowed value range of the gene and a generated random number. By the means of these two input arguments, the new value of gene can be generated.

However, another situation arises if *CGP* algorithm is implemented by logic, it means perhaps by an FPGA device. In this case, the process of mutation becomes harder. In the published projects, the bit mutation and bit representation of genes are often used [62][63][64]. The bit mutation is represented by the inversion of bit/s of a chromosome. At the first glance, this approach can seem preferable to a classical integer representation. However, the use of integer representation yields considerable benefits. An integer directly determines the used function of function cells or the interconnections among them. On the other hand, the algorithm has to keep information on particular parts of chromosome,

because every integer of chromosome can take only limited values. Another disadvantage is that the generation of a new part of chromosome is certainly more difficult than the mutation in a bit representation – it means bit negation. The Figure 38 shows the hardware implementation of a bit mutation which was published in [62].

A simple consideration dealing with both – bit vs. integer – representations will be made. Assume the following instance: structure 1 x 16 (rows x columns); all cells can be connected to primary inputs; the *l-back* parameter = maximum; each cell can perform one of these 2-input functions – AND, OR, XOR, interconnection. Further, 5 primary inputs are taken into account. In case of integer representation, 3 integers are needed for each cell. Only acceptable values determining the interconnection are variable.

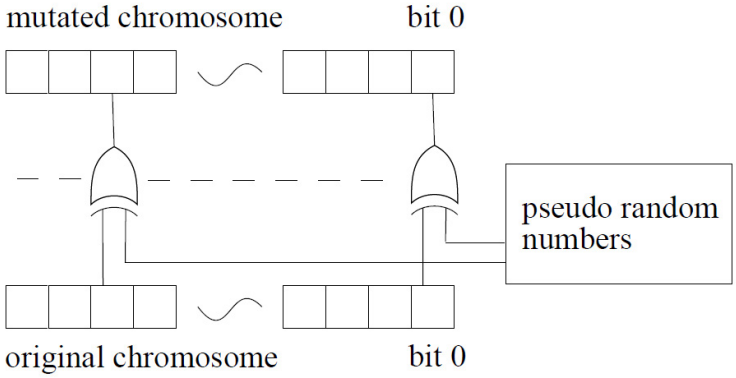


Figure 38. Hardware implementation of a bit mutation [62]

Note that a particular integer represents particular nodes directly. For example, the first cell can be connected only to primary inputs; therefore, the interconnection integer takes the values between 0 and 4. For the last cell, the interconnection gene value lies between 0 and 19.The integer determining the output cell is in the range between 5 (the first cell) and 20 (the last cell). In the bit representation, it is desired for the chromosome to be as short as possible. For that reason, each cell is defined by a different number of bits. In our instance, the first cell needs 2 x 3 bits for interconnection (5 primary inputs) and 2 bits to determine the function; it means 8 bits as a whole. Indeed, the last cell needs 2 x 5 bits for interconnection, thus 12 bits as a whole. The output cell is defined by 4 bits (16 cells). If we approach the chromosome only as a bit stream, it is obvious that it will contain incorrect values. For example, the last cell can be connected only to 20 nodes (5 primary inputs and 15 previous cells); however, 5 bits code to up to 32 nodes. If we want to use the search algorithm working with a bit chromosome, it is necessary to ensure the correct calculation of a fitness function even with an incorrect chromosome. The easiest way how to solve it is the assignment of a maximum value for a certain group of bits. If the group of

bits represents a higher than allowable value, the fitness function calculation uses this maximum value. It is necessary to note that only a change in the implementation of a fitness function is discussed; the bit approach to the chromosome stays unchanged.

It is clear that these steps increase the probability of neutral mutations. Let a 5-bit vector be assumed with its value meaningful range of [0; 20); the initial value is "11000" (it means 24 in decimal). If the bit mutation is performed so that one of the first three bits is inverted, the effective value of gene does not change. This effect can be reduced if *CGP* parameters are chosen so that all values given by the bit vector are valid/allowed. However, this approach can also reduce the connectivity of a structure. This step may or may not affect the evolutionary process. Now, assume the initial value "00000". If only one random bit is mutated (inverted), the result gene value may take values (in decimal) 1, 2, 4, 8, 16. It can represent a significant difference in comparison with the mutation in integer representation which generates the gene value without reference to the preceding value. Indeed, if more bits in gene are inverted, this feature is significantly suppressed.

Because of the facts introduced in the previous paragraphs, the author of this thesis has decided to implement *CGP* with a classical integer mutation. Indeed, this approach is not necessary to the successful hardware implementation of *CGP*. However, it brings a very significant feature; the behaviour of the implemented evolutionary system should be consistent with the *CGP* algorithm implemented by the software without the mentioned limits.

4.4.2.1 Implementation of Random Gene Generator

A special unit/component has been designed that generates the genes of a *CGP* chromosome. This unit provides a new value of genes continuously. There is an essential fact that all generated gene values are correct in relation to the gene type. It means that the designed *Random Gene Generator* produces only meaningful values.

The aim of this unit is to produce several vectors. The vector determining the position of a gene in the chromosome is called *Gene Index*. It can take unsigned value in the range [0; the number of genes). The second fundamental vector – the *Gene Value* vector – produces the new value of the gene. The two values provided by both vectors are in a mutual relationship and they have to be understood in this way. Other features (index of a cell, type of a gene etc.) of the produced genes may be derived from these two vectors.

The Figure 39 shows the basic concept of the unit.

The introduced concept required two sources of random numbers for its correct function. The input port *Individual Active Cell Flags* is not necessary, technically speaking; it is used to implement an additional functionality as will be discussed later. In the first step, the value of the *Gene Index* vector is generated. For this functionality, it is necessary to generate random numbers in a defined limited range.

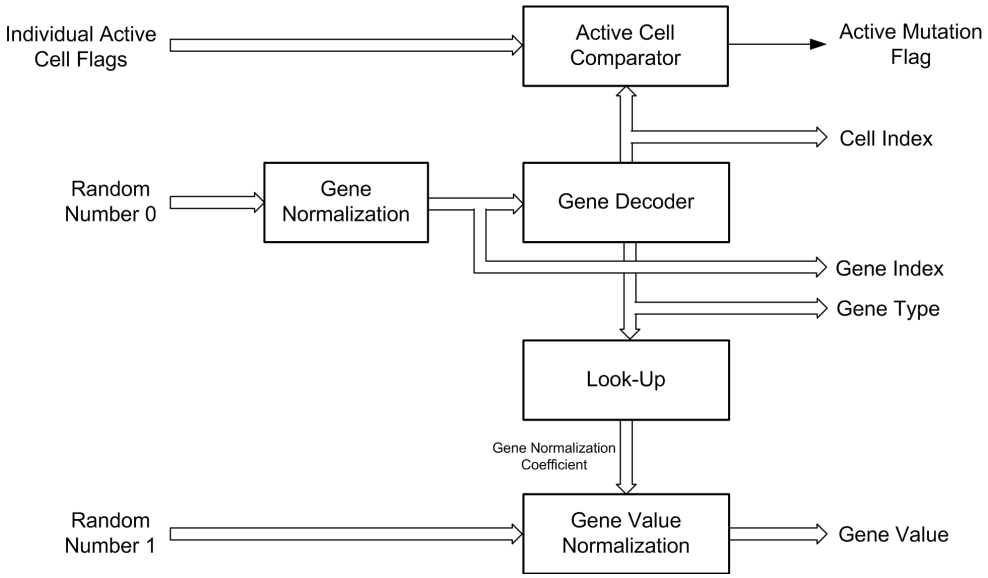


Figure 39. Concept of the gene generator

However, the available (pseudo) random number generators (e.g. LFSR generators) often produce values in a reduced range; for example, the range is given by powers of two. For that reason, it is necessary to normalize the obtained random value in order to be in accord with the required value range. This process is very simple and the following formula expresses it:

$$G_i = floor\left(R_n \frac{\Lambda}{R_{nmax} + 1}\right) \quad (15);$$

where

G_i ..... *the resulting value of Gene Index,*

R_n ..... *the generated random number,*

R_{nmax} ..... *the maximum random number,*

Λ ..... *the length (number of genes) of a chromosome.*

For example, let the length of a chromosome be 10 genes. Assume that a 8-bit LFSR pseudo-random number generator is used and its maximum produced value is 255. According to these values, the *Gene Index* will be produced in the value range [0; 10). The division in the formula can be substituted by a constant because both input arguments are fixed.

The new generated *Gene Index* feeds the *Gene Decoder*. This decoder derives other features of a gene – that informs about the index of the column, the type of gene (determining cell function or cell input) and the *Cell Index* (determines a cell/node represented by a gene). On the basis of these features, the look-up table defines a suitable coefficient for the normalization of a gene value. For example, if the newly generated *Gene Index* represents the configuration of a cell function, the normalization coefficient will be equal to the number of cell functions. The normalization of gene value is executed in the same way like the normalization of the *Gene Index* – see the formula (15). However, the value of the normalization coefficient is used instead of Λ. By this approach, a new gene and its values are generated.

In addition, the module *Active Cell Comparator* decides whether the generated gene belongs to the active cells of an individual represented by the input vector *Individual Active Cell Flags*. The bit width of both vectors – *Individual Active Cell Flags* and *Cell Index* – is equal to the number of cells in the *CGP* structure. They take the value of '1' for the given position if the corresponding cell is active/used. The comparator implements only a simple AND-OR logic. It drives logic high, if both vectors contain '1' on the same position. Indeed, despite all this, if the generated gene represents an output configuration gene, this gene is automatically declared as active.

According to the introduced principle, the mutation unit was implemented and described by VHDL. The parameters of this unit were chosen so that it produces genes for the topology of 5 rows x 5 columns, *l-back* = 1, 5 primary inputs and also 5 primary outputs and each cell can implement up to 4 functions. The primary inputs can feed whichever cell and each cell can represent a primary output. Note that the generated gene values define the cell function directly. However, the genes specifying the connectivity of the cells or the primary output/s point relatively to nodes. For example, if a gene of a cell in the last column takes the value equal to 9, it means that the cell is fed by the bottom cell in the previous column. In other words, the values [0;5) determine the primary inputs, and the range [5;10) points to cells in the previous column. On account of these facts, the coefficients for normalization of the gene values have to be deduced – see Table 13. The table doesn't present the normalization coefficient for genes defining the output nodes. The value of this coefficient is 25 for all these genes, because all cells (25) can drive primary output/s.

It can be seen that the defined parameters of a *CGP* structure imply only several different coefficients. For that reason, they can be produced by a simple decoder or a look-up table.

Table 13. Gene normalization coefficients

Column	Type of Gene		
	Function	Input A	Input B
0	4	5	5
1	4	10	10
2	4	10	10
3	4	10	10
4	4	10	10

The implemented *Random Gene Generator* was simulated and tested on Altera FPGA Cyclone IV. It was fed by a pair of LFSR pseudo-random generators (with the data width of 19 and 16 bits). The mutation unit consumes approximately 206 LEs (LFSR generators are included). The structure of the unit is fully pipelined and it provides a new gene in every clock cycle. The normalization of the *Gene Index* is implemented by the multiplier constructed from logic elements because one argument is variable and the second is fixed. The embedded multiplier is used for the normalization of a *Gene Value*, because both input arguments of the multiplication are variable. The mutation unit implemented by this way can operate at the frequency approaching roughly 200 MHz. The critical data path (6 logic levels) was detected in the decoder which derives the *Cell Index* from the value of *Gene Index*.

The simulation waves are depicted in the Figure 40. The designed unit uses 7 pipeline stages. Assume the mark *A* (in the figure) is an initial point when the random number is generated (by 19-bit LFSR) and feeds the *rand_number_0* port. The value of the random number is 14,023. It implies that the *Gene Index* (called *gene_address* in the waves) will take value of 2 (see formula (15)) after passing through pipeline processing. Also on the basis of this value, a new value of the *Gene Normalization Coefficient* (in the Figure 40 denoted as *gen_value_norm_coef*) is produced after 5 clock cycles. See the mark *B*, it is depicted that the *gen_value_norm_coef* is equal to 5 and the *rand_number_1* takes the value (produced by a 16-bit LFSR) 22,151. After another two clock cycles (see the mark *C*), the unit provides a completed valid output. The newly generated gene points to the third gene of a chromosome (*gene_address* = 2), this gene maps the top cell in the first column (*gene_column_sel* = „000001“; the MSB of this vector denotes the genes determining the output nodes) of the *CGP* structure. The input vector *indiv_act_cell_flags* represents the active cells of the ‘parent’ chromosome. Because of this value, the gene is

considered an active gene (signal *gene_active* is high; LSB of the vector *indiv_act_cell_flags* implies that the first cell of the 'parent' is active and the generated gene encodes just this cell). The generated gene takes the value of 1.

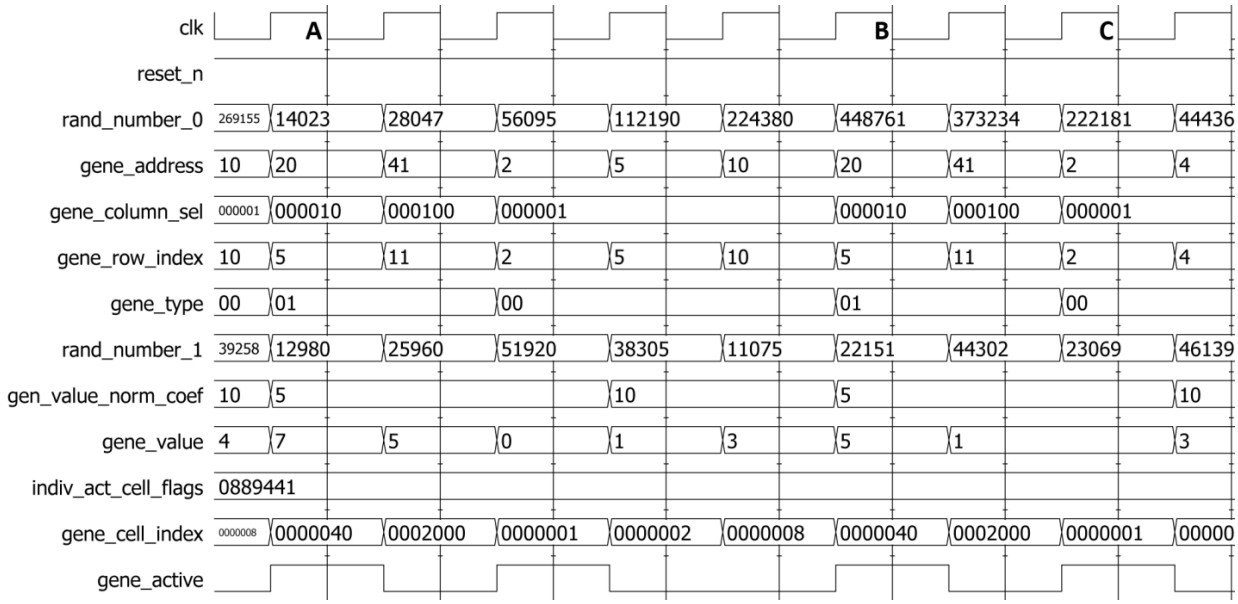


Figure 40. Simulation waves of Random Gene Generator

4.4.3 Fitness Function Calculation

The calculation of the fitness function performed by FPGA devices is the subject of this chapter. The topic is divided into two sections – the first one describes the reconfigurable structure which provides response to the input stimuli; the second section deals with the process of the calculation and its control.

4.4.3.1 CGP Reconfigurable Structure – Virtual Reconfigurable Circuit

The response of the candidate solution (chromosome) to the input stimuli is obtained by a *Virtual Reconfigurable Circuit (VRC)*. The *VRC* is a structure that truly represents the grid structure defined by *CGP*. This *VRC* is inspired by the publications [65][66][9][10][67][68][69] and a lot of evolvable systems are based on it. Note that the *VRC* designed and used by this work observes the rules according to *CGP*. It means that *VRC* does not implement any simplification of the structure, for example limitation of cell connectivity as that is used by some authors.

The following requirements were defined before the design of the *VRC*:

- Easy and quick reconfiguration of the *VRC*
- Pipelined processing
- *L-back* parameter equal to 1 or 2

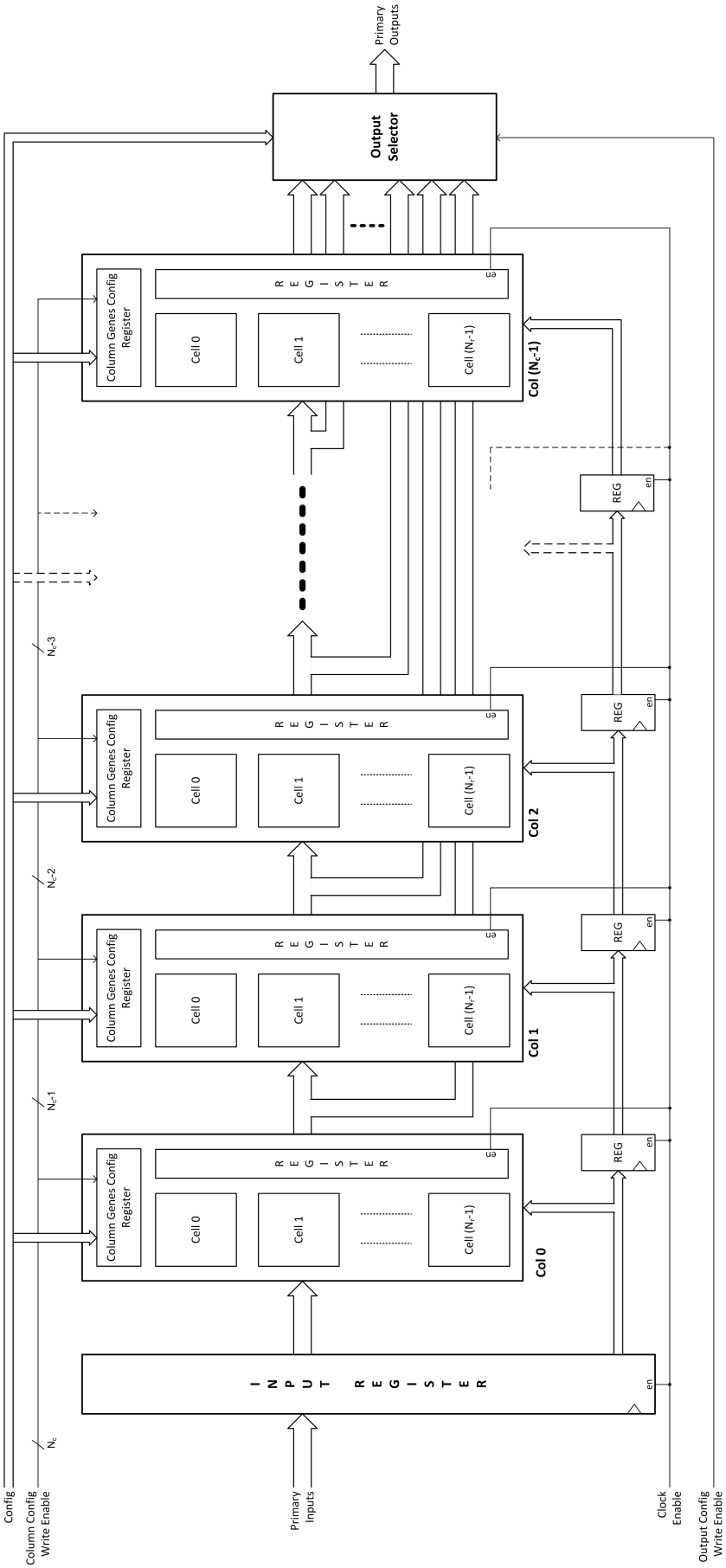


Figure 41. Virtual Reconfigurable Circuit

- Modular architecture
- Multi-output support
- Heterogeneous structure (various cell function sets within one *VRC*)

The VHDL source codes of the *VRC* may be generated automatically by the *Evolutionary Designer* (see chapter 4.2). The Figure 41 describes the whole *VRC* structure with the *l-back* parameter equal to value 1. Other general features are assumed, which means that the primary inputs can be connected to an arbitrary cell and each cell can feed primary outputs.

The primary inputs are directed to the input register. This register is not obligatory, but its use improves the timing parameters of the final circuit. Further, the primary input data are shifted by means of the shift register so that relevant input data are valid in each pipeline stage. The *VRC* consists of fundamental columns which correspond to the columns of the *CGP* grid structure. These columns ensure the processing of primary input data. Each column output is registered and feeds the inputs of the next column. It accepts data from the previous column and the primary input data (from stages of the shift register). Indeed, the only primary inputs are connected to the first column. The functionality and response of the *VRC* depends on the configuration data representing the chromosome (individual) of an evolutionary algorithm (*CGP* in our case).

The parts of this configuration are stored into registers in the fundamental columns and in the *Output Selector* (will be described later). The special 'write enable' signal belongs to each part of the chromosome and as such it can be stored independently on other parts; it means that partial reconfiguration is supported.

Now, the length of the configuration bit stream will be analysed. At first glance, it is clear that the total number of bits will be given as sum of three configuration parts – the first column, the remaining columns and the output configuration. However, when the *l-back* parameter is equal to 2, the length of the second column must be calculated separately.

Assume the *VRC* in grid structure 5x5, 4 primary inputs, 2 primary outputs, *l-back* = 1, the cell function set implements up to 4 functions. The length of the configuration of the first column is given as $L_{c0} = [2 + (2 \cdot 2)] \cdot 5 = 30$ bits; two bits are needed to encode 4 primary inputs and the same number of bits for the selection of the cell function. The configuration of the second column is expressed by the same relation, but it is necessary to be aware that each input of a cell can be fed by primary inputs and output of the first column (it follows

9 signals => 4 bits). It means that $L_{c1} = [2 + (2 \cdot 4)] \cdot 5 = 50$ bits. The remaining columns need the same length of configuration as the second column. The output configuration is given by the number of cells which can feed the primary outputs. In our case, the total number of these cells is 25; it follows that ($L_{out} = 5$) for the mapping of single primary output 5 bits are required. Thus, the whole length of the configuration bit stream is 240 bits. If the *l-back* parameter takes the value of 2, the length of the bitstream remains unchanged (in this demonstrated example).

The designed *Evolutionary Designer* software tool makes it possible to generate the *VRC* with heterogeneous structure. This term is established by the author of this thesis and represents the *CGP* structures using more than one set of cell functions. It means that the cells can implement various sets of functions. This fact can affect the length of the configuration bitstream if the sets contain a different number of functions. The tool also introduces novel parameters limiting the connectivity of primary inputs and outputs. These features are discussed in the chapter 4.2.2.2 and are in relation to the length of bit stream as well.

The connectivity of the primary outputs also corresponds to the general definition of *CGP*. In some published projects (e.g. [70][47]), the outputs are fixed on certain cells/nodes in the last column. The designed *VRC* contains the module called *Output Selector* which ensures the connectivity of the primary outputs according to output configuration. However, it doesn't implement only a simple multiplexer.

The *VRC* implements the *clock enable* signal for all registers so that the pipelined processing can be suspended. The bit width of the data signals is also optional; it may be a single-bit signal or a vector.

In the followings paragraphs, the fundamental part of the *VRC* will be discussed in detail. The primary element is the (reconfigurable) cell. The implementation of the cell (see the Figure 42) is similar to the architecture of an arithmetic logic unit. It implements certain number of cell functions and the output of the cell is given by one output of these functions selected by the multiplexer depending on the cell configuration (*function gene*). The cell is implemented as a combinational circuit and the cell output is not registered. Each cell function is described by the VHDL entity. The complexity of the cell functions and their total number affect the timing parameters of the final *VRC*.

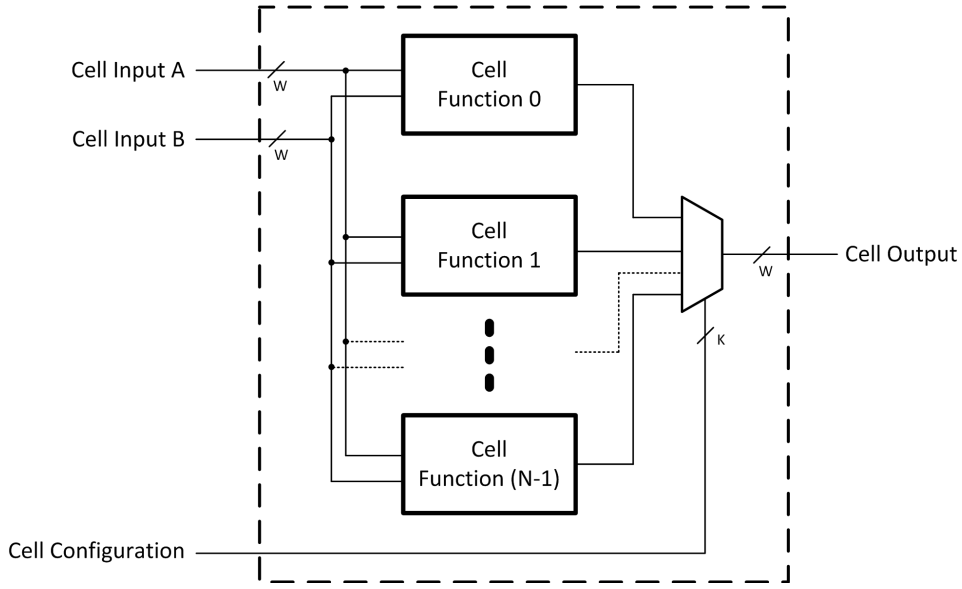


Figure 42. Cell of the VRC

Note to the Figure 42: The symbol 'W_d' expresses the bit width of the data signals. The 'K' denotes the bit width of Cell Configuration; the 'N' means the number of cell functions.

The cells form a column which ensures a suitable connectivity amongst the column data inputs, primary inputs and cell inputs. The column also keeps appropriate configuration in the register ("Cell [x] Gene Register"). One gene determines the cell function and two genes control a pair of multiplexers. These multiplexers select the primary inputs or data from the previous column/s and feed the inputs of a cell. The outputs of cells packed in the column compose the column data output which is registered by the output register – this register separates the whole *VRC* to particular pipeline stages. Note that multiplexers in the columns consume a lot of logic elements. Their complexity is directly proportional to the number of rows in the structure and the number of primary inputs. However, the value of the *l-back* parameter is the most significant factor. The diagram of the column is shown in the Figure 43.

The columns process the input stimuli data; the last task of the *VRC* is to connect cells/nodes according to the gene value to the primary/program output. If the *VRC* implements only one primary output, this situation is optimal, because the normal multiplexer can be used in order to connect the cell with the primary output. The latency between the primary input and output is given by the column that produced the output value and the presence of the input or output register. However, it is necessary to note that the position of the column producing primary output is changed during the evolutionary process. It follows that the latency of the output changes as well. Of course, this behaviour is not desirable. However, if the *VRC* contains two and more primary outputs, the situation

becomes more complicated. Each of the primary outputs can be fed by a different column; in consequence of that, the latency of the primary outputs may vary. This nuisance has to be compensated in the consequential module that processes the primary outputs produced by the *VRC*. For that reason, the *Output Selector* has been designed to compensate the different latencies of the primary outputs. According to the structure parameters and the output connectivity, a suitable compensation is implemented within the *VRC*. This feature ensures the constant latency of the primary outputs. The *Output Selector* is depicted in the Figure 44. The diagram shows the selector managing only one primary output; the same circuit is needed for each and every output.

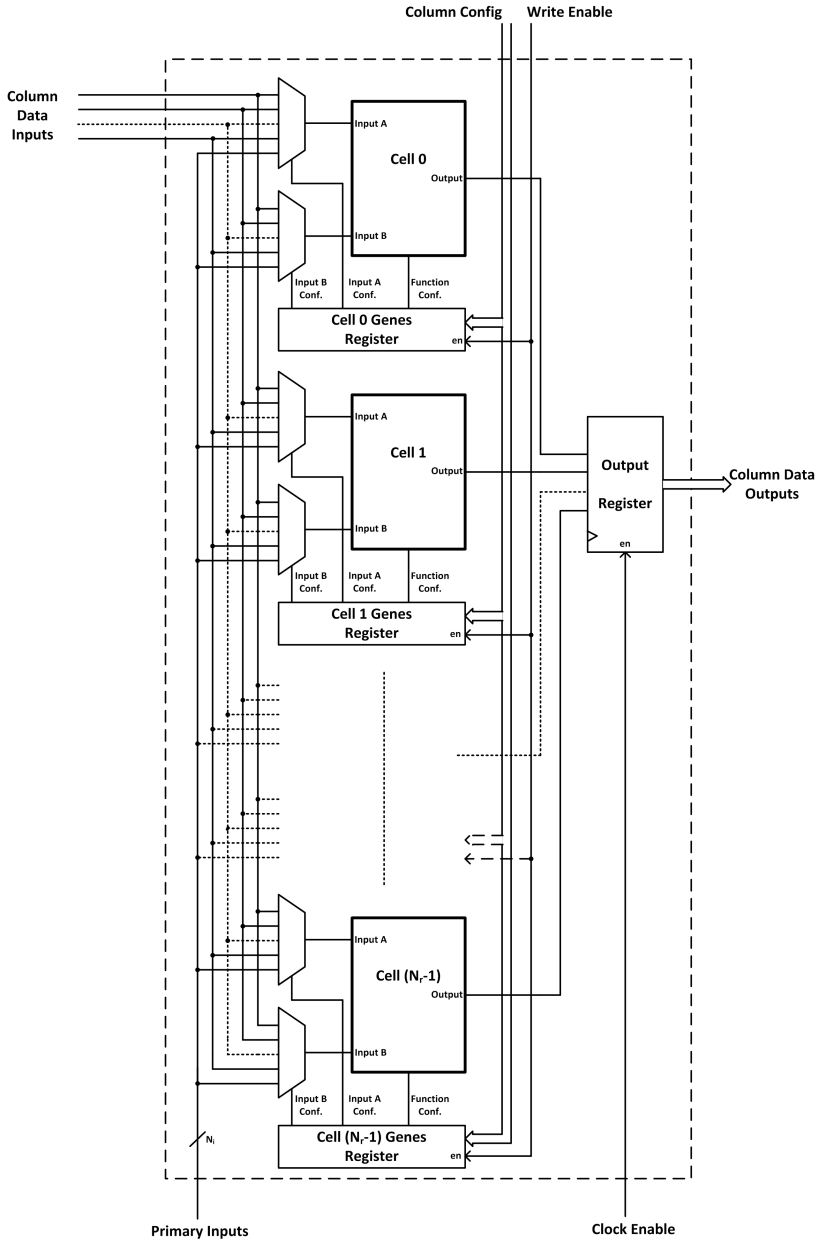


Figure 43. Column of the VRC

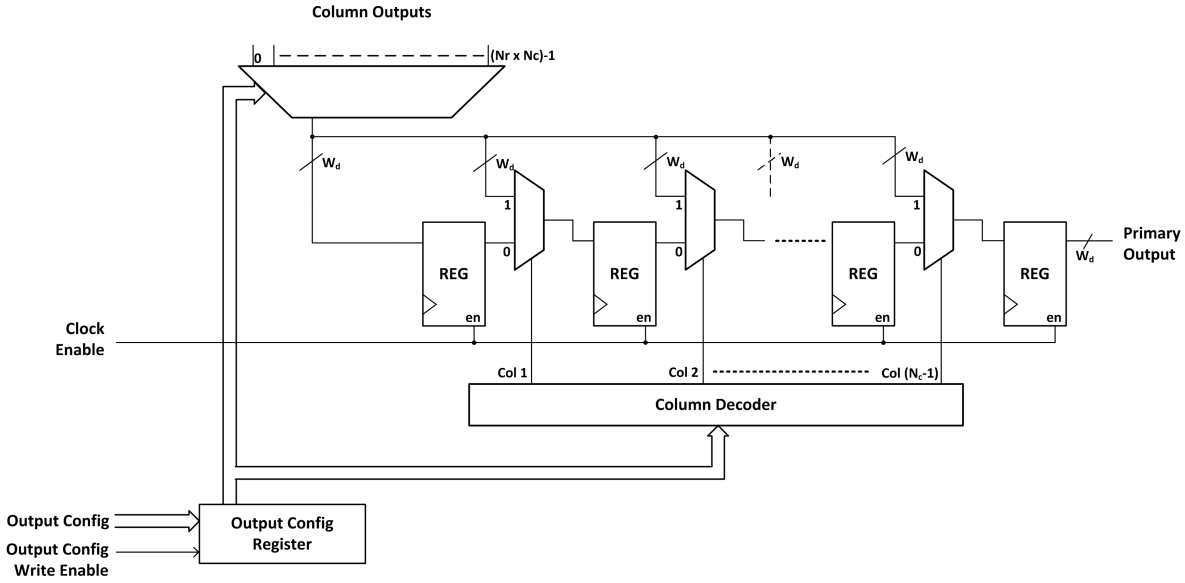


Figure 44. Output Selector

Note to the Figure 44: The symbol 'W_d' expresses the bit width of the data signals.

The data from all cells (if available) are connected to wide multiplexer. It is controlled by the output configuration gene stored in the register by a simple interface (data bus and 'write enable' signal). By this multiplexer, the cell is selected and the rest of circuit compensates the output latency. The *Column Decoder* produces flags determining the column which contains the output node/cell. These flags control a group of multiplexers which defines the length of a shift register (comprising particular registers).

For example, assume that the output configuration points to the cell in the first (index 0) column. The flag indicating this column is not produced because it is not used by the circuit. It means that all multiplexers are in the position 0; hence the signal from an output cell passes through all registers. On the contrary, if the output cell/node is situated in the last column (flag in the diagram is called "Col (N_c-1)"), the output signal is registered only by the rightmost register. This register serves as an output register and ensures that all outputs of the *VRC* are registered.

Until now, mainly the structure with the *l-back* parameter equal to 1 was discussed. However, the designed software tool *Evolutionary Designer* also generates the *VRC* corresponding with the *l-back* which takes the value of 2. The principle of the *VRC* remains identical, as it has been introduced in the Figure 41. The increase in the value of the *l-back* parameter implies to implement auxiliary pipeline registers. These registers shift outputs of particular columns to be processed by the next columns. The Figure 45 represents the placement of the auxiliary registers. For five columns, three registers are

needed. The higher number of registers does not present the main problem, because the architectures of FPGA devices are 'register rich'. However, it is important to realize that the multiplexers in the columns become more complex, because the number of their inputs increases. If the focus is given to "Col 3", it is clear that the column accepts data from two previous columns and primary inputs (they are not shown in the Figure 45).

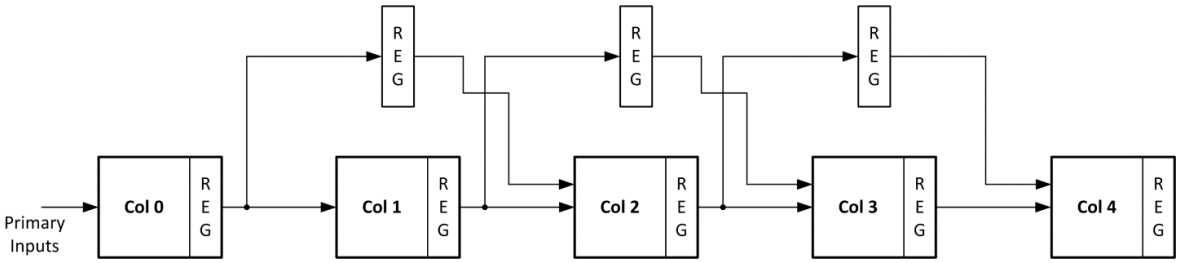


Figure 45. VRC with *l-back* = 2

Further, let's assume the structure in the Figure 45 with *l-back* = max. The number of auxiliary registers rises to six and the last column accepts data from four columns and primary inputs. In the line with the fact that the data length is usually wider than one bit, the multiplexer structures can allocate significant area in an FPGA device. For that reason, other authors also restrict the *l-back* parameter to the value of 2.

At the close of this subchapter, the author would like to mention that the *VRC* also implements a so-called 'valid shift register'. This register shifts the signal *input_data_valid* through the pipeline chain and it indicates (at the end of the chain) that the output data (on primary outputs) are valid – the signal *output_data_valid* is produced. The length of this shift register corresponds with the number of pipeline stages (input and output registers are also taken into account) of the *VRC*.

4.4.3.2 Control Process of the Calculation

This section deals with the control process of the fitness function calculation. However, only a general process of this task is discussed. The definition of a fitness function depends on the task which can require a specific stimuli and calculations. For that reason, it is impossible to design the generic form of this function and the input stimuli generators. The following text describes the designed modules which control the process of the fitness calculation independently on the kind of an application. The following figure shows the general (simplified) concept of the designed system for the fitness calculation.

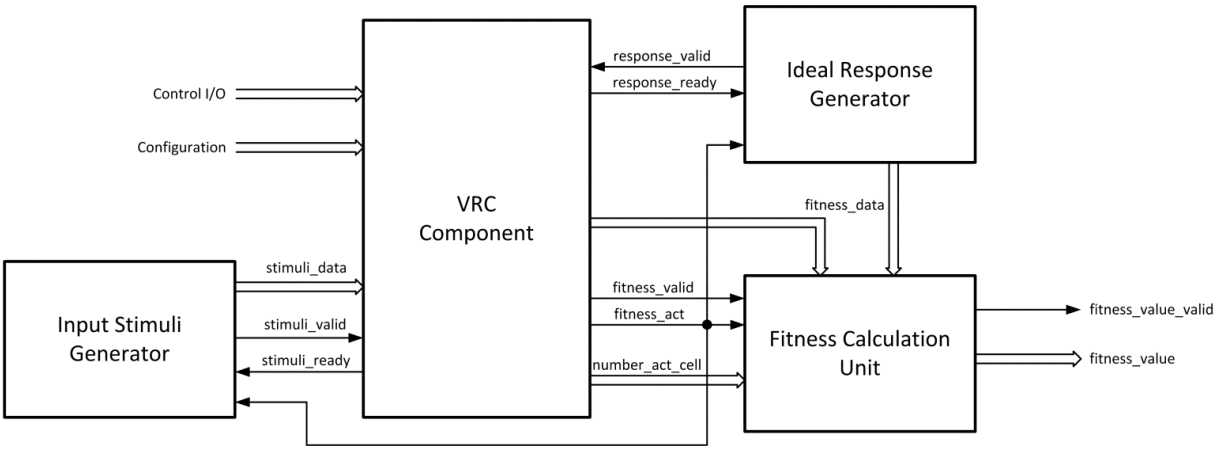


Figure 46. Fitness calculation – the general concept

The *VRC Component* module is the main engine that controls the whole process of the calculation. This part is task-independent and it can be reused in other projects. For successful calculation of the fitness function, a pair of data sources is needed – the source of input stimuli and the ideal response of a *CGP* structure (implemented by a *VRC*). The former one feeds the inputs of the *VRC*; the real output response of the *VRC* is compared with the ideal response. According to the degree of the correspondence between these two signals, the unit of the calculation determines the resulting value of fitness. From the Figure 46 it may be seen that the *VRC Component* contains the control I/O and an input port for the configuration (chromosome of *CGP*). Further, the module implements handshake interfaces for input stimuli (module called *Input Stimuli Generator)* and for a generator producing an ideal response (*Ideal Response Generator*). The module *Fitness Calculation Unit* performs the calculation of the fitness function; it must confirm the completion of a calculation by the signal *fitness_value_valid* and provide the final value on the *fitness_value* port. For this module, the *VRC Component* provides a response of the *VRC* and gives information about the state of fitness process. It can also give the number of used cells in the candidate solution (chromosome).

Further, the *VRC Component* will be described in detail by means of the Figure 47. In the diagram, there are two already discussed elements – the *VRC* and the *Active Gene Detector*. They are completed by the unit *Fitness Calculation Controller*. It implements a state machine that controls the process of the fitness calculation. The work of the machine is controlled by three signals – the *calc_start* starts the whole process; the *calc_finished* indicates that the process of calculation has been finished; the input vector *number_cycles* represents the unsigned number determining the number of input stimuli vectors. It is necessary to note that the signal *calc_finished* gives information that the state machine

finished its function. However, the final fitness value is derived by another component which can cause additional latency.

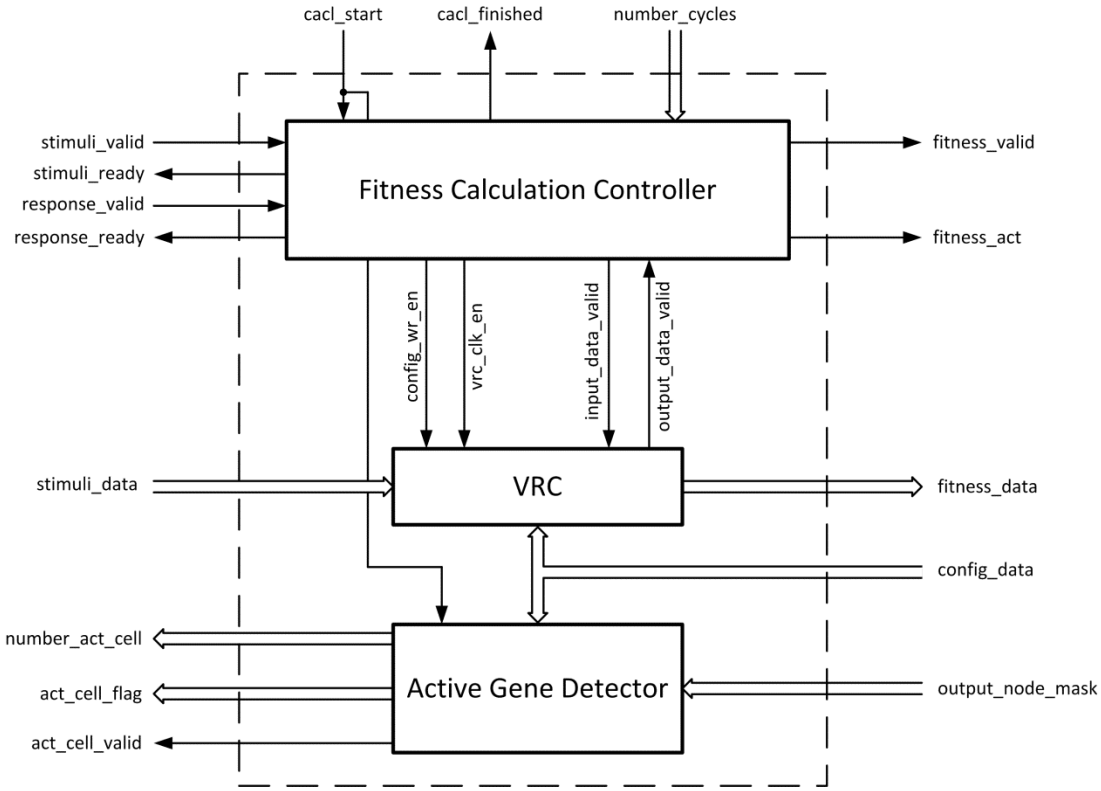


Figure 47. Diagram of the VRC Component

The state machine controls the *VRC* and it is controlled by three signals. It enables or disables the data processing in the *VRC* by means of the clock enable signal (called *vrc_clk_en*). The configuration of the *VRC* is stored into its inside registers by the *config_wr_en* (configuration 'write enable') signal. The validity of the input data is indicated by *input_data_valid*. Similarly, the *output_data_valid* tells us that the output data produced by the *VRC* are valid. The state machine also implements the control flow for the generators of stimuli (signals *stimuli_valid*, *stimuli_ready*) and the ideal response (signals *response_valid*, *response_ready*). It is formed by an often used interface – a pair of signals *valid* and *ready*. The signal *ready* is driven by the *Fitness Calculation Controller* and indicates that the controller is ready to accept data. By the *valid* signal, the connected component (stimuli or ideal response generator) confirms the validity of the data. If the controller drives the signal *ready* to low, the generator has to wait – the output data of a generator remain unchanged. If both signals – *ready* and *valid* – are high, the state machine receives valid data; the generator can provide new output data vector in the next cycle.

This interface is compatible with Avalon Streaming Interface [71] developed by the Altera Corp.

The remaining two signals produced by the *Fitness Calculation Controller* indicate that the data for fitness calculation (it means the response of the *VRC* and the ideal response generated by the generator) is valid – the signal *fitness_valid*; and that the fitness calculation is in progress – the signal *fitness_act*.

The input port *stimuli_data* is fed by the generator of the input stimuli. The control signal *calc_start* also stores a chromosome to the *Active Gene Detector*. The meaning of other signals is obvious.

The *Fitness Calculation Controller* is a simple state machine implementing only four states. Now, let's analyse the work of machine. If the generator of the input stimuli and an ideal response provides data continuously (this means they are able to generate new vector every clock cycle), the functionality of the machine is restricted only to the starting of calculation process and the performing the number of cycles (defined by the input vector *number_cycles*). However, if one of the generators at least provides vector with latency time, the machine's functionality expands. Assume the following situation when the generator of an ideal response stopped to provide new data, while the input stimuli are available without intermission. In this case, the *VRC* can process data until the whole pipeline chain is filled. After that, the *VRC* provides valid output data (primary outputs), but this data cannot be processed to calculate the fitness, because the corresponding ideal response is not available. For that moment, the work of the *VRC* is suspended by means of the signal *vrc_clk_en* and the state machine waits for the ideal response data. As soon as this data are provided (*response_valid* goes high), the work of the *VRC* is resumed. If the *Fitness Calculation Unit* can accumulate the output data from the *VRC* and the ideal response is not required together with this data at the same time, this functionality is not vital. On the other hand, this 'waiting' functionality has to be implemented somewhere, either by the control state machine, or by the unit performing the fitness calculation.

The state machine terminates the work if the specified number of cycles (vectors of the primary outputs) is achieved.

The functionalities of the state machine are described by the state diagram shown in the Figure 48. It contains the following states: RESET, IDLE, CALC_PROCESS, WAIT_FOR_DATA. The machine is initialized into RESET state after asynchronous reset

(it occurs also after an FPGA's boot sequence) of the system. When the reset is released, the state IDLE is set spontaneously. In this state, the machine remains as long as the starting signal *calc_start* is not set. After this, the signal *calc_finished* goes low and the state machine passes to the state CALC_PROCESS. In this state, the machine starts to count the number of input stimuli and ideal response vectors. If the data processing of the *VRC* runs without delays, the machine state stays unchanged. After achieving the number of cycles, the internal condition *calc_completed='1'* is satisfied and the state machine returns to the IDLE state. If the stream of ideal response is not continuous and the situation described above arises, the *VRC* processing is suspended; the machine goes to the state WAIT_FOR_DATA where it remains until the stream of ideal response vectors is resumed.

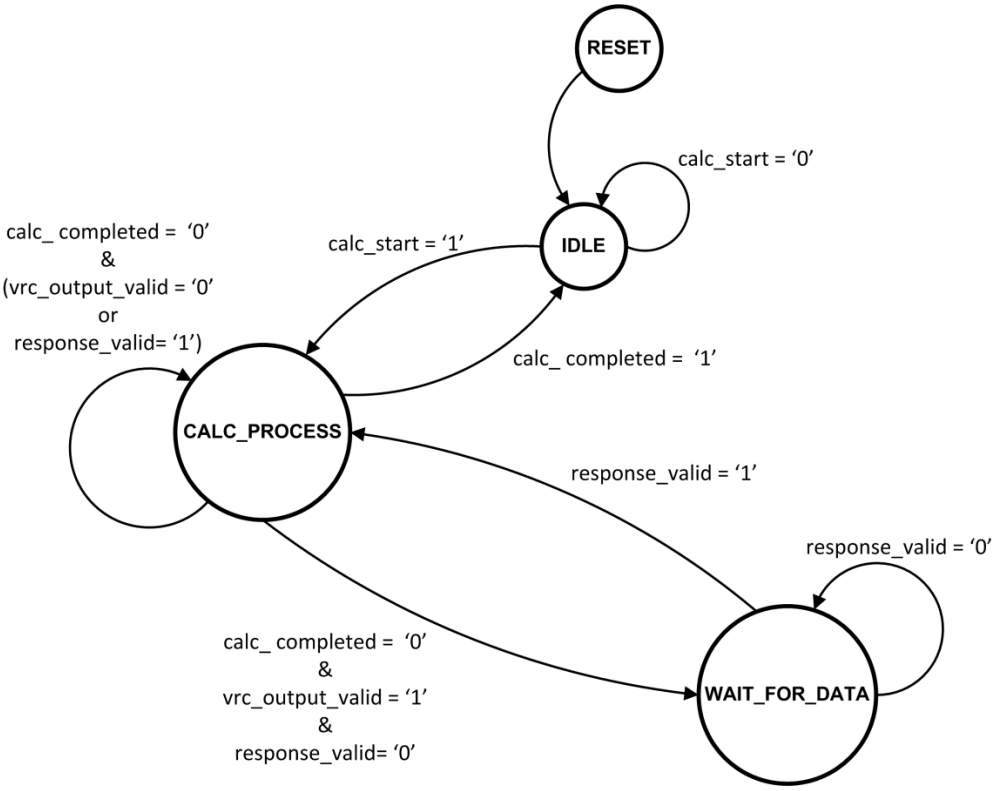


Figure 48. State diagram of the Fitness Calculation Controller

The above-mentioned functionality is also described by the simulation waves in the Figure 49. The waves show that after the start of the calculation process, *stimuli_valid* is high, but *response_valid* is low. After several clock cycles, the pipeline chain of the *VRC* is filled, the *vrc_output_data_valid* indicates the availability of valid data on its outputs, but the signal *response_valid* remains low. For that reason, the controller disables the work of the *VRC* (signal *vrc_clk_en* goes low) and the state machine passes into the

WAIT_FOR_DATA state. When *response_valid* is changed to high, the calculation process continues in the CALC_PROCESS state.

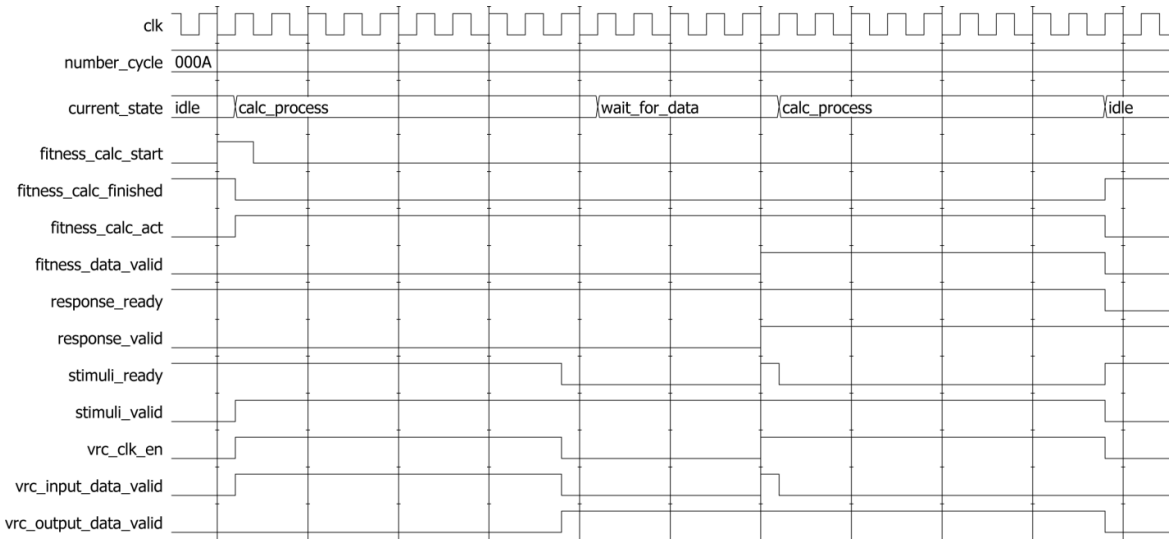


Figure 49. Simulation waves of the Fitness Calculation Controller

In terms of the FPGA resources, the *Fitness Calculation Controller* is a very simple circuit; it consumes roughly 43 LEs and can operate at clock frequency roughly 190 MHz. These presented values are derived from synthesis on the FPGA device Altera Cyclone IV.

4.4.4 Configuration Memory

The search algorithm needs a memory for the storage of the individuals (chromosomes). This memory can be implemented in different ways. If the algorithm is performed by the processor, the individuals can be kept in its operational memory. On the other side, the FPGA can also implement this memory by registers. By means of the registers, it is possible to design the appropriate structure of the memory [73][62]. However, this approach is limited by the memory area requirements.

It is also necessary to design the memory architecture to enable the cooperation with other parts of the system to be trouble-free. In the previous chapters, the modules performing a random generation of new genes and the fitness calculation were introduced. The character of the search algorithm and these designed modules specify the configuration memory requirements. The *Random Gene Generator* produces a new value of the gene, its address (locus) in the chromosome and the index of the column. On account of this fact, it is obvious that the memory should be able to write a single gene. The *VRC Component* can accept the configuration data (chromosome) of the *VRC* every clock cycle. For that reason, it is convenient to read the whole chromosome from the memory in one clock cycle as

well. The third request is based on the behaviour of the search algorithm which often copies the parent chromosome; subsequently, this duplicated chromosome undergoes the mutation process. It follows that the ability of the fast memory writing of the whole chromosome also sounds like useful.

The blocks of the embedded memory in an FPGA device can partly meet the mentioned requirements. And for that reason, this type of memory is often used in the implementation of the evolutionary algorithms. [73][62][65][74][75][9] Unfortunately, not all series of FPGA devices offer the same possibilities in terms of the embedded memory. The memory architectures, which can be formed, are limited by the used FPGA device. This is a very significant fact; it means that the designed memory architectures described by the VHDL code are not fully portable on other series/models of FPGA devices.

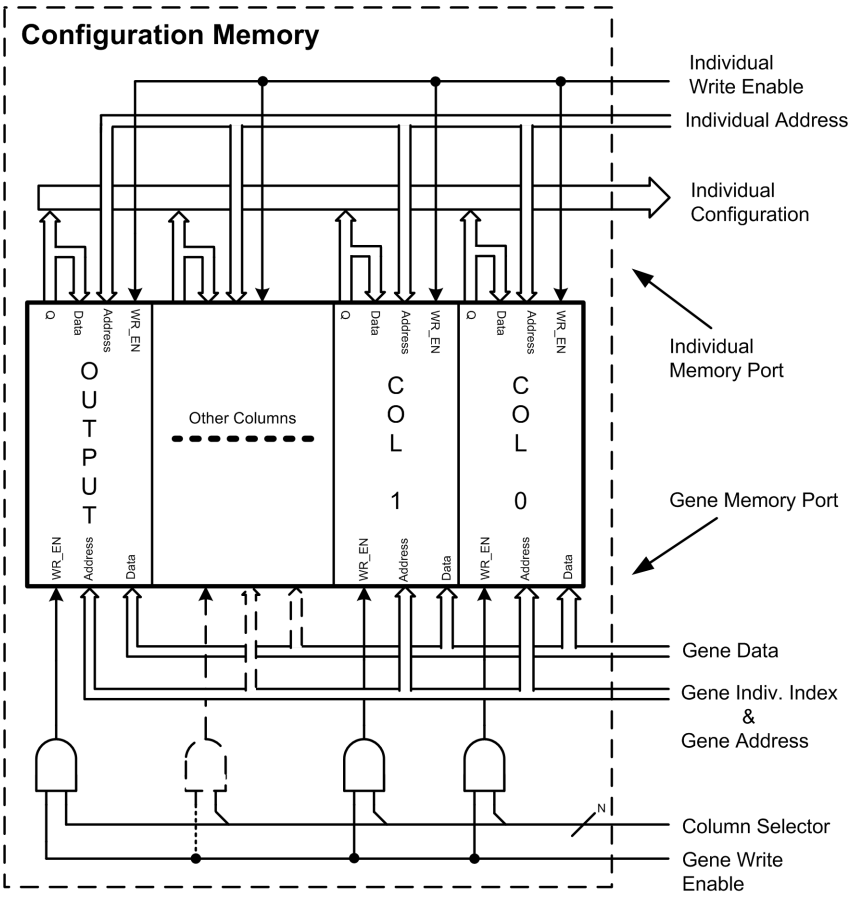


Figure 50. Architecture of the configuration memory

The designed architecture of the configuration memory shown in the Figure 50 is inspired by [65] and modified to be compatible to other designed modules. The memory consists of embedded memory modules. Each module forms just one column over all individuals. For example, the module implements memory for the column 0; however, eight individuals are

needed, so this module provides memory space for eight configurations of the column 0. The number of used memory modules depends on the number of columns in the *VRC* structure. One module is dedicated to the output configuration.

The modules are implemented as a true dual-port synchronous memory with different widths of data ports. The first port is called *Individual Memory Port* and its data width is derived from the data width of the whole column. From the Figure 50 it can be seen that all outputs of the memory modules (called *Q*) form the *Individual Configuration* vector. If the address (*Individual Address*) defining the position of an individual in the memory is set, the full configuration data (chromosome) is available in the next clock cycle. The data input of the *Individual Memory Port* is directly fed by the memory output; this link makes it possible to copy the configuration data from the source to the destination position within two clock cycles (the first cycle for reading, the second cycle for writing). The operation of writing to the memory is controlled by the *Individual Write Enable* signal.

The second memory port called *Gene Memory Port* implements the narrower data width. The addressable element is represented by one gene. The address value is composed of a pair of vectors; the *Gene Indiv. Index* (higher part of the address) defines the offset of the individual and the port *Gene Address* (lower part of the address) specifies the memory location of the gene in the chosen individual. The width of the data port is equal to the data width of the widest gene (of the given column). The address and the data ports are connected to all memory modules (columns). By the means of the simple AND logic, the signals *Column Selector* and *Gene Write Enable* produce *WR_EN* signals for the modules. The *Column Selector* indicates the active column (module); for example, if the *Column Selector* takes the value of "000010", the signal *WR_EN* of the second column can be active when *Gene Write Enable* is set.

The designed memory architecture is able to write the value of a particular gene and it also reads/writes the complete configuration data in one clock cycle. Nevertheless, it is necessary to be careful, because the implementation of a dual-port memory brings risk of data conflicts. If the user attempts to write data to the same address location from both ports at the same time, write conflicts happen. This results in unknown data being stored to that address location. If the FPGA device does not implement any conflict resolution circuitry, it is needed to handle this conflict explicitly/externally. The FPGA device Altera Cyclone IV used for the tests in this thesis contains no conflict arbiter; see [76] for a detailed description of the dual-port memory mode.

The dual-port memory also brings along another disadvantage described below. As has already been noted, the width of the *Gene Memory Port* equals to the data width of the widest gene of the column. However, the genes in the column can be of different widths. The next restriction is related to the number of genes; the embedded memory limits this number to powers of two. All these facts cause that more bits of the embedded memory than really needed are allocated. Let's assume the following example: the number of individuals is equal to 4; the genes are 4 bits (connectivity configuration) and 2 bits (configuration of a function) wide; the column consists of 5 cells => 15 genes. By a simple calculation, it is given that 200 bits of memory are required for this column (in four individuals). Unfortunately, the memory allocated by the FPGA will be larger. The widest gene is 4 bits wide; the power of two which is closest to the number of genes is 16 (= 2^4). It follows that this column allocates 256 bits (16 x 4 x 4) of embedded memory for 4 individuals. This overhead represents a trade-off between the appropriate memory architecture and the memory requirements.

4.4.5 Search Algorithm

In the previous chapters, the particular elements of *CGP* were discussed. This section describes the cooperation of these components in order to form a complete system performing *CGP*. Indeed, the designed components can be used separately and the presented approach shows only one of many possible ways. The search algorithm is often performed by a processor core and the time-consuming parts of the algorithm (fitness calculation) are executed in user logic of the FPGA device. For this approach, the *VRC Component* (including *Active Gene Detector*) can be used so that major modifications are not needed.

However, a simple search form of an algorithm was designed; the complete *CGP* algorithm can be implemented only in an FPGA device and no processor core (hard or soft) is required. According to the previous discussion in the section 4.3, related to the reduction of the fitness calculations, the compact version of *CGP* was established and presented. Remind that the author denotes it by the abbreviation *CCGP (Compact CGP)*; *CCGP* produces only one offspring/mutant per generation and the fitness calculation is performed only when the phenotype is changed. This type of *CGP* was implemented. It was to prove that the principles discussed in the section 4.3 can be easily performed and implemented by the designed logic in an FPGA device. The author of this thesis is not aware of a similar project that would have already been published.

At first, the whole system performing *CGP* will be described and discussed further. It is shown in Figure 51. Except the already known components as the *Random Gene Generator*, the *Configuration Memory* and the *VRC Component*, there are also other two components – the *Algorithm Controller* and the *Avalon Interface*. The former one controls the realization of the evolutionary design. Master control, transfer of the algorithm parameters and results are implemented by the module called *Avalon Interface*. If the focus is given to the *Configuration Memory* it can be seen that the *Individual Memory Port* (described in the Figure 50) feeds the *VRC Component* and, as mentioned in the section 4.4.4, the output of the memory provides chromosome data in parallel every clock cycle. The *VRC Component* uses this chromosome data in order to obtain the response of the *VRC* to input stimuli. The fitness function is calculated by the *Fitness Calculation Unit* which accepts ideal response data and real response produced by the *VRC*.

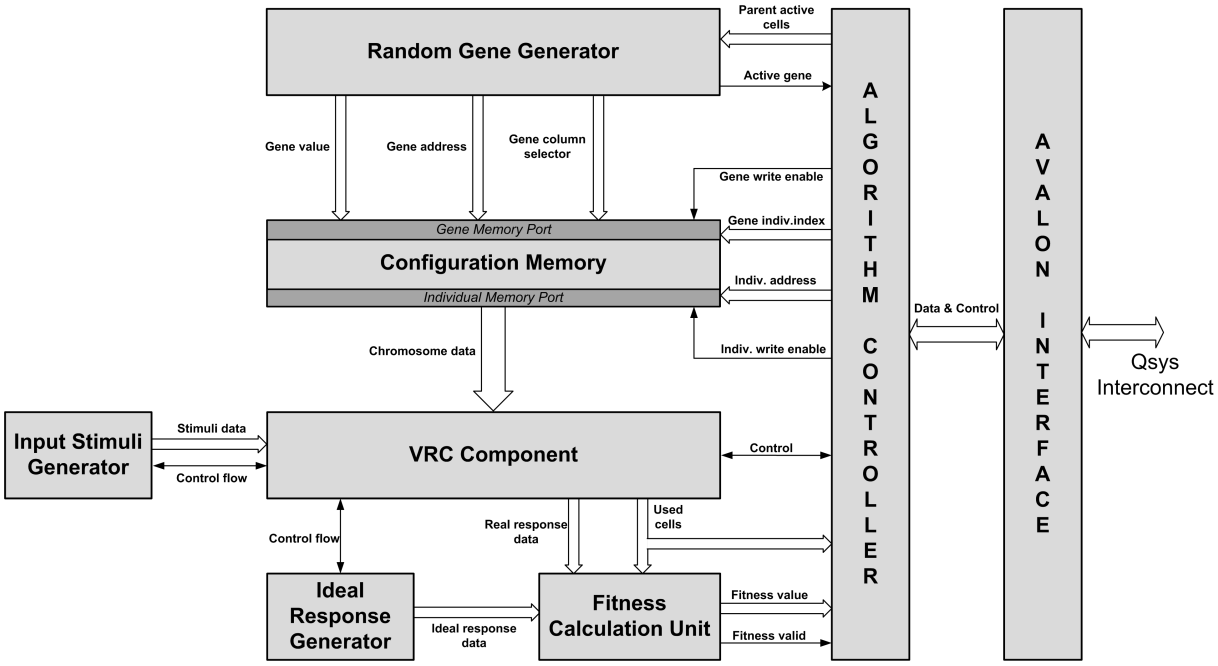


Figure 51. CGP system

The output port representing the current fitness value is connected to the *Algorithm Controller* that also controls the *VRC Component* by the means of the control signals.

The second memory port of the *Configuration Memory* – the *Gene Memory Port* – is connected to the module generating random genes. The *Random Gene Generator* produces signals specifying the gene position and its value. Remaining input ports of the *Configuration Memory* are fed by the *Algorithm Controller* that generates the values of the addresses, 'write enable' signals and controls the read/write memory operations.

Apart from producing random genes, the *Random Gene Generator* also generates the flag indicating the mutation of an active gene. For this reason, it needs a vector (called *Parent active cells*) that represents active cells/nodes of the parent chromosome.

As noted previously, the *Algorithm Controller* performs the search algorithm, controls the evolution process and produces all control signals. This controller is closely linked with the module that implements the *Avalon* slave port. This modern and frequently used memory-mapped interface (developed by Altera Corp.) allows an easy connection to other components and a superior system. By this interface, the designed system can be integrated to the Qsys system (integration tool made by Altera Corp.).

The used search algorithm is fully consistent with the algorithm described by the flow-process diagram in the Figure 22. The algorithm was implemented by the finite state machine corresponding to the state diagram shown in the Figure 52.

The fundamental states will be discussed; however, it is important to note that the presented transition conditions are simplified (only final conditions are introduced) by reason of clarity and better readability.

The machine is initialized into the RESET state after the asynchronous reset (it occurs also after an FPGA's boot sequence) of a system. When the reset is released, the state IDLE is set spontaneously. The machine remains in IDLE state until the starting signal *evol_start* is set. Then the machine passes to the INIT state.

During the INIT state, all signals and registers are set to the initial values. From the point of view of an evolutionary algorithm, the initial parent has to be generated. This initialization is carried out randomly so the controller sets the signal *Gene write enable* to high – this causes that random genes are written to a parent chromosome in the *Configuration Memory*. The position of a parent in the memory is given by the value of the signal *Gene individual address*. If this memory write operations of random genes are repeated many times, the parent chromosome contains random genes and thus the initialization can be terminated (flag *init_finished* goes high).

After the initialization, the machine passes to the FIT_PROCESS state that controls the calculation of a fitness function. The machine starts this calculation (performed by the *VRC Component*) and waits until the end. If the fitness value is valid, the flag *fitness_calculated* is set and the state machine passes to another state. After the fitness calculation, the termination condition is tested; if the evaluated individual brings the required fitness value,

the algorithm can be terminated (it passes to the FINISHED state). If the terminate condition is not satisfied, the state passes to one of the following two states. It passes to the state called PARENT_COPY if the candidate chromosome gets a lower fitness value than found until now (the fitness value of a parent). On the contrary, if the fitness value is better than the parent fitness or remains the same as the parent, the machine passes to the MUTANT_STORE state.

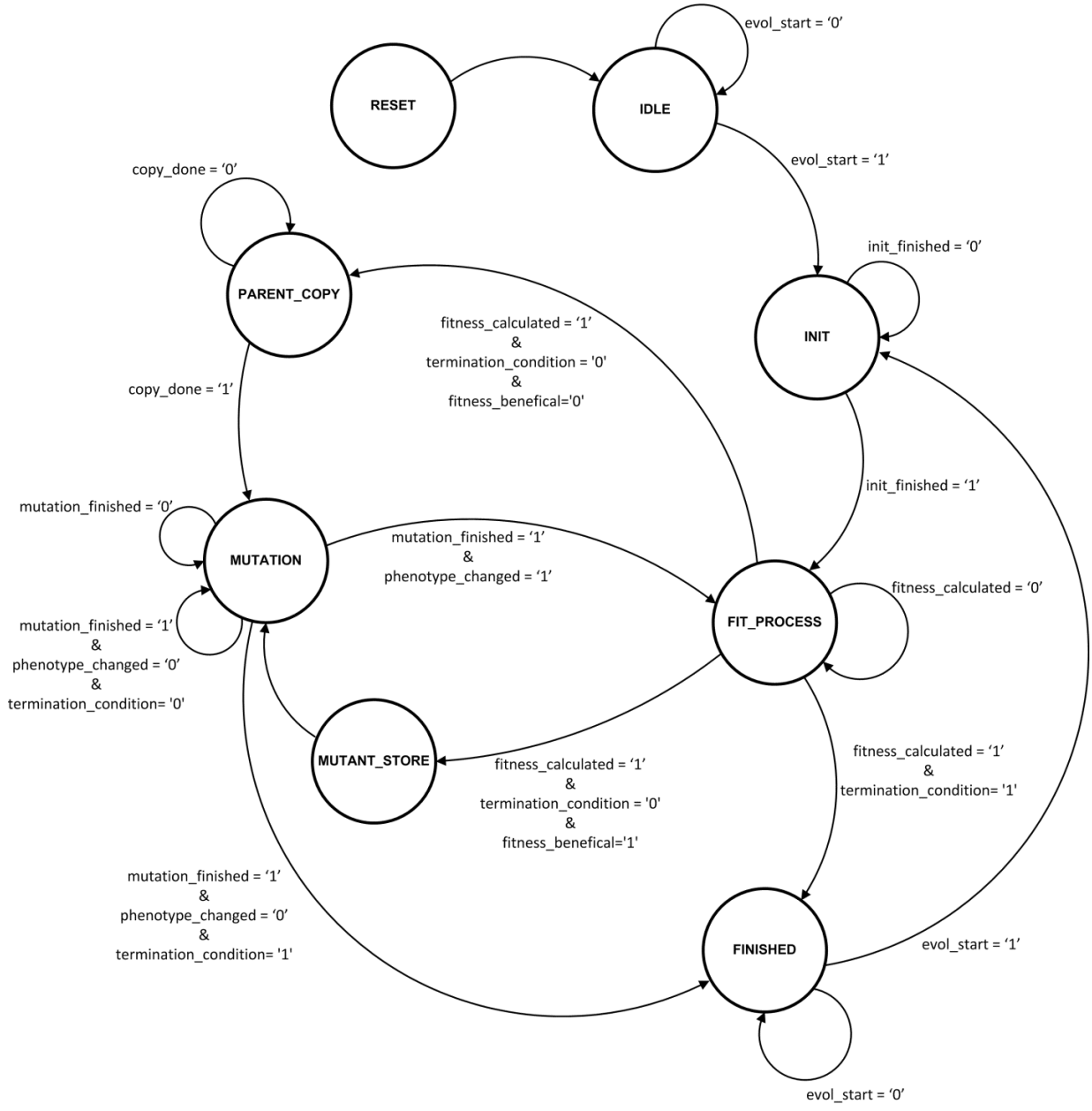


Figure 52. State diagram of the search algorithm

The PARENT_COPY state copies the parent chromosome to the position of the mutant chromosome. This process occurs when the fitness value of the current mutant is not beneficial. A useless mutant is discarded – replaced by the parent. Such duplication forms

the starting position for the next mutation operation. The machine passes to the MUTATION state after the parent is duplicated.

The MUTANT_STORE state causes the progress of an evolutionary process. It replaces the parent chromosome by the current mutant. It means that the mutant chromosome is duplicated in both memory positions and the machine is ready to go to the MUTATION state.

The MUTATION state creates a mutant by modifying the duplicated parent chromosome in the *Configuration Memory*. The state machine sets the signal *Gene write enable* for a period given by the mutation rate parameter. If the mutation rate takes the value equal to 2, two random genes are written into the memory, and so on. By the means of the *Active Gene* signal, the state machine is informed whether an active or an inactive gene was mutated. According to this fact, either the machine passes to the FIT_PROCESS state (a phenotype was mutated), or the mutation process is started again (a phenotype remains unchanged). If the limit of cycles is achieved, the algorithm is terminated (it passes to the FINISHED state).

The machine stays in the FINISHED state until a new run of evolution starts.

To increase the effectiveness of memory accesses and to avoid the risk of data conflicts (see subchapter 4.4.4), the memory positions of the mutant and the parent are not fixed. They can swap their positions – the pointers are swapped actually. In the PARENT_COPY state, the parent chromosome is loaded from the memory. In the next clock cycle, the chromosome is stored into the memory position of the current mutant. However, the first mutation is performed in the same cycle. At this moment, the swapping happens. The former position of the parent is the new position of the mutant and vice-versa. This change of positions makes it possible to write to the memory from both ports at the same time without the risk of data conflicts.

From now on, the focus is given to the moment when the designed algorithm reduces the number of fitness calculations. Assume that the mutation rate takes the value of 2. This means that the state machine allows writing two random genes. However, if these two genes do not change the phenotype (see the *phenotype_changed* signal), the next mutation process follows immediately. This moment is shown in the simulation waves in the Figure 53. Two mutation processes are performed between the points *A* and *B*. It is shown that the four random genes are stored. Note that the current numbers of performed

generations and fitness calculations are expressed by the signals *evol_cycles_performed* and *evol_fit_calc_performed*, respectively.

The Figure 54 shows the part of the evolutionary process. The state machine starts (the signal *fitness_calc_start asserted*) the fitness calculation in the point *A*. The fitness function is calculated (the point *B*) after certain number of clock cycles (the number of cycles is task-dependent) and the result value is equal to 137. The fitness of the leader was 135. For that reason, it is clear that the chromosome yields an improvement (the *fitness_benefical* signal is high). The chromosome is stored and represents the parent in the next part of the evolution.

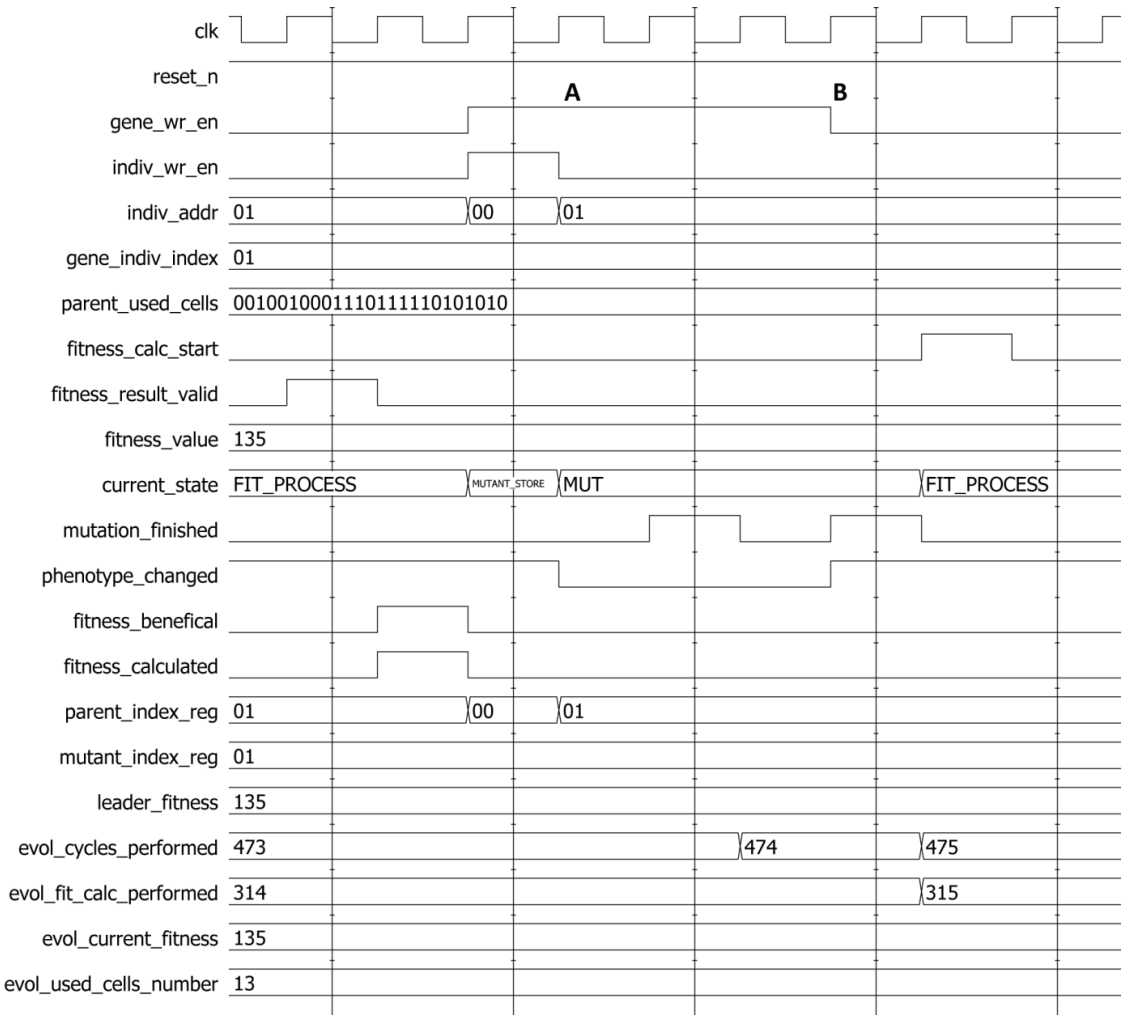


Figure 53. Simulation waves – the mutation process

Note to the Figure 53: The signal 'gene_wr_en' substitutes the signal Gene Write Enable presented in the Figure 51. The abbreviation M_S means MUTANT_STORE state.

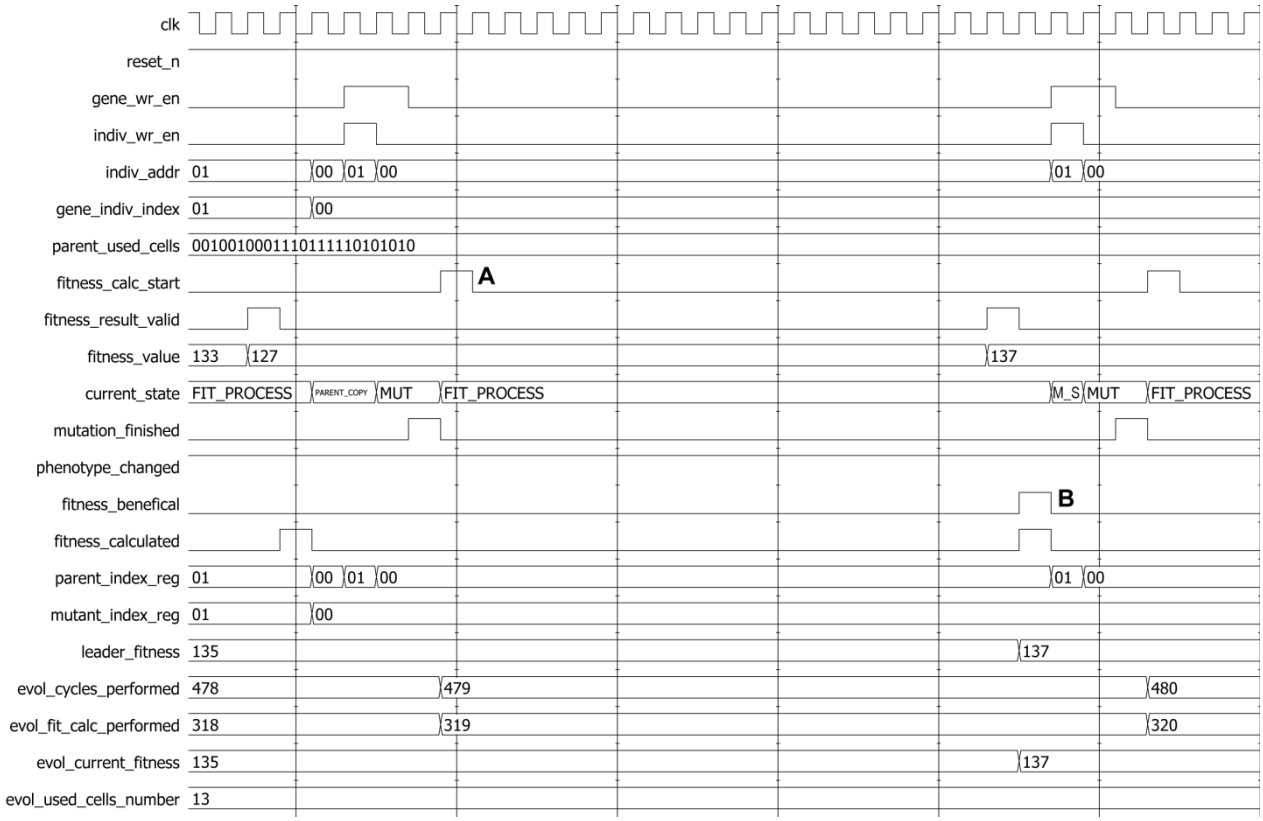


Figure 54. Simulation waves of the evolution

Note to the Figure 54: The abbreviations M_S and MUT mean MUTANT_STORE state and MUTATION respectively.

4.4.5.1 Testing of Search Algorithm

According to the previous chapters, the implementation of *CGP* was tested and demonstrated. The evolutionary design of a multiplier 3x2 bit was chosen as a benchmark. The author implemented the simple stimuli generator, ideal response generator and the fitness module that performs a fitness calculation. These additional modules were connected to the designed components in accordance with the diagram in the Figure 51.

The following algorithm parameters were used. The topology of the structure is formed by 5 columns and 5 rows. Due to the type of the benchmark, the structure implements 5 primary inputs and the same number of primary outputs. All primary inputs can feed cells/nodes in all columns. No restrictions are applied to the connectivity of primary outputs; they can be represented by any cell. The value of the mutation rate was variable in the range from 1 to 7. The *l-back* parameter takes both allowed values; it means values 1 and 2. All cells/nodes could implement one of these functions: AND, OR, XOR, identity. The data width of the *VRC* signals is 8 bits. The maximal number of generations was limited to 2,000,000. The aim of the evolutionary design was to find the full functionality

of the multiplier and the optimal solution from the point of view of the used cells/gates. For that reason, the required fitness takes the value 172. This value corresponds to the full functionality of the multiplier and 13 used cells/nodes. The fitness function is based on formula (3). Because of the *Active Gene Detector* component, it is possible to use this formula directly. Each run of the algorithm was repeated 100 times.

The Table 14 summarizes the obtained results when the *l-back* parameter was set to 1.

Table 14. Results of an experiment (HW implementation) with the *l-back* = 1

Mut. Rate	Mean # Gen.	Mean # Val.	Mean Time [ms]	VR [%]	Run Succ. [%]	Cells/Gates			
						Min	Max	Mean	Min. Achieved [%]
1	287,230	178,518	22.8	37.8	99	13	18	14.354	26
2	357,541	301,509	40.0	15.7	100	13	16	14.232	34
3	416,829	389,511	53.7	6.6	97	13	18	14.265	31
4	675,994	657,953	94.2	2.7	92	13	18	14.287	31
5	1,052,728	1,040,539	154.7	1.2	80	13	16	13.965	34
6	1,299,658	1,292,038	199.5	0.6	69	13	18	14.226	20
7	1,415,388	1,410,645	225.8	0.3	57	13	20	14.577	14

The items in the first three columns of the table were explained in the previous chapters. The *Mean Time* item expresses the run time of the algorithm in milliseconds. This time is related to the full functionality of the multiplier. The *VR* means *Valuation Reduction* and it was established in the chapter 4.3.3 by the author. The column called *Run Succ. (Run Successful)* expresses the percentage share of runs where the required functionality was achieved. The last four columns of the table show the results related to the number of used cells. There are shown the maximal, the minimal and the mean number of used cells. The item *Min. Achieved* expresses the number of successful algorithm runs where the number of used cells was equal to the minimum (it means 13 cells).

It can be seen from the obtained results that the fastest runs were performed when only one gene was mutated in the chromosome. Almost all these runs found a circuit with full functionality. The *VR* takes 37.8%. However, only 26% of runs found the minimal solution. The mutation rate equal to the value of 2 causes better results in terms of the minimal solution, but the runs take more time.

The same experiment was performed by the *Evolutionary Designer* software tool. Slightly better results were obtained. The Table 9 (subchapter 4.3.5.1) shows that the algorithm needs 159,671 fitness calculations (the *VR* was equal to 38.4% and is not presented in the

table). For all settings of the mutation rate except the value of 1, the mean number of used cells was less than 14.0. These different results are caused by the use of other number generators. The implementation in the FPGA uses only simple LFSR generators (19-bit and 11-bit).

The Table 15 shows the results when the *l-back* is equal to 2.

Table 15. Results of an experiment (HW implementation) with *l-back* = 2

Mut. Rate	Mean # Gen.	Mean # Val.	Mean Time [ms]	VR [%]	Run Succ. [%]	Cells/Gates			
						Min	Max	Mean	Min. Achieved [%]
1	138,261	82,263	10.6	40.5	100	13	15	13.5	57
2	150,616	124,607	16.6	17.3	100	13	16	13.5	61
3	210,802	195,675	27.0	7.2	100	13	15	13.4	62
4	309,272	299,741	42.9	3.1	100	13	16	13.4	66
5	507,623	501,241	74.6	1.3	99	13	15	13.5	56
6	754,612	749,626	115.7	0.7	93	13	17	13.7	44
7	1,051,454	1,047,560	167.7	0.4	81	13	17	13.9	32

From the table is clear that the obtained results are better at all points. If the mutation rate is equal to 1, the evolution is two times faster. Of course, this behaviour is to be expected. The same experiment was also performed by the software tool; the results are shown in Table 16. The software implementation gives slightly better results in general. However, if the focus is given on the first line, it is shown that the FGPA implementation was more efficient when the number of fitness calculations (*Mean #Val.)* was the considered aspect. In both implementations, the *Min. Achieved* takes the best value when the mutation rate is 4; and the worst value when the mutation rate is set to 7.

Table 16. Results of an experiment with *l-back* = 2; implemented by the software tool

Mut. Rate	Mean # Gen.	Mean # Val.	VR [%]	Run Succ. [%]	Cells/Gates			
					Min	Max	Mean	Min. Achieved [%]
1	139,394	84,874	39.1	100	13	16	13.4	67
2	128,858	108,316	15.9	100	13	16	13.3	77
3	176,470	165,062	6.5	100	13	15	13.2	80
4	221,685	215,554	2.8	100	13	15	13.2	82
5	323,096	318,983	1.3	100	13	14	13.3	74
6	510,010	507,302	0.5	99	13	15	13.3	72
7	813,700	811,661	0.3	93	13	16	13.5	62

It is also necessary to mention the FPGA resources were used for the implementations. All tests were carried out by the means of the FPGA Altera Cyclone IV. The synthesis results are described in Table 17. The implementations were tested at the maximal possible frequency at 175 MHz. The table shows the higher requirements of the implementation with *l-back* = 2.

Table 17. Summary of the synthesis results

l-back	Resources		LEs	Registers	Embedded Multipliers	Embedded Memory [bits]
1	***Total***		***4,285***	***1,560***	***1***	***688***
		Mutation	201	121	1	0
		Algorithm Controller	349	168	0	0
		Fitness Unit	154	104	0	0
		VRC (8-bit)	2,792	679	0	0
		Active Gene Detector	621	375	0	0
		Fitness Controller	43	21	0	0
		Avalon Interface	187	88	0	0
		Configuration Memory	6	0	0	688
2	***Total***		***5,462***	***1,772***	***1***	***688***
		Mutation	167	133	1	0
		Algorithm Controller	347	168	0	0
		Fitness Unit	130	104	0	0
		VRC (8-bit)	3,600	799	0	0
		Active Gene Detector	705	465	0	0
		Fitness Controller	43	21	0	0
		Avalon Interface	136	88	0	0
		Configuration Memory	6	0	0	688

5 EVOLVABLE FIR FILTER

This chapter deals with demonstration of an evolvable system which is represented by "Evolvable FIR filter".

The main goal of this experiment/project is the design of an evolvable *FIR* filter. The parameters (the impulse response) of this filter are obtained by the evolution. It means that no user intervention is needed for correct function of the filter. The standard *FIR* filter has one data input and one data output. In addition, the evolvable 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.

Figure 55. Evolvable 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 [77]. 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 as well. 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 was used. The taps of odd numbers are preferable if it is required 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 device. The *FIR* filter also has to be able to perform a quick reconfiguration if it cooperates with the evolutionary algorithm.

5.1 Evolutionary Algorithm

In this project, *Standard Genetic Algorithm* was used. It is clear that standard version of this algorithm might not be optimal for this type of application; however, only a detailed testing and analyses can show its suitability.

The algorithm uses 16 individuals. Each individual consists of 15 parameters of the *FIR* filter. It is: 15 x 8 bits = 120 bits, then 4 bits for the control of the filter output and 4 bits for the reserve are used. On the whole, the individual consists of 128 bits (16 bytes). If an offspring (max. 16), mutants (max. 16), a new population (16 individuals) and a former generation (16 individuals) are taken into account, a memory for 64 (16 x 4) individuals (1,024 bytes) is needed. Only 15 parameters are in encoded by a individual. However, the *FIR* filter can be created by 30 (only 29 are used in this project) parameters, because the filter is symmetric. The initialization of a population is made so that individuals are filled by random values.

This algorithm uses only 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.

The situation with the mutation is more difficult as the mutation can be implemented in many ways. In the 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 (according to the probability of the mutation – P_m) whether a certain byte will undergo the mutation. Remaining 3 bits determine position of the bit for the mutation. The mutation of the bit means its negation. All individuals of the population undergo the mutation process. Note that this implementation of the mutant operator causes that the P_m parameter is relative to byte (filter coefficient), not to bit (gene).

For the selection of a new population, the algorithm utilises a principle of tournament. Members are selected from the former population, offspring and mutants. The selection process picks out two random individuals, and the individual with a higher value of fitness is chosen for the new population. However, this principle does not guarantee that the individual added to the new generation will be the best one; hence, this selection is extended by elitism. Elitism is a technique which ensures moving the best individual (the leader) into the new population.

The most difficult part of the algorithm is the fitness function.

5.1.1 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 quality of a found solution of the optimization task. The fitness function can be classified into two categories: it can be static or dynamic. The kind of the function depends on a concrete application. For example, in case of the implemented evolutionary design of combinational logic circuit, the 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 adaptive behaviour is required, the fitness function must be time variable because the application has to react dynamically on the environment variable. In case of the *FIR* filter, the function has to react on the current input data of the filter. This fact affects the implementation of the evolvable component. Evolution in the environment variable runs de facto continually; in contrast to the static environment, where the evolution can be terminated if a sufficient solution is found. [10]

5.1.2 Fitness Function Calculation

It is necessary to modify the algorithm and its fitness function so that the algorithm can work with the environment variable. The dynamic fitness function has to be transformed into the static function; then the algorithm works with the static fitness function, and the adaptive character is conserved. Suitable transformation into the static function ensures that all individuals within one generation have the same evaluative criteria.

Work with dynamic fitness function was solved by means of a special sample memory – the *Samples unit*. Principle of this unit is based on shift registers. 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 sampler are stored in work registers in a fitness module (part of *Evolutionary unit*). The evaluation of the whole population within one generation processes the same samples of signals; this way equal conditions for all individuals are ensured. However, the dynamic fitness function and adaptive character of the application require other changes in the algorithm. The evaluation of individuals of the former population (generation) is needed. It is a significant difference against the static fitness function. The individuals which were not

changed will have to be evaluated too, because parameters of the fitness function can be different in comparison with last evaluation.

The fitness function of the project 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 resultant values of fitness. The best individual has the smallest value of fitness.

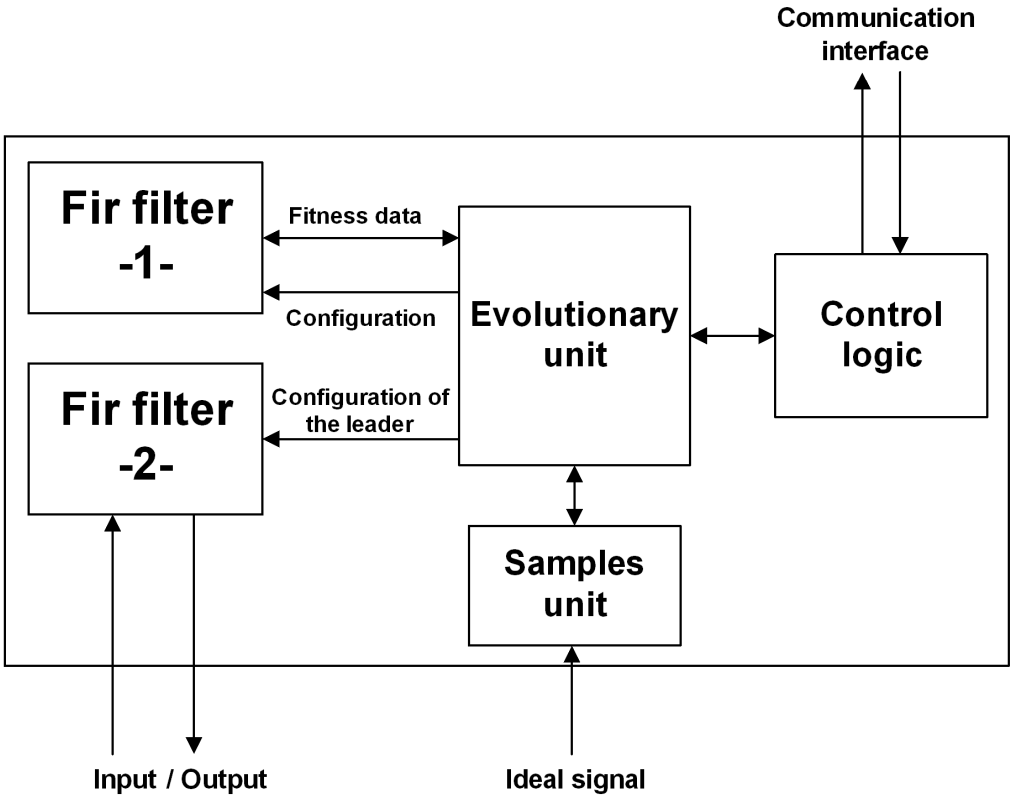


Figure 56. Diagram of the evolvable system

The required adaptive character of the application also demands changes in the structure (see the Figure 56) of evolvable system [10]. There have to be minimally two *FIR* filters in the system: the first used for calculation of the fitness function, the second one serving to the processing of 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.

5.2 Implementation

The whole system was implemented by the FPGA Cyclone II device [78]. Audio codec Wolfson WM8731 [79] was used for an input and an output of the analogue signals. The

system is designed by means of the VHDL language and Altera *Avalon* Memory Mapped interface. This conception is very profitable for the debugging and the testing. It makes it possible to very easily observe and control the process of the algorithm by the Altera NIOSII soft-core processor and its J-TAG console [80]. However, the core of the algorithm is implemented as user logic.

For storage of the individuals, offspring and mutants an embedded memory is utilised. This memory allows very fast data operations. As noted previously, 1,024 bytes of this memory were needed for all individuals. Further, the memory for the values of the fitness is also needed. Every value of fitness allocates 4 bytes (32 bits). Total, the system needs 256 bytes of embedded memory to store fitness values.

Several modules (see the Figure 57) perform the main operations of the *Standard Genetic Algorithm*. They are called: crossover, mutation, selection, fitness and elitism. These modules are controlled by means of the control register. The modules are connected with the memory by the multiplexer. This multiplexer is also controlled by the control register. Therefore, this register controls the whole process of the evolution. It is also possible to read the current fitness value of the leader. The modules and the register form component called *SGA Periphery*.

All modules are optimized so that access into the memory is exploited effectively. The pipeline structures are used in the modules. For example: the process of the crossover (the creation maximum number of the offspring – 16 individuals) needs only 21 cycles of system clock, the process of the mutation (the mutation of 256 parameters) 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: fewer samples for fitness function can be used or some parallel structure can be implemented. One generational cycle needs roughly 12,000 system cycles. If system clock is 100 MHz, the performance of the evolution is approximately 8,300 generations per second.

The system also includes the generator of pseudo-random numbers. The generator is created by the 32-bit LSFR with the polynomial $x^{32} + x^7 + x^6 + x^2 + x^0$. This generator is designed so that a random number is generated in one cycle. [81]

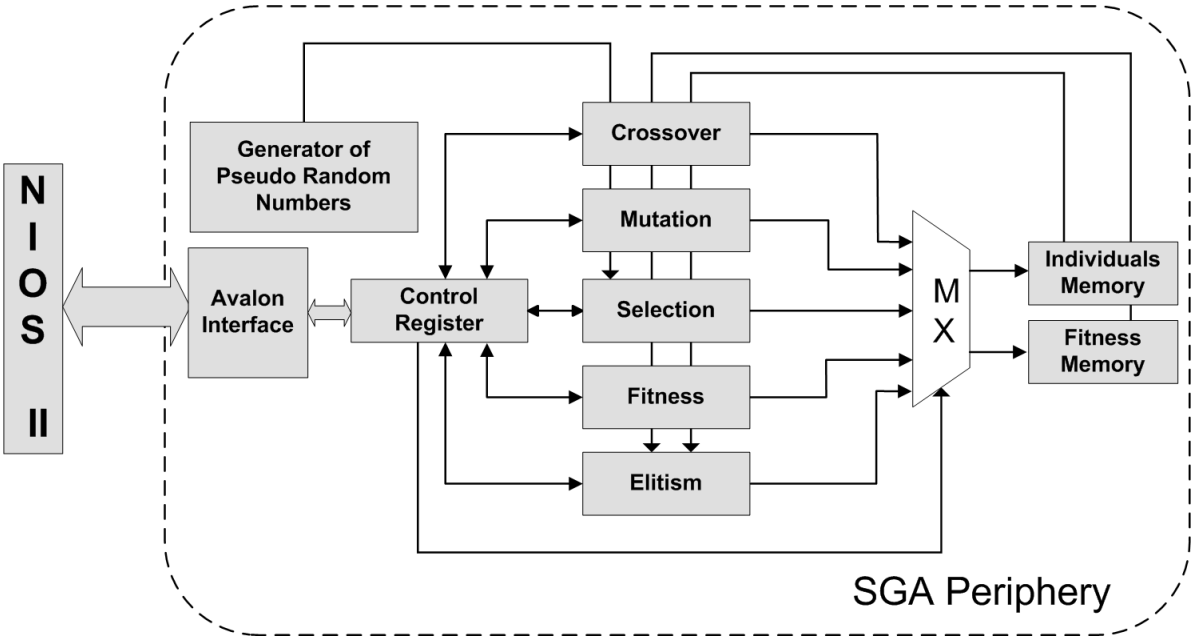


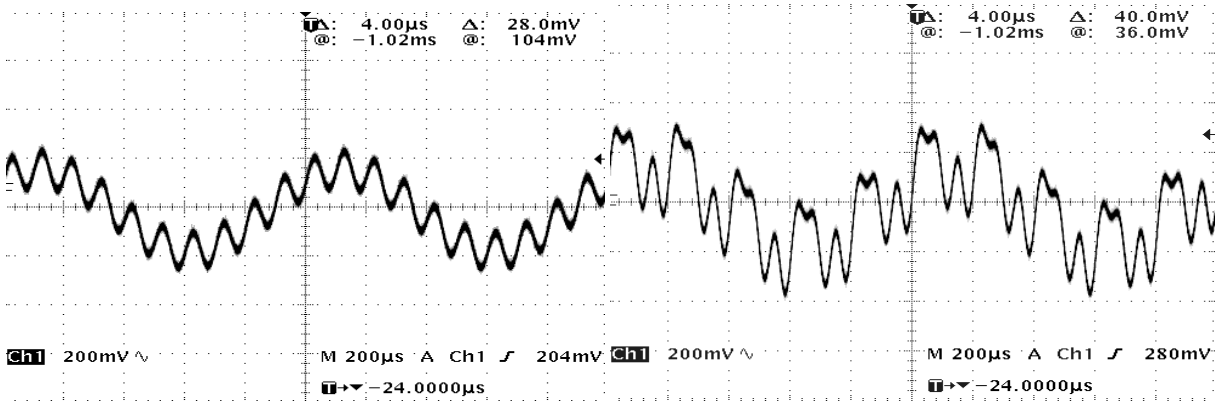
Figure 57. Standard Genetic Algorithm Periphery

5.3 Testing

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 evolvable *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.

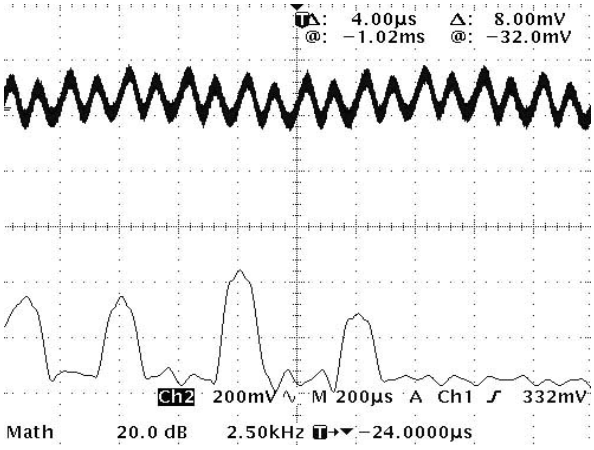
The next figures show an example of the performance of the filter. One signal with frequency 1 kHz and another with frequency 10 kHz were regarded as the useful signal (see the Figure 58-a). Other signals with frequency 5 kHz and 15 kHz were superimposed (see the Figure 58-b) on this signal. The useful signal was termed as an ideal filter output.

The sample frequency 48 kHz was used for the testing. The example of the output signal after the first generation is shown in the Figure 59-a. The good working of the filter is detectable already in the 6^{th} generation (Figure 59-b), and in the next generations the results continue improving. The Figure 59-c shows the output signal after 1,000 generations; the attenuation of the 'interference signals' is approximately 42 dB (5 kHz) and 43 dB (15 kHz). When interpreting the results, let's take into account the limited possibilities of the *FIR* filter with 29 taps.

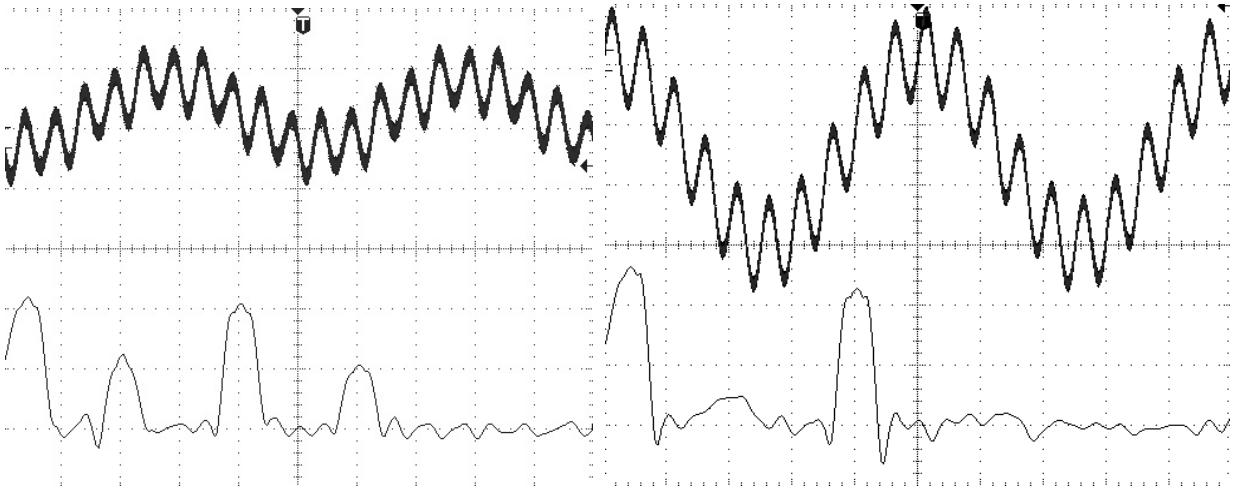


a) b)

Figure 58. Input signals – a) useful (ideal) signal; b) input signal (useful + interference signal)



a)



b) c)

Figure 59. Filter outputs and their spectrums
a) 1st gen.; fitness = 11,208; b) 6th gen.; 20dB/div; 2000mV/div
c) 1,000th gen.; fitness = 1,005; 20dB/div; 100mV/div

5.4 Influence of Evolutionary Parameters

The test application introduced in the previous section was used to analyse the influence of the evolutionary parameters on the behaviour of evolvable system. By means of quality analyses, the optimal values of parameters of the evolution can be determined. They could get better time needed for successful evolution. The testing can also point to the pertinence of chosen recombination operators.

As mentioned, a crossover operator is not suitable for the evolution in *Cartesian Genetic Programming* domain. However, the evolution of the *FIR* filter represents absolutely the different task. The chart in the Figure 60 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 1,000 times.

In the chart it can be observed that the crossover does not yield a significant improvement of evolution speed. The process of search for solution is quicker (less generations) when the P_c value is about 0.6, but it must be taken 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 (see the Figure 61), the dependence on P_m is analysed. It shows 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. Although P_m parameter is relative to bytes of a chromosome; these values are higher than generally recommended values. 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.

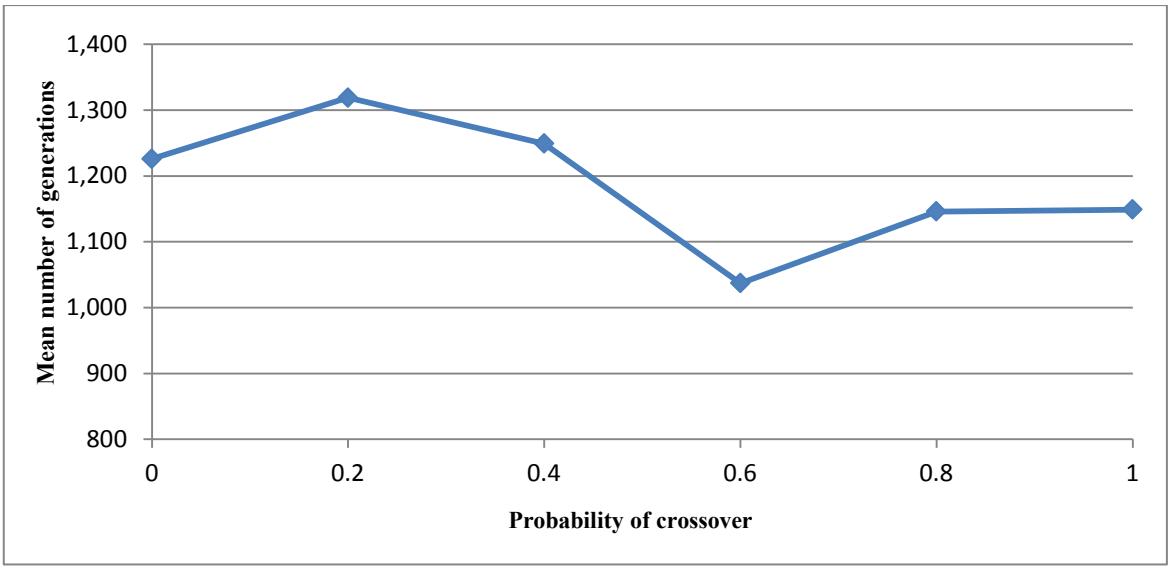


Figure 60. Dependence of the number of the generations on P_c (P_m = 0.4)

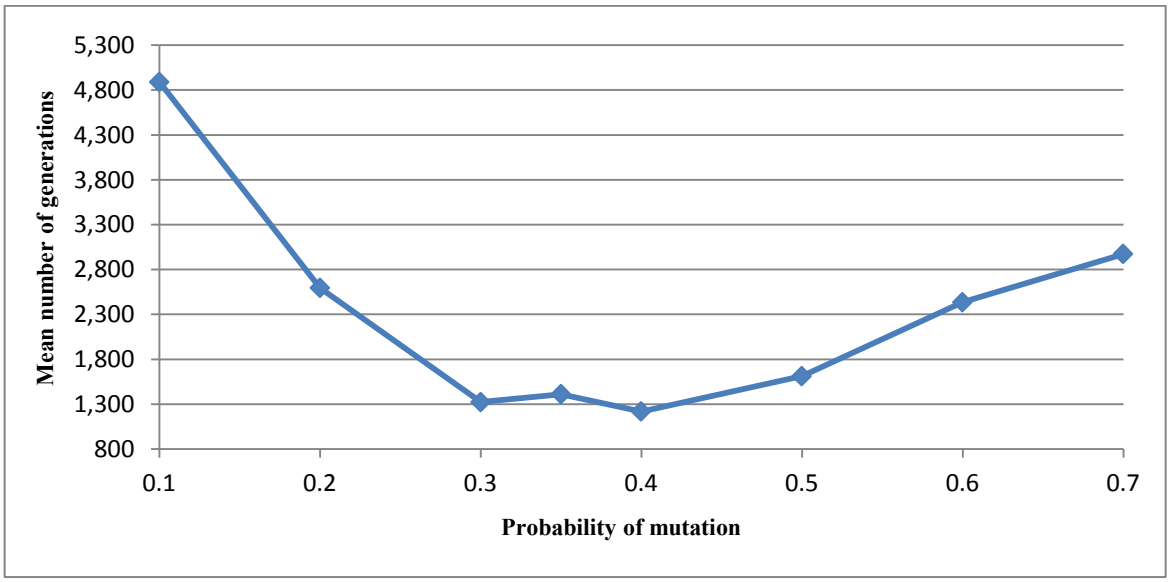


Figure 61. Dependence of the number of the generations on P_m value (P_c = 0.5)

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 swarm seek for food and if some member 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. [32] This principle was implemented and it provides markedly better results than the previous version of the crossover. It is shown in the Figure 62.

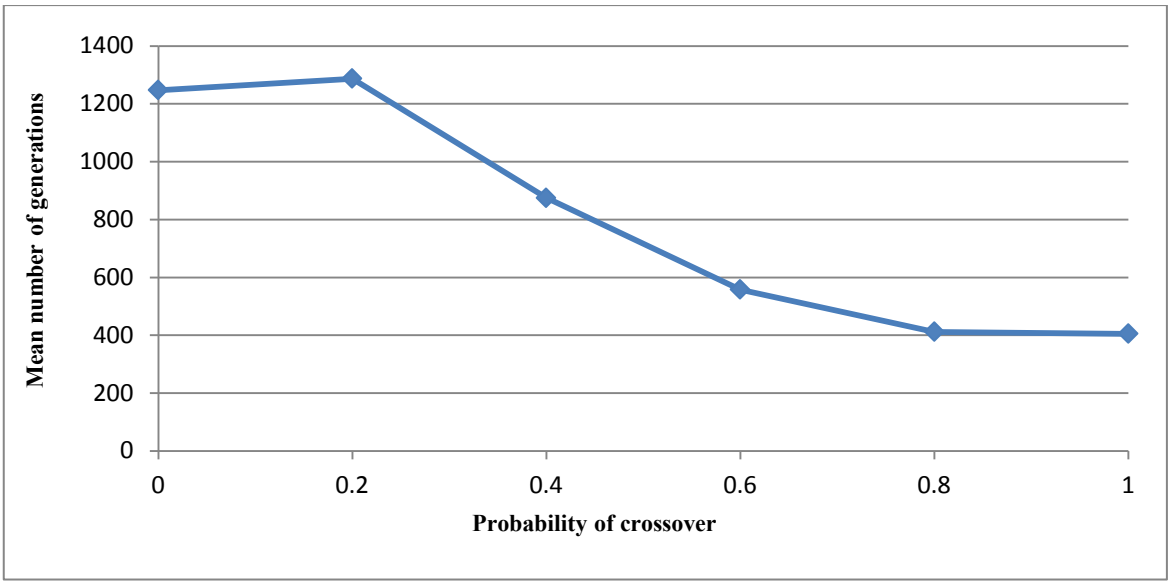


Figure 62. Dependence of the number of generations on P_c value ($P_m = 0.4$)

The results of this crossover operator 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 suppress the population diversity. This fact could negatively affect especially dynamic characteristics of the evolutionary *FIR* filter – the adaptability competence could be restricted.

6 IMAGE RECOGNITION BASED ON *N-TUPLE* NEURAL NETWORKS

This chapter deals with the special type of neural networks – *n-tuple neural networks* – and its utilization in image recognition system based on FPGA devices. It explores an *n-tuple* methodology using node 'grouping' and the possible advantages offered by this little-known technique. In cooperation with Holota, the research introduced in this chapter was published in [A1] by the author of this thesis.

6.1 Background

6.1.1 *N-tuple* Methodology

In principle, the *n-tuple* technique is equivalent to that of using *Single Layer Networks (SLNs)* consisting of 'deterministic logic nodes'. The term 'deterministic logic nodes' was originally used by Aleksander and Stonham over 30 years ago. It simply means that an *n-tuple* node can perform all 2 to the power of *n* logic functions and that, after training, it can only respond to those training patterns and, therefore, it is 'deterministic'. The logic nodes of these *SLNs* are realised by Random Access Memory (RAM). Sometimes it is described as RAM-based networks [82] in the literature. Each logic node consists of a RAM with an *n*-bit address space (see the Figure 63). The following paragraphs describe the *n-tuple* methodology in a simple way. An exact mathematical description of the *n-tuple* recognition method can be found in many articles, e.g. [26][83].

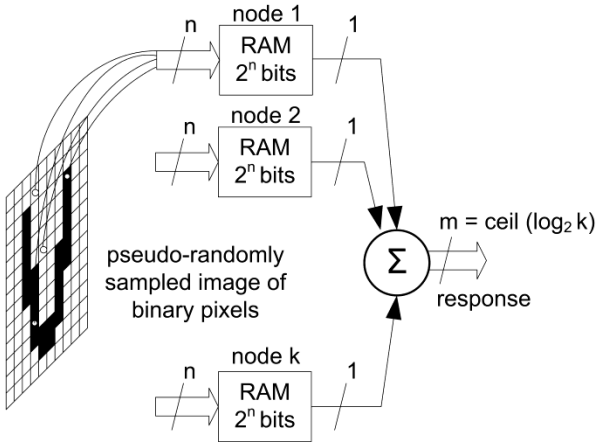


Figure 63. Single Layer Network architecture

The pattern which is applied to the address inputs consists of *n* sample points. These points are taken pseudo-randomly from the input data. The pattern applied to the input is termed

an '*n-tuple'*. All the logic nodes have to be initialised (to zero state) before the training phase. During training, each RAM stores a logic '1' to the position given by relevant sampled *n-tuple* data words. The recognition phase is based on reading the stored data from the address given by *n-tuple* data words which are sampled from the input image by the same pseudo-random sequence. During recognition, the RAM operates as a look-up table without requiring any arithmetic or logical operations. Each *SLN* (or 'discriminator') is composed of *n-tuple* nodes (the number of nodes is defined by 'k') and the response is given by the summation of all logic nodes in the layer. The bit width of response depends on the number of nodes (see Figure 63). In principle, '*n-tuple* classifier systems' operate in a multi-discriminator configuration (see Figure 64). In the training mode, each discriminator is trained on each class of patterns. The class is dedicated to one object and is formed by the set of similar images of this object (e.g. with different rotation, scale, noise). The number of discriminators is equal to the number of classes which are defined for future classification. The classification mode can be started when all discriminators have been trained. Each discriminator gives a response to an unknown input image. The input image is assigned to the class which corresponds to the discriminator with the highest response.

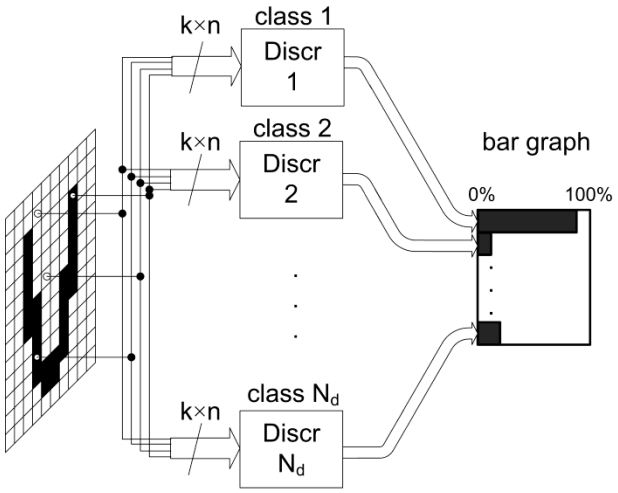


Figure 64. Multi-discriminator configuration

The discriminators' responses are usually represented either numerically (by the number of pass logic nodes or the percentage value) or as a bar graph display. In general, each bar in the bar graph display represents the degree of compliance within a given class. Usually, the responses are also displayed in the training mode. The monitoring of these responses gives an indication of the amount of training required and assists in determining possible over-training or under-training of the neural networks.

6.1.2 Network with Grouped *N-tuple* Nodes

It is possible to utilise grouping methods in these types of SLNs as in [21][23]. The effect of grouping is to increase the differences among responses to the discriminator (for the given class) and other discriminators. The principle of this method is based on creating groups of *n-tuple* nodes in each SLN. Each group consists of the given number of nodes, a summation unit, and a threshold unit (see Figure 65). The final response of each discriminator then constitutes the sum of the groups' responses.

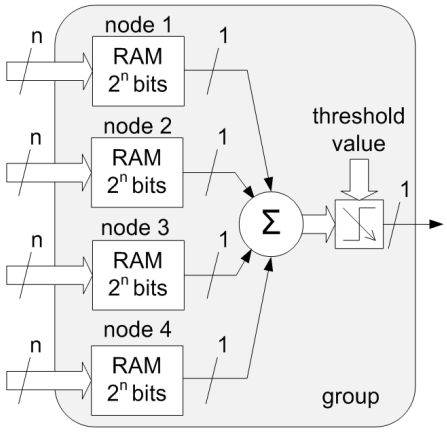


Figure 65. Principle of grouping

6.1.3 Utilisation and Memory Requirements

Generally, an *SLN* composed of *n-tuple* nodes or grouped *n-tuple* nodes can be used for the recognition of binary images. In the case of greyscale images, it is necessary to use a suitable method for converting them into binary images. In order to perform colour recognition, the networks are composed of 'trixel' *n-tuple* nodes (TNT nodes) as shown in [21][23].

The memory requirements are given by the formula:

$$M_r = \frac{N_d \,.\, 2^n \,.\, N_p}{n} \; [bits] \qquad (16);$$

where
N_d ..... *the number of discriminators,*
N_p ..... *the number of selected pixels,*
n ..... *type of tuple (n=8 => 8-tuple),*
M_r ..... *memory requirements in bits.*

The variable N_p has to be chosen in accordance with the size of the tuple node and the group size.

6.2 Hardware Implementation

6.2.1 General Description

The designed recognition system is designed as a System on the Programmable Chip (SoPC) architecture based on an FPGA device. Several Qsys [84] components form the system (see the Figure 66). The Qsys interconnect ensures connectivity and mutual communication.

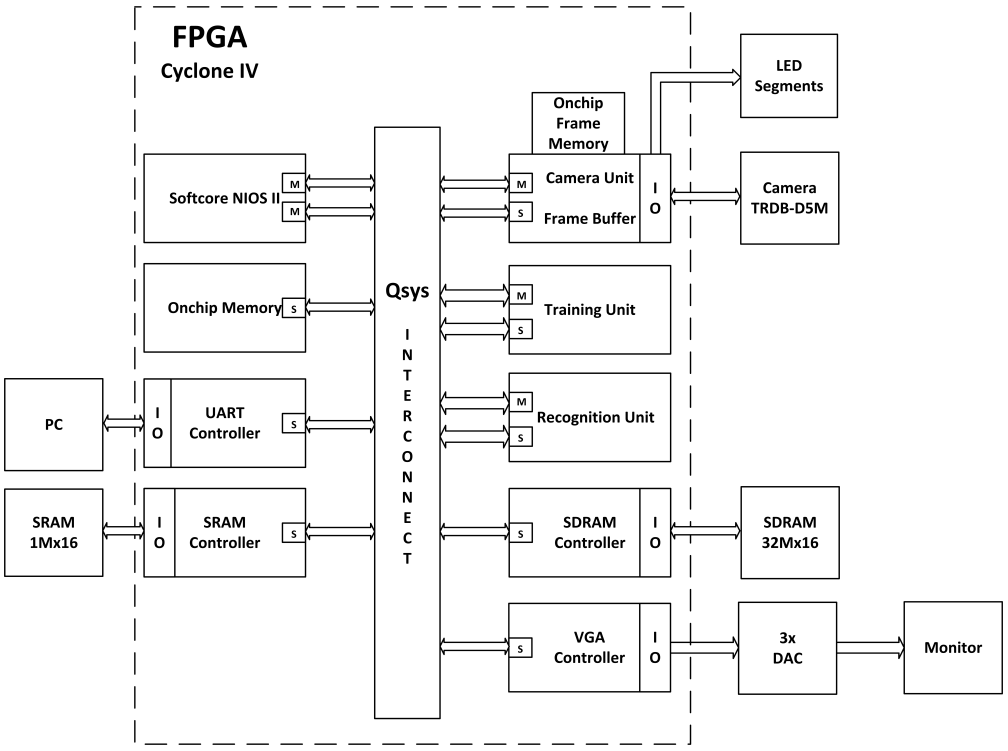


Figure 66. Block diagram of the system

The *Camera Unit* together with the *Frame Buffer* obtain image data from the camera module TRDB-D5M (5 Mega Pixel Digital Camera Package from Terasic Corp.). It also provides the conversion of the raw image data to RGB (red – green – blue) colour model and BW (black/white) format. The *Frame Buffer* selects and stores data needed for image recognition. In addition, the component may send full image data to other components in the system. The *Training Unit* and *Recognition Unit* are the next main (in the line with the *Camera Unit*) components of the system; they are discussed later. Further, the system contains two memory controllers – the *SRAM* and the *SDRAM* controller. The external SRAM memory (2MB – 1M x 16) is connected to the *SRAM Controller* and is used for storage of the neural networks' data. The latter controller – the *SDRAM Controller* – manages data transfers to the 64MB (32M x 16) SDRAM. This memory is used for video

storage in this project. Data from this memory can be displayed on the PC monitor by means of the *VGA Controller*. The *VGA Controller* works with a resolution of 800x600 at 75 frames per second (fps). The remaining three components control the training and recognition process and communicate with the supervisory system. The core of the supervisory software is implemented in the softcore CPU NIOSII; this 32-bit processor controls the operations of all the other components. This processor also communicates with the high-level system (PC) via a UART (implemented by the *UART Controller*). The *On-chip Memory* serves as the program and operational memory for the NIOSII. The special user application on the PC controls the whole image recognition process via the UART and NIOSII processor [80].

6.2.2 Camera Unit with Frame Buffer

This is the key component of the system. It obtains data from the external camera module and provides two outputs which are the data stream for the video memory and image data for *n-tuple* processing in the appropriate format. The block diagram of this component is shown in the Figure 67**.** This component is connected with the camera module TRDB-D5M via the Avalon Conduit Interface. It captures image data in RAW format (resolution 800x600) and can convert this data to the following formats: RGB 16bit, Greyscale and BW. The selected format depends on the setting of the component by means of the *Avalon Slave Port*.

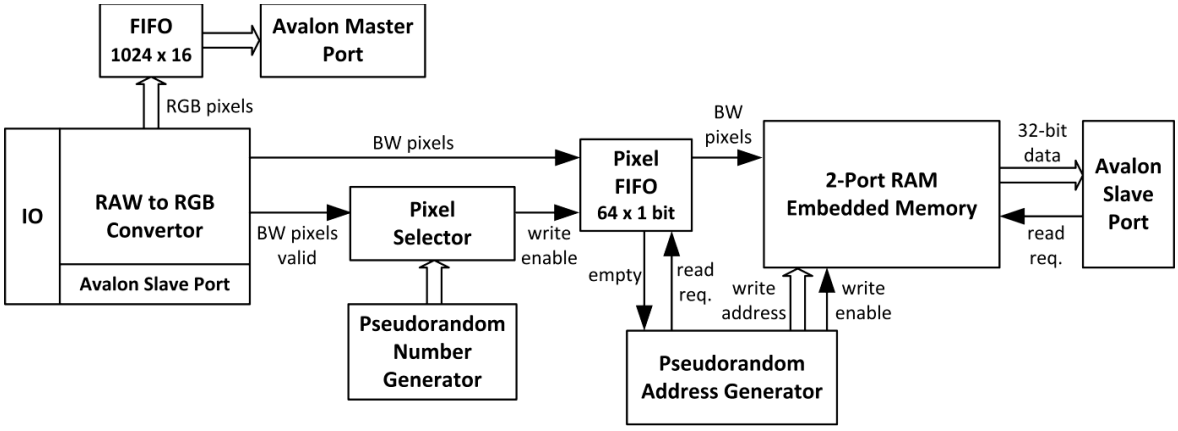


Figure 67. Camera Unit with Frame Buffer

The conversion to a BW image uses a fixed threshold that is set via the Avalon Slave Port or automatic threshold detection. The final RGB vectors enter the FIFO memory and can be sent to the video memory by means of the *Avalon Master Port*. This function makes it possible to observe the captured image.

The remaining parts of the component create an appropriate data format for subsequent processing by the *Training* and *Recognition Units*. The author of this thesis designed and implemented the new approach of image data buffering. Compared to the conventional architecture [85], this approach doesn't need a buffer for full image frame. They usually store a full frame to memory and then pixels for processing are selected, afterwards these pixels are randomly combined to form the *n-tuples*. The main disadvantage of these techniques is higher memory requirements.

The *Pixel Selector* selects pixels for the next processing stage. The full image contains 480,000 (800x600) pixels of which 5 percent are selected (i.e. 24,000 pixels). The pixel data are fed into the *Pixel FIFO*, and the *Pixel Selector* generates the 'write enable' signal which allows writing of pixels into the FIFO memory. The image frame is divided into 15 segments in which pixels are selected with variable intensity. The pixels more closely placed to the centre of the frame have preference over those near the borders.

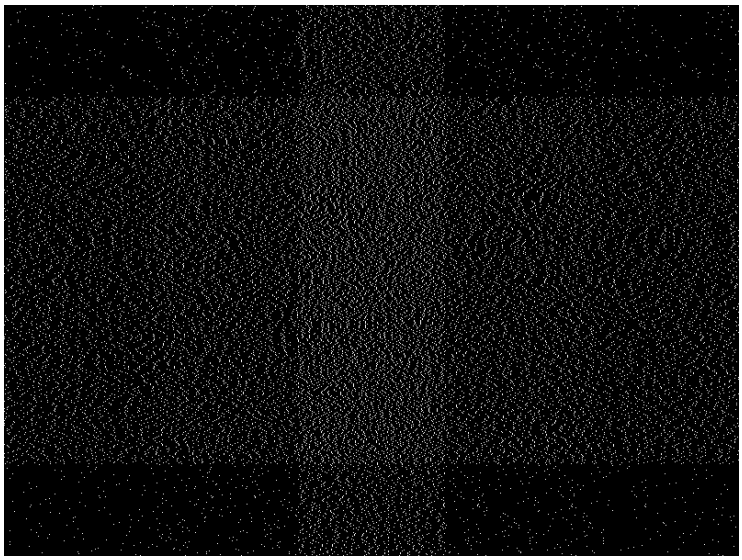


Figure 68. Selected pixel positions (white dots)

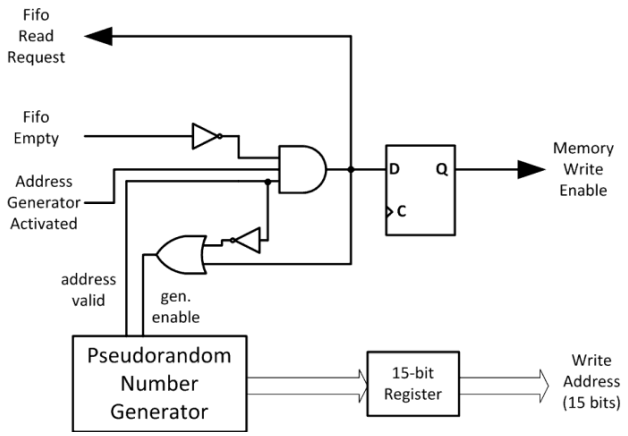


Figure 69. Pseudo-random Address Generator

The *Pixel Selector* detects the start of the frame and then selects 24,000 pixels from each frame. The optional module for performing image data processing can be inserted between the *RAW to RGB Convertor* and *Pixel FIFO*.

After selection of the pixels, it is necessary to assemble them in a pseudo-random manner before applying them to the *n-tuple* memory nodes. In this case, an *8-tuple* is used. The random composition is made by means of an embedded memory which is configured as a dual-port RAM memory with a capacity of 24,000 bits. The write port consists of a 1-bit data input and a 15-bit address. The read port consists of a 32-bit data output with a 10-bit address input. The component generates a random write address; it means that pixels are stored in random locations within the memory. The correct write address and control signals are created by the *Pseudo-random Address Generator* (Figure 69). A simple 15-bit Linear Feedback Shift Register (LFSR) with a minimal polynomial of $x^{15} + x^{14} + 1$ is used. This pseudo-random generator generates random numbers in the (modified) range of 0 to 32,766; however, correct write addresses are in the range of 0 to 23,999. For this reason, it is necessary to check the value of the address vector. The LFSR also implements a digital comparator which flags (by the signal *address valid*) whether the generated address vector is valid. Further, the generator contains a *gen. enable* signal which enables/disables its function and thereby issues the next random number (address). The writing into memory is allowed if several conditions are satisfied. If the *Pixel Fifo* is not empty, the address generator is activated and the generated address vector is correct, the memory *write enable* signal will be set to high in the next clock cycle (via the D flip-flop). Also, the generated address vector is registered by means of a 15-bit register. If the memory *write enable* signal is high, the pixel of the image will be stored at a random position in the memory. The *gen. enable* signal is high and it causes that the next random address will be released in the next clock cycle, if the generated address vector is not correct or if the *Memory Write Enable* signal will be set in the next clock cycle. In other words, the *gen. enable* signal is high if the next address vector is needed; either the current value of the address vector is not accepted or the new address vector for another memory write is demanded. This approach makes it possible to use a simple pseudo-random generator implemented as a LFSR. If the generator gives incorrect values, the pixel data are held in the *Pixel Fifo* memory. It is necessary to note that the implemented logic works with a 110-MHz clock in this project. It is a higher frequency than 50MHz which is the maximum pixel rate. It must also be taken into account that only 5% of pixels are processed in this way; thus the delays

caused by waiting for the correct address vector are irrelevant. The pixels are stored in memory and form three thousand *8-tuple* nodes. As mentioned earlier, the read data memory port is 32 bits wide which means 750 double-words of data. This configuration of memory was chosen for more effective data transfers inside the system; four *8-tuples* can be read within one read transfer operation. The read port of the memory is connected to the *Qsys Interconnect* via the *Avalon Slave Port*. It implements an auto-increment read pointer in which the whole memory is mapped as one 32-bit word in the memory space and a new read address is released automatically. This function simplifies the address calculation to other components in the system.

The main benefit of the *Camera Unit with Frame Buffer* component is that a buffer for a full image is not needed. It represents a significant reduction of memory requirements. The amount of reduction is dependent on the coverage of an input image. In our case, the reduction of memory requirements is approx. 95% within 5% coverage. Another advantage is lower time consumption because the random selection and forming of image pixels are performed already during data storing.

6.2.3 Training Unit

The purpose of this component is to train neural networks consisting of *8-tuple* nodes (each node composed of 8 image pixels). The component reads *8-tuples* from the embedded memory in the *Camera Unit* that define the memory position (for neural network data) where logic high will be stored. Other memory positions remain unchanged. Before training the first image frame, it is necessary to clear the neural networks memory to ensure logic low value on all positions. In this project, neural network data is stored in an external SRAM memory with a 16-bit memory data bus. The memory requirements for one discriminator are 768,000 (256 x 3,000) bits, where 3,000 is the number of *8-tuple* nodes each of which requiring 256 bits. On account of a more effective access to the memory, the following structure of the neural network data in the memory was designed – see Table 18. The data for a particular discriminator (class) is not stored in the continuous memory area but the ‘column structure’. Data of particular discriminators are organized as columns within the memory area. For example: data for the discriminator (class) 1 is stored in the LSBs of the memory words, class 2 in bits with index 1, and so on. In Table 18, it is shown that 512 (0x200) bytes are needed for keeping one *n-tuple* of sixteen discriminators. For that reason, the memory offset for the next *n-tuple* is always 0x200. Because of this

designed memory structure, the formula for the memory requirements (16) has to be modified in the following way:

$$M_r = \frac{f_{mo} \cdot W_b \cdot 2^n \cdot N_p}{n} \; [bits] \tag{17};$$

where W_b ..... *width of memory data bus in bits,*

N_p ..... *the number of selected pixels,*

n ..... *size of tuple (n=8 => 8-tuple),*

M_r ..... *memory requirements in bits,*

f_{mo} ..... *memory organization factor.*

Memory organization factor is defined as:

$$min\left\{f_{mo} \in \mathbb{N} \;\middle|\; f_{mo} \geq \frac{N_d}{W_b}\right\} \tag{18};$$

where W_b ..... *width of memory data bus in bits,*

N_d ..... *the number of discriminators.*

From these two formulas, it is obvious that the memory is fully and effectively utilised if the number of discriminators is an integral multiple of the memory data bus width.

The author of this thesis assumed the use of 16 classes; this means that 12.288 Mbits of memory are needed. The process of training is the following. At first, *8-tuples* are read (4 x *8-tuples* in one 32-bit vector) from the embedded memory. The word on the position defined by the value of the *8-tuple* and its order is read from the SRAM memory. In this word, the appropriate bit (determined by the chosen class) is set to high. Afterwards, the modified word is written back on its position. Training is complete once all *8-tuples* (in this case 3,000 => 750 vectors) are read.

Table 18. Memory structure

Word Index	**Memory Address**					**Class**
	0x000000	***0x000200***	***0x000400***	**…**	***0x176E00***	
0 (LSB)	1st tuple	2nd tuple	3rd tuple	…	3,000th tuple	***1***
1	1st tuple	2nd tuple	3rd tuple	…	3,000th tuple	***2***
2	1st tuple	2nd tuple	3rd tuple	…	3,000th tuple	***3***
3	1st tuple	2nd tuple	3rd tuple	…	3,000th tuple	***4***
…	1st tuple	2nd tuple	3rd tuple	…	3,000th tuple	**…**
15 (MSB)	1st tuple	2nd tuple	3rd tuple	…	3,000th tuple	***16***

In addition, the component can determine the networks' responses on the training data. This response may be useful for training process control.

The component is controlled by means of control registers via the Avalon MM interface. By these registers, the user can start training and setup values (grouping and group threshold) for response calculations.

6.2.4 Recognition Unit

This component determines the responses of the neural networks based on the currently unknown images. As well as the *Training Unit*, it reads *n-tuple* data from the embedded memory. The data are loaded into a small FIFO memory; this means that the delay caused by their reading occurs only at the beginning of the recognition process. The rest of the process is determined only with the latency of the SRAM (memory for neural networks data). The component's Avalon ports are able to work with any type of memory (an appropriate controller is needed). Responses of (up to) 16 networks may be calculated in parallel because of the designed data structure of the memory (introduced in previous chapter). The word from the SRAM memory represents neural network data for 16 discriminators (depends on the width of the memory bus). Hence, only one read operation is required for each *n-tuple*. From this, the time needed for obtaining the responses of W_b (width of memory data bus in bits) classes can be derived. The requirements T_r recalculated to this number of discriminators is given by the formula:

$$T_r = \frac{(M_l \,.\, N_n) \; + \; 8}{f_s} \; [\mu s] \qquad (19);$$

where M_l ..... *latency of reading from memory (in clock cycles),*

N_n ..... *the number of n-tuples,*

f_s ..... *frequency of system clock (in MHz).*

Note to the formula (19): Constant 8 represents latency of component's pipeline.

It follows that the time needed for evaluation of one discriminator is the same as the time for the evaluation of W_b discriminators. For that reason, the real time consumption related to one discriminator depends also on a relationship between the number of discriminators and the width of the memory data bus.

The final computed responses are obtainable via the component's response registers which are mapped by means of the Avalon Memory Mapped Interface. The grouping is supported

by the component as well. The component contains a few control registers for the component's setting. The desired group size and group threshold can be set; the permitted range is from 1 (without grouping) up to 15. However, it is necessary to take into account that the group size must be a divisor of the total number of used *n-tuples* (in given class). In this project, three thousand *8-tuples* are used and this means that grouping factors of 7, 9, 11, 13 and 14 are not utilizable. The value of the grouping factor has no impact on the recognition speed.

6.2.5 System Control

As mentioned in section 6.2.1, the whole system is controlled by the softcore CPU NIOSII. The firmware only controls the operation of the components and ensures communication with the supervisory system (the PC in this case) via the Avalon MM Interface. However, all sophisticated operations related to image processing and *n-tuple* operations are performed by the above introduced components. This means that no CPU's computing time is needed for these operations. The utilization of a softcore processor is very profitable and a modern way to control the components in the FPGA devices. It enables simpler debugging and testing of the system. The processor also can perform more complex control algorithms which can be difficult for a user. This was exploited to control the training process when training on real images from the camera (not during the test images training). An algorithm was tested so that the processor could terminate the training process if defined conditions were satisfied. The use of two termination conditions was tested. The first condition limited the maximum number of frames. Secondly, the training can be terminated if a defined number of successive frames yield responses which are equal or higher to the defined response value. The parameters of this algorithm may be set by the supervisory system. Of course, this algorithm might be implemented by the supervisory system as well. The recognition process is very simple; it can be described in a few steps:

1) The supervisory system sends command requiring recognition. It also transfers grouping and group threshold parameters.
2) The CPU sends a command to the *Camera Unit* in order to capture a frame from the camera and process it to a suitable form for the recognition process.
3) The CPU sets the grouping parameters and starts the recognition process by setting the start bit in the control register of the *Recognition Unit*.

4) The CPU detects activity of the *Recognition Unit* by means of its status register. If the recognition process is finished, the CPU reads the responses and sends them to the superior system. Afterwards, the process can continue by step 1) or be finished.

The process of training is very similar; instead of the *Recognition Unit*, the *Training Unit* is used.

6.3 Experimental Results

In this section, a real-world recognition task was used to test the designed HW system. The performed tests had two main goals. The first objective was to verify the designed HW system. The second objective was to present the possibilities of system utilization in a real recognition task and to show the influence of different system settings.

6.3.1 Description of Recognition Task

As mentioned above, a real-world recognition task was chosen. This task was road signs recognition which is a really difficult problem. The presented system should be a powerful part of a more complex system for the solution of this problem. The main task of the recognition system was to classify the images of signs with a slightly different position, rotation and size. So the supervisory system should include processing for the sign detection and processing for the normalization of position, rotation and size.

6.3.2 Database of Input Images

For the presented tests, artificially generated images were used for testing due to the fact that one of the tasks was to verify the implementation of neural networks in HW. For that reason, a program named *SourceImageGen* for generating the datasets of images with different positions and rotation was created. The test database included 11 classes – road signs (see Figure 70). The resolution of source images was 800x600 pixels.

There were six datasets for each class which differed in the position of the sign and its angle of rotation. These datasets were generated by *SourceImageGen* and the parameters for each dataset are shown in Table 19. The program output are the binary files with pseudo-randomly mapped and thresholded pixels, which are used for recognition from the image of a road sign. This feature significantly reduces the amount of data which have to be transferred to the HW system. The program performs the same pseudo-random selection and forms the data in the same way as in HW. That ensures the possibility to upload binary

data directly to the HW system memory (*2-port RAM Embedded Memory* in *Camera Unit* – see Figure 67) through the J-TAG interface.

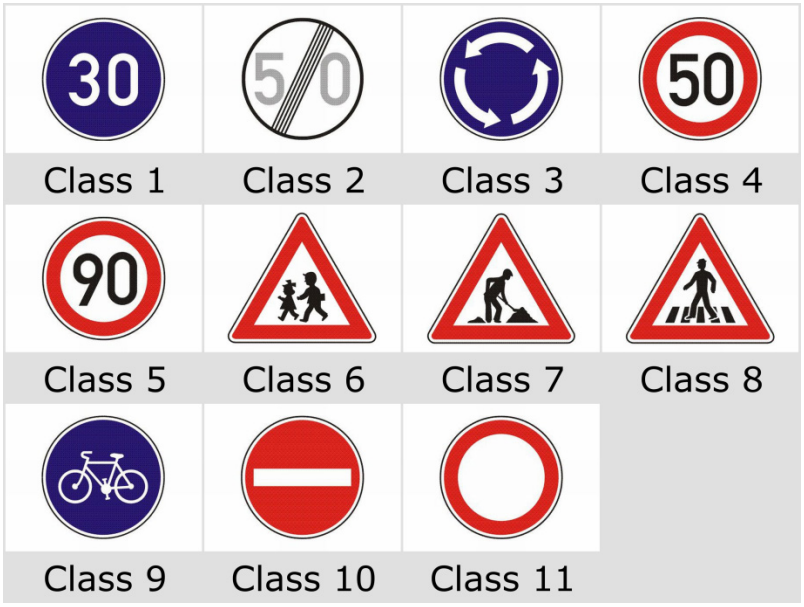


Figure 70. Road sign classes

Table 19. Datasets parameters

Dataset	Parameters		
	Position	***Rotation***	***No. of variation***
1	constant	±5°, step 1°	11
2	constant	±4.5°, step 1°	10
3	±2px, step 1px	constant	25
4	±3px, step 2px	constant	16
5	±1px, step 1px	±5°, step 1°	99
6	±1px, step 1px	±4.5°, step 1°	90

6.3.3 Tests and Results

The tests can be divided into three groups. The first group includes the datasets 1 and 2 where the angle of rotation was varied. The second group works with the datasets 3 and 4 where the position was varied. The last group used the datasets 5 and 6 where both variations were recombined. Table 19 shows that the differences between the two datasets in the same group were small. The reason for this was the possibility to use different datasets in the training and classification modes. A detailed overview of the tests is shown in Table 20. Each test included several measurements with different neural network settings – G1T1 (Group size=1, group Threshold=1), G4T1, G4T2, G4T3, G4T4, G8T8, G15T13.

Tests 1, 3 and 5 were performed in order to verify the neural network implementation by the FPGA device. According to the theoretical assumptions, all discriminators must have a 100% response for given classes in the cases where the same datasets are utilised in both modes. This verification was successful for all classes and tests.

Table 20. Summary of tests

Test	Used Datasets	
	Training Mode	***Classification Mode***
1	Dataset 1	Dataset 1
2	Dataset 1	Dataset 2
3	Dataset 3	Dataset 3
4	Dataset 3	Dataset 4
5	Dataset 5	Dataset 5
6	Dataset 5	Dataset 6

Test 6 was chosen for the illustration of results which are reached by this recognition system. For the illustration of the obtained results, a bar graph is presented in the Figure 71. It shows the maximum and minimum levels of responses in the case that input images belong to the class 4 (Speed limit – 50) and G8T8 neural network settings. The figure indicates the similarity between Class 4 and Class 5 (Speed limit – 90) but the difference between the minimum response of the Class 4 discriminator and the maximum response of the Class 5 discriminator is still sufficient for reliable recognition.

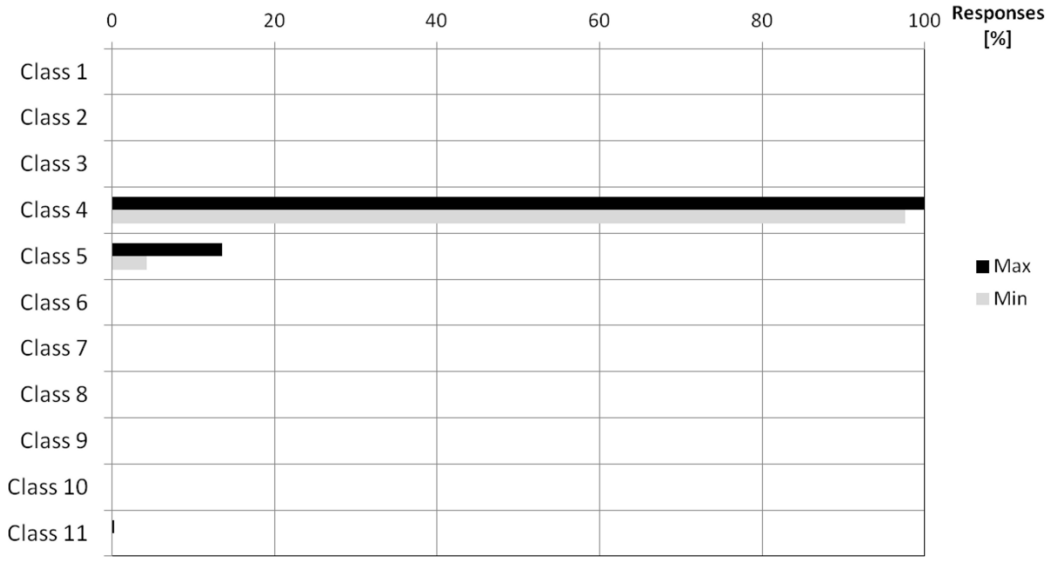


Figure 71. Responses for ‘Class 4’ images and G8T8 configuration

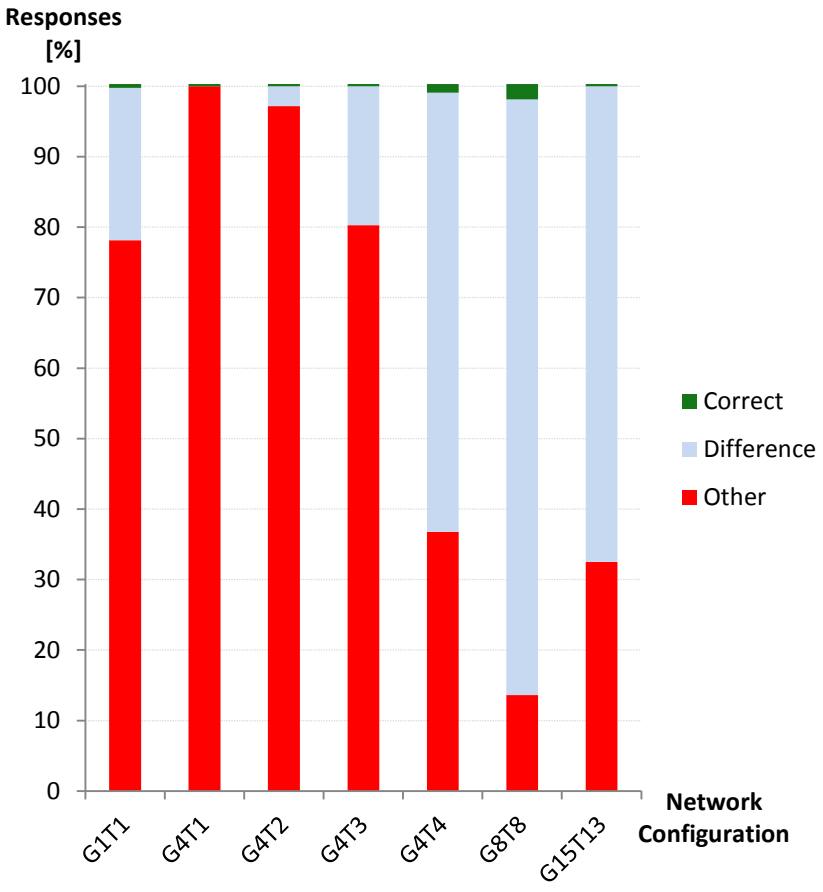


Figure 72. Levels of responses for different neural network settings

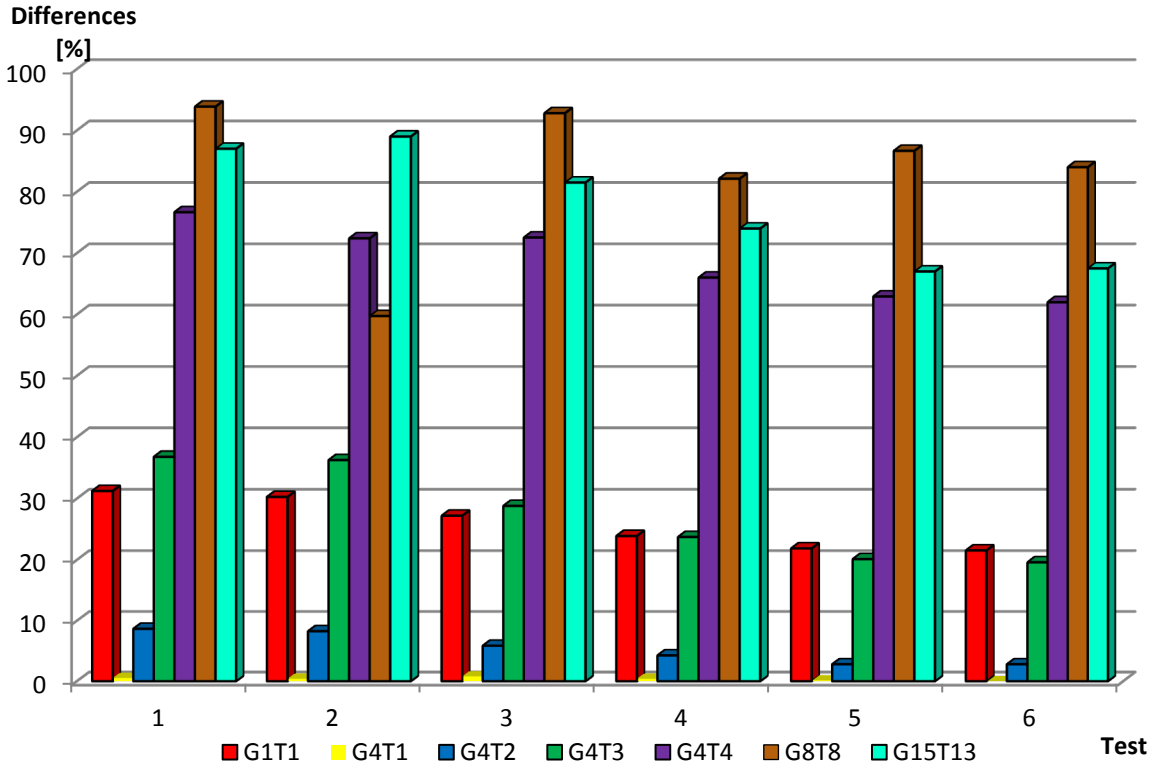


Figure 73. Comparison of minimum differences

The bar graph in the Figure 72 shows the levels of responses for different neural network settings. Each column displays the range of discriminator's responses to correct class (green), to other classes (red) and the differences (gray). By comparing of each column, it is possible to observe a benefit of grouping method with a higher group threshold against grouping with a too low group threshold or without grouping (G1T1). On the other hand, too high group threshold in combination with a higher group size can cause significant reduction of responses to correct class. It means that the differences between correct and other classes can be decreased.

The second bar graph (Figure 73) shows the differences between responses to correct and other classes for different neural network settings (see the legend of the graph) and for all tests. The bar graph clearly shows that the differences are really high for appropriate neural network settings and, therefore, the designed system could be successfully used for road sign recognition system in the future.

It is impossible to say generally which network setting is the best or acceptable. The suitable network parameters are dependent on many factors like target application, quality of image acquisition (noise, clutter, etc.), similarity of classes, and others.

6.3.4 System Performance

This part specifies the time-consumption and the memory requirements of the recognition process. Initially, the parameters of HW system are summarized. Eleven test classes were used. Each image consisted of 480,000 pixels, matching the resolution of 800x600 pixels. For the recognition process, 5% of pixels were selected; this represents 24,000 pixels. If the formula (16) is applied to these values, the memory requirement is 8.448Mb. This value is the minimum needed for representation of the discriminators. However, because of the exploited memory organization introduced in the chapter 6.2, the formula (17) must be used. By this formula, 12.288Mb of memory is required. This value is equal to that using 16 discriminators. The difference between these two values represents an overhead of the memory organization. It can seem to be markedly disadvantageous. However, it is only a trade-off between the memory requirements and the time required to obtain the responses. If a large number of discriminators are assumed, then this overhead is marginal.

The SRAM memory with a 16-bit data bus and a reading latency of 3 matched clock cycles was used. By using formula (19), the response times for the discriminators can be calculated; this is 81.89μs from 1 up to 16 discriminators. If 11 discriminators are used,

then the time requirement for 1 discriminator is longer and this means that a time of 81.89μs is required to obtain all 11 responses. This time doesn't include the time needed to control the components. In theory (image processing is not taken into account), the designed system may evaluate over 12 thousand unknown images per second. The same number of images can be evaluated even if 16 discriminators are used. From another point of view, the system with a frame rate of 1 fps may obtain responses of up to roughly 195 thousand discriminators per second. Indeed, enough memory space is assumed.

The following chart (the Figure 74) generally summarizes the previous two paragraphs. It shows the trend of two coefficients (defined by the author) depending on the number of classes. The first of them is the *Coefficient of Acceleration (CA)*; it means the speed benefit of the designed memory organization. In other words, it expresses the ratio between the number of memory read operations needed for recognition not using (sequential organization is assumed) and using our designed memory organization. The latter coefficient is the *Coefficient of Memory Requirements (CMR)*. It expresses the ratio between memory requirement needed for author's organization and for classical sequential memory organization. It is clear from the chart that developed memory organization is beneficial for the cases where the number of classes is in multiples of bus width or for larger numbers of classes.

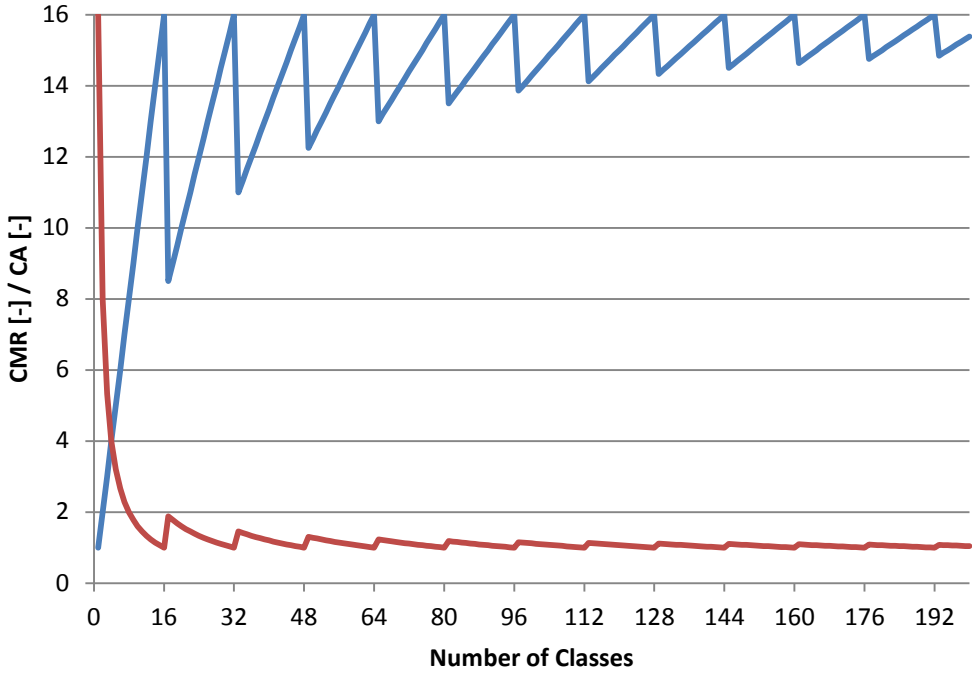


Figure 74. Coefficients of Memory Requirements (CMR - red) and Acceleration (CA - blue)

If a large number of classes is considered, we obtain the coefficient of acceleration approaching 16, and the coefficient of memory requirement approximately 1. It may be said that for an infinite number of classes, the speed benefit of our memory organization is equal to the width (16 in our case) of used memory; memory requirements remain unchanged (identical with sequential organization). However, this is only theoretical consideration.

The training process takes a significantly longer time. The *Training Unit* has to perform 3,000 read-modify-write cycles. In the experiments, the used SRAM memory has a writing latency of 4 clock cycles. In total, 21,007 clock cycles are needed. This matches approximately to 190μs.

6.3.5 FPGA Resources

The FPGA resources are acceptable. The *Recognition Unit* needs 905 logic elements (LEs), the *Training Unit* only 241 LEs; each unit consumes 128 bits of embedded memory. In comparison with these, the *Camera Unit* is the most demanding; it requires 1,784 LEs and 60,302 bits of memory. However, these values include a component part for streaming the full image to the SDRAM memory to display it. As long as this functionality is not needed, some resources could be saved. The whole system (shown in the Figure 66) consumes 7,125 LEs. The maximal frequency of this design is approximately 130 MHz. All tests and measurements were performed on a Cyclone IV device from Altera Corp. The development kit DE2-115 from Terasic Corp. was exploited.

6.4 Comparison with Other Methods and Implementations

This section presents and compares the obtained results in two domains: in recognition performance domain and from the point of view of the processing speed. For comparison, the following conventional classification algorithms were chosen:

- Nearest neighbour algorithm
- Minimum mean distance algorithm
- *K-nearest* neighbour algorithm

These methods were implemented in software by the National Instruments Vision Builder for Automated Inspections (NI VBAI). All mentioned methods used the sum distance

metrics (city block metrics). In the case of the *k-nearest* neighbour algorithm, the tests with three different *k*-parameters were performed.

Furthermore the special software application in MATLAB was programmed. It implements the image recognition based on the *n-tuple* method by the same way as the hardware described above. The results obtained by this application can be used to verify the presented hardware implementation and to compare software and hardware solutions. The hardware system introduced in [29], which is based on the same *n-tuple* method, was chosen to compare system the performance of the two different architectures.

The benchmarks use the Test 2 which is described in Table 20. In this kind of test the different datasets are exploited for the training phase (Dataset 1) and the classification phase (Dataset 2). Each run of the test was repeated 1,000 times and the results were averaged in order to obtain relevant data.

Table 21 summarizes the results of the benchmarks. It is evident that the time requirements of the classification algorithms realized by NI VBAI are almost identical. The achieved speeds are around 170 fps. In terms of recognition performance, the minimum mean distance algorithm reached the smallest false rate (FR) from the tested conventional methods but it still didn't reach 0% like the *n-tuple* method. The misclassification was observed mainly in the cases of these road signs: speed limit 50 / speed limit 90 or children / pedestrian crossing.

The software implementation of the *n-tuple* method in MATLAB produced the same responses like the HW realisation but the time requirements were significantly higher. All software tests were performed on a PC Intel(R) Core2 Quad CPU Q9550 @ 2.83GHz, 3GB RAM.

Table 21 also presents the results of two hardware implementations. However, it is necessary to note that the speed of the HW realisation by Bonato [29] is estimated for the same configuration (number of *n-tuple* nodes). This HW architecture is based on the use of the embedded memory inside the FPGA; it makes it possible to implement memory configuration optimized for given *n-tuple* network.

From the values in Table 21 it is clear that the time requirements of Bonato's architecture are slightly better. However, the amount of embedded memory is significantly limited and it doesn't allow the implementation of larger number of classes and higher input image resolution. The novel memory organization presented in this thesis solves this disadvantage

and makes it possible to use a conventional external memory with similar time requirements.

Table 21. Comparison of speeds and false rates

Tools	Classification Algorithm	Parameters	Speed [fps]	FR[%]
NI VBAI	nearest neighbour	Sum distance metrics	169	7.3
	Minimum Mean Distance	Sum distance metrics	169	1.8
	k-nearest neighbour	Sum distance metrics, k=3	173	6.3
		Sum distance metrics, k=6	170	9.1
		Sum distance metrics, k=10	168	8.2
Matlab	*n-tuple* method	n=8, coverage=5%	37	0
HW-Burian	*n-tuple* method	n=8, coverage=5%, mem. read latency = 3	12,211	0
HW-Bonato [29]	*n-tuple* method	n=8, coverage=5%	12,247	0

7 CONCLUSION

- ***Cartesian Genetic Programming***

  Cartesian Genetic Programming (CGP) presents the main topic of this thesis. A significant emphasis was placed on the reduction of fitness function calculations. The author performed an analysis related to the silent mutation phenomenon. According to this feature of *CGP*, the modifications of the algorithm were made so that the fitness function is calculated only when a change in mutant's phenotype occurs. The influence of these modifications was tested and analysed by means of benchmarks – evolutionary designs of multipliers. The modified version of *CGP* may reduce the number of fitness calculations very markedly. As it was proved by the performed experiments, the reduction can achieve up to 40% in some cases. The results of the experiments are summarized in the tables in the section 4.3.3. The degree of this reduction depends on the chromosome length, the value of mutation rate and, naturally, on the designed circuit. These dependences are discussed and expressed by several formulas. For the sake of the obtained results, the author also implemented and tested the *CGP* that produces only one mutant; it applies the search algorithm *(1+1)*. Indeed, the algorithm also evaluates only mutants with changed phenotype and was called *CCGP (Compact CGP)*. If the number of performed fitness calculations is considered to be the main viewpoint, the *CCGP* provides better results than the often used search algorithm *(1+4)* in relation to all performed benchmarks.

  All experiments were executed by the software tool called *Evolutionary Designer* which had been created by the author. This tool makes it possible to design digital combinational circuits by means of *CGP*. It can convert the result of the evolutionary design to VHDL source codes. The tool also implements VHDL source codes of the *Virtual Reconfigurable Circuit (VRC);* it can be used for the implementation of *CGP* by an FPGA device.

  The implementation of the algorithm by an FPGA device is designed so that the number of fitness calculations is reduced by the same way as it was presented in the previous analyses. Note that the implemented algorithm and structures imply the definition of *CGP* very exactly; the author uses no simplification – the full integer representation is used. The mutation produces valid integer genes. A special component – the *Active Genes Detector* – detects active genes of a parent

chromosome/genotype and can infer active or inactive mutations very fast. By means of this component, the algorithm calculates the fitness only when it is indispensable. The component also provides the number of used cells/nodes in the genotype; it makes it possible to optimise the size of the final circuit by the same way as in the software implementation. The designed *VRC* implements a structure for the *l-back* parameter taking values 1 or 2. It also supports more than one program/primary output and implements interfaces for data needed for the fitness calculations.

The functionalities of all designed components were demonstrated on the evolutionary design of a multiplier. The reduction of fitness calculation was very similar to the reduction obtained by the software tool. The evolutionary design of the test task – 3x2 bit multiplier – was finished in 10ms in average. The results are summarized in the Table 15.

It is necessary to note that the main goal of the implementation were not the achieved times of the evolution of a benchmark; however, the design system has shown that the detection of active genes can be implemented in the logic. By means of this detection, the number of performed fitness calculations is reduced. It is obvious that this feature is beneficial especially in real-world application when the fitness calculation is very time-consuming.

The components and the constructions presented in this thesis can be modified and re-used so that they form the required embedded evolutionary circuit design or evolvable system.

- ***Evolvable FIR***

  The design of the evolvable *FIR* filter serves as a demonstration of the evolvable system implemented by an FPGA device. The system is based on the *Standard Generic Algorithm* and contains two *FIR* filters. The first one is used for the calculation of the fitness function, the second one for the processing of the input signal. The evolvable *FIR* filter is able to change its impulse response and the filter type. The filter uses the fitness function based on the shape of the signal in the time domain. Tests showed that the one-point crossover is not suitable for the evolvable filter. For that reason, the crossover based on ‘averaging with a leader’ was implemented and tested. This kind of operator provides better results. The presented

evolvable system can be used in the projects where it is needed to change impulse response of a *FIR* filter and an ideal response is available. Of course, it is not limited only on the use of a *FIR* filter.

- ***N-tuple Neural Networks***

  A new hardware implementation of *n-tuple* neural networks based on an FPGA device has been introduced. The presented novel structure for memory organisation offers effective access to the data memory for the processing of more classes at one time. It enables a very fast image recognition in a relatively simple HW architecture. The important benefit of this structure is also the fact that the system does not need a buffer for full image frame (in comparison to [85]). The system performance and the required FPGA resources are included for the estimation of the possible system performance. A real-world recognition task was chosen as the benchmark. This task consisted of road signs recognition which is a really difficult problem - 11 road signs were used. The comparison of the presented hardware solution with other methods and architectures is summarized in Table 21. The values in this table show that the designed approach is comparable to the solution published in [29] and it is not limited by the size of the embedded memory. This advantage makes it possible to recognize the images in high resolution and/or higher number of classes. If the chosen benchmark is assumed, the designed system may evaluate over 12 thousand unknown images per second. From another point of view, the system with a frame rate of 1 fps may obtain responses of up to roughly 195 thousand discriminators per second. Indeed, enough memory space is required.

  Unfortunately, the comparison of the 'learning' capability of *n-tuple* networks with the conventional feature extraction systems is not particularly fair to both techniques. The classical *n-tuple* classifiers have the generalisation properties (i.e. their probabilistic nearest matches). They can tolerate variations in the input image. It is also of a great importance that the parameters of the object in the image do not have to be analysed. On the other hand, the conventional methods – namely Feature Extraction – usually provide concise and reasonably accurate measurements of an object within an image. However, it is difficult to determine how many and which features should be extracted after the initial edge detection (i.e. perimeter, area, shape factor, min/max enclosing rectangles, centre of area, min/max radius, etc.). From the theoretical point of view, it

is evident that the conventional methods mentioned above require a lot of different operations which are needed for feature extraction. In opposite to this approach, the *n-tuple* classifiers only need reading from memory and simple ‘add’ instructions in the recognition phase. For these reasons, the time requirements should be lower.

BIBLIOGRAPHY

[1] Thompson, A. Hardware Evolution: Automatic design of electronic circuits in reconfigurable hardware by artificial evolution. New York: Springer-Verlag, 1998. ISBN 3-540-76253-1

[2] Miller, J.F.; Downing, K. Evolution in materio: Looking beyond the silicon box. In: *Evolvable Hardware, 2002. Proceedings. NASA/DoD Conference on*. IEEE, 2002. pp. 167-176.

[3] Harding, S; Miller, J.F. Evolution in materio: Initial experiments with liquid crystal. In: *Evolvable Hardware, 2004. Proceedings. 2004 NASA/DoD Conference on*. IEEE, 2004. pp. 298-305.

[4] Higuchi, Tetsuya, et al. Evolvable hardware with genetic learning: A first step towards building a darwin machine. In: *Proceedings of the 2nd International Conference on the Simulation of Adaptive Behavior*. pp. 417-424.

[5] Kajitani, I.; et al. An evolvable hardware chip for prosthetic hand controller. In: *Microelectronics for Neural, Fuzzy and Bio-Inspired Systems, 1999. MicroNeuro'99. Proceedings of the Seventh International Conference on*. IEEE, 1999, pp. 179-186. ISBN: 0-7695-0043-9. doi: 10.1109/MN.1999.758862

[6] Keymeulen, D.; et al. An evolutionary robot navigation system using a gate-level evolvable hardware. In: *Evolvable Systems: From Biology to Hardware*. Springer Berlin Heidelberg, 1997. pp. 193-209. ISBN 978-3-540-63173-6. doi: 10.1007/3-540-63173-9_47

[7] Lohn, J. D.; Hornby, Gregory S.; Linden, Derek S. An evolved antenna for deployment on nasa's space technology 5 mission. In: *Genetic Programming Theory and Practice II*. Springer US, 2005. pp. 301-315. ISBN 978-0-387-23253-9. doi: 10.1007/0-387-23254-0_18

[8] Higuchi, Tetsuya, et al. Real-world applications of analog and digital evolvable hardware. *Evolutionary Computation, IEEE Transactions on*, 1999, 3.3: pp. 220-235.

[9] Sekanina, L. *Evoluční hardware: od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Prague: Academia, 2009, 321 p. ISBN 978-80-200-1729-1

[10] Sekanina, L. *Evolvable Components: From Theory to Hardware Implementations*. Berlin: Springer-Verlag, 2004, xvi, 194 p. ISBN 35-404-0377-9.

[11] Higuchi, Tetsuya, et al. *Evolvable Hardware*. Springer US, 2006. 232 p., ISBN 978-038-73-123-85

[12] Sipper, Moshe, et al. The POE model of bio-inspired hardware systems: A short introduction. Koza et al.[786], page, 1997.

[13] Bradley, Daryl W.; Tyrrell, Andrew M. Immunotronics: Hardware fault tolerance inspired by the immune system. In: *Evolvable Systems: From Biology to Hardware*. Springer Berlin Heidelberg, 2000. pp. 11-20.

[14] Rohwer R., Morciniec M. A theoretical and experimental account of n-tuple classifier performance. *Neural Computation*, Vol. 8, Issue 3, April 1996, pp. 629-642, doi: 10.1162/neco.1996.8.3.629

[15] Bledsoe, W. W.; Browning, I. Pattern recognition and reading by machine, in: *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference* (IRE-AIEE-ACM '59 (Eastern)), ACM, New York, NY, USA, 1959, pp. 225-232, doi: 10.1145/1460299.1460326

[16] Aleksander, I.; Albrow, R.C. Pattern recognition with adaptive logic elements, In: *IEE Conference on Pattern Recognition*, 1968, pp. 68-74.

[17] Aleksander, I.; Stonham, T.J. Guide to pattern recognition using random-access memories, *IEE Journal on Computers and Digital Techniques*, vol.2, no.1, pp. 29-40, February, 1979, doi: 10.1049/ij-cdt:19790009

[18] Wilkie, B.A. A stand-alone, high resolution adaptive pattern recognition system, Ph.D. Thesis, Department of EE and E of Brunel University, UK, November 1983.

[19] Aleksander, I.; Stonham, T.J.; Wilkie B.A. Computer vision systems for industry: Wisard and the like, *Digital Systems for Industrial Automation*, vol. 4, 1982, pp. 305-320.

[20] Aleksander, I.; Thomas W.V.; Bowden P.A. WISARD. A radical step forward in pattern recognition, Sensor Review, July 1984.

[21] Holota, R.; Trejbal, J.; Wilkie, B. A. Software realisation of a colour image recognition system with an image normalisation stage. *Proceedings of the University*

of West Bohemia, section 1, vol.4/2000, pp. 75-86, ISBN 80-7082-718-1, ISSN 1211-9652

[22] Holota, R. Position, size and rotation normalisation. *Diploma project report, Dept. of Electrical Engineering, University of West Bohemia*, 2000.

[23] Trejbal, J. Software realisation of a colour image recognition system using ntuple and MIN/MAX node neural network. *Diploma project report, Dept. of Applied Sciences, University of West Bohemi*a, May 2000.

[24] Wang, Y. S.; Griffiths, B. J.; Wilkie, B. A.; Silverwood P. A.; Norgate P. Complex and coloured object inspection, *Computers in Industry*. Volume 25, Issue 2, December 1994, pp. 125-130, doi: 10.1016/0166-3615(94)90043-4.

[25] Wang, Y.S.; Griffiths, B.J.; Wilkie, B.A. A novel system for coloured object recognition, *Computers in Industry*. Volume 32, Issue 1 December 1996, pp. 69-77, doi: 10.1016/S0166-3615(96)00065-6

[26] Lucas, S.M. Real-time face recognition with the continuous n-tuple classifier. In: *High Performance Architectures for Real-Time Image Processing (Ref. No. 1998/197)*. IEE Colloquium on, pp. 11/1-11/7, 12 Feb 1998, doi: 10.1049/ic:19980051.

[27] Hepplewhite, L.; Stonhamm, T.J. Texture classification using n-tuple pattern recognition. In: *Pattern Recognition, 1996, Proceedings of the 13th International Conference on*. vol.4, pp.159-163, 25-29 Aug 1996, doi: 10.1109/ICPR.1996.547253

[28] França, H.L.; J.C.P. da Silva; De Gregorio, M.; Lengerke O.; Dutra M.S., França F.M.G. Movement persuit control of an offshore automated platform via a RAM-based neural network, Control Automation Robotics & Vision (ICARCV). In: *2010 11th International Conference on*. pp. 2437-2441, 7-10 Dec. 2010, doi: 10.1109/ICARCV.2010.5707913

[29] Bonato, V.; Sanches, A.; Fernandes, M.; Cardoso J.; Simoes E.; Marques E.. A Real Time Gesture Recognition System for Mobile Robots, In: *International Conference on Informatics in Control, Automation and Robotics*. August 25-28 2004, Setúbal, Portugal, pp. 207-214.

[30] Pattichis, C.S.; Schizas, C.N.; Sergiou, A.; Schnorrenberg, F. A hybrid neural network electromyographic system: incorporating the WISARD net. In: *Neural*

Networks, 1994, IEEE World Congress on Computational Intelligence. 1994 IEEE International Conference on. vol.6, pp. 3478-3483, 27 Jun- 2 Jul 1994, doi: 10.1109/ICNN.1994.374894

[31] Gajda, Z.: *Evolutionary Approach to Synthesis and Optimization of Ordinary and Polymorphic Circuits*, doctoral thesis, Brno, FIT BUT, 2011.

[32] Zelinka, I. *Evoluční výpočetní techniky: principy a aplikace*. BEN – technical literature, 2009. ISBN 978-80-7300-21-83.

[33] Wolpert, D. H.; Macready, W. G. No free lunch theorems for optimization. *Evolutionary Computation*, *IEEE Transactions on*, 1997, 1.1: pp. 67-82.

[34] Goldberg, David E.; Holland, John H. Genetic algorithms and machine learning. *Machine learning*, 1988, 3.2: pp. 95-99, doi: 10.1023/A:1022602019183

[35] Lažanský, M.; Mařík, V.; Štěpánková, O.; et al. *Umělá inteligence 4*. Prague: Academia, 2003. ISBN 80-200-1044-0

[36] Haupt, Randy L.; Haupt, Sue E. *Practical genetic algorithms*. 2nd ed. Hoboken: John Wiley, 2004, 253 p. ISBN 04-714-5565-2

[37] REEVES, Colin R.; Haupt. Sue E. *Genetic algorithms: principles and perspectives: a guide to GA theory*. 2nd ed. Boston: Kluwer Academic, 2003, vii, 332 p. ISBN 14-020-7240-6

[38] Lažanský, M.; Mařík, V.; Štěpánková, O.; et al. *Umělá inteligence 3*. Prague: Academie, 2003. ISBN 80-200-0472-6

[39] LOHN, Jason D.; et al. Evolutionary design of an X-band antenna for NASA's space technology 5 mission. In: *Antennas and Propagation Society International Symposium, 2004. IEEE*. IEEE, 2004, pp. 2313-2316. ISBN: 0-7803-8302-8. doi: 10.1109/APS.2004.1331834

[40] Lohn, Jason D.; et al. Evolutionary optimization of Yagi-Uda antennas. In: Yasunaga, M.; Higuchi, T.; et al. (eds.) *Evolvable Systems: From Biology to Hardware*. Springer Berlin Heidelberg, 2001, pp. 236-243. ISBN: 978-3-540-42671-4. doi: 10.1007/3-540-45443-8_21

[41] GIBSON, Jerry D. (ed.) *Digital compression for multimedia: principles and standard.* San Francisco: Morgan Kaufmann, 1998, xvii, 476 p. ISBN 15-586-0369-7.

[42] Stoica, A.; Zebulum, R.; Keymeulen, D. Mixtrinsic evolution. In: Miller, J. (ed.) *Evolvable Systems: From Biology to Hardware.* Berlin: Springer, 2000, pp. 208-217. ISBN 35-406-7338-5. doi: 10.1007/3-540-46406-9_21

[43] Salvador, R.; et al. Self-reconfigurable Evolvable Hardware System for Adaptive Image Processing. *IEEE Transactions on Computers*, vol. 62, issue: 8, IEEE, 2013. ISSN: 0018-9340. doi: 10.1109/TC.2013.78

[44] Dobai, R.; Sekanina, L. Image Filter Evolution on the Xilinx Zynq Platform. In: *Proceedings of the 2013 NASA/ESA Conference on Adaptive*, IEEE Circuits and Systems Society, Torino, IT, IEEE CAS, 2013, pp. 164-171. ISBN 978-1-4673-6381-5

[45] Zynq-7000 Programmable SoC – Overview [online]. Xilinx Cor. [cited 21.7.2013] Available on: http://www.xilinx.com/content/xilinx/en/products/silicon-devices/soc/zynq-7000.html

[46] Upegui, A. *Dynamically reconfigurable bio-inspired hardware*, doctoral thesis, École Polytechnique Fédérale de Lausanne, 2006.

[47] Sekanina, L. Virtual reconfigurable circuits for real-world applications of evolvable hardware. In: Torresen, J. (ed.) *Evolvable Systems: From Biology to Hardware.* Springer Berlin Heidelberg, 2003, pp. 186-197. ISBN: 978-3-540-00730-2. doi: 10.1007/3-540-36553-2_17

[48] Miller, J.F.: An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach. In: *Proc. Genetic and Evolutionary Computation Conference*, pp. 1135–1142. Morgan Kaufmann (1999)

[49] Miller, J.F., Thomson, P.: Cartesian Genetic Programming. In: *Proc. European Conference on Genetic Programming, LNCS*, vol. 1802, pp. 121–132. Springer (2000)

[50] Miller, J.F.; Thomson, P.; Fogarty, T.C.: Designing Electronic Circuits Using Evolutionary Algorithms: Arithmetic Circuits: A Case Study. In: Quagliarella, D.; Periaux, J.; Poloni, C.; Winter, G. (eds.) *Genetic Algorithms and Evolution Strategies*

in Engineering and Computer Science: Recent Advancements and Industrial Applications, pp. 105–131. Wiley (1998)

[51] Miller, J.F.; Job, D.; Vassilev, V.K.: Principles in the Evolutionary Design of Digital Circuits - Part I. *Genetic Programming and Evolvable Machines 1(1)*, 8-35 (2000), doi: 10.1023/A:1010016313373

[52] Gajda, Z., Sekanina, L.: An Efficient Selection Strategy for Digital Circuit Evolution, In: Miller, J.F.; Tyrrel, Andy M.; Tempesti, G. (eds.) *Evolvable Systems: From Biology to Hardware*, Berlin, DE, Springer, 2010, pp. 13-24, ISBN 978-3-642-15322-8

[53] Miller, J. F. (ed.) *Cartesian genetic programming.* Springer Berlin Heidelberg, 2011, 344 p., ISBN 978-3-642-17309-7

[54] Miller, J.F.; Smith, S.L. Redundancy and Computational Efficiency in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation, 10(2),* (2006), pp. 167-174, doi: 10.1109/TEVC.2006.871253

[55] Vassilev, V.K.; Miller, J.F.: The Advantages of Landscape Neutrality in Digital Circuit Evolution. *Proc. International Conference on Evolvable Systems, LNCS*, vol. 1801, pp. 252–263. Springer (2000).

[56] Walker, J.A., Miller, J.F.: Automatic Acquisition, Evolution and Re-use of Modules in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 12, pp. 397–417 (2008).

[57] Miller, J.F. *Collection of CGP programs.* [online]. [cited 20.7.2013]. Available on: http://www.cartesiangp.co.uk/resources/CGP-version1_1.7z

[58] Oranchak, D. *CGP implemented in Java.* [online]. [cited 20.7.2013]. Available on: http://oranchak.com/cgp/doc/

[59] Pollack, J. *MATLAB implementation of CGP.* [online]. [cited 20.7.2013]. Available on: http://www.cartesiangp.co.uk/resources/cgpmatlab.zip

[60] Vašíček, Z.; Sekanina, L. *Nástroje pro kartézské genetické programování Tools4CGP.* [online]. [cited 20.7.2013]. Available on: http://www.fit.vutbr.cz/~Vašíček/cgp/tools/

[61] Vassilev, V. K.; Miller, J. F.; Fogarty, T. C. On the nature of two-bit multiplier landscapes. In: *Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on*, IEEE, 1999, pp. 36-45. ISBN: 0-7695-0256-3. doi: 10.1109/EH.1999.785433

[62] Sekanina, L.; Friedl, Š. An evolvable combinational unit for FPGAs. *Computing and Informatics*, 2012, 23.5-6: pp. 461-486. ISSN: 1335-9150

[63] Vašíček, Z.; Sekanina, L. An evolvable hardware system in Xilinx Virtex II Pro FPGA. *International Journal of Innovative Computing and Applications*, 2007, 1.1: pp. 63-73. ISSN 1751-648X

[64] Vašíček, Z., Sekanina, L.: Evaluation of a New Platform For Image Filter Evolution, In: *Proc. of the 2007 NASA/ESA Conference on Adaptive Hardware and Systems*. Los Alamitos, US, IEEE CS, 2007, pp. 577-584. ISBN 076952866X

[65] Vašíček, Z.; Sekanina, L. Hardware Accelerator of Cartesian Genetic Programming with Multiple Fitness Units. *Computing and Informatics*, 2012, 29.6+: pp. 1359-1371. ISSN: 1335-9150

[66] Nirmalkumar, P.; et al. On Suitability of FPGA Based Evolvable Hardware Systems to Integrate Reconfigurable Circuits with Host Processing Unit. *IJCSNS - International Journal of Computer Science and Network Security*, 2006, Vol. 6, No. 9, pp. 216-222. ISSN: 1738-7906

[67] Kumar, P. Nirmal; et al. Testing Virtual Reconfigurable Circuit Designed For A Fault Tolerant System. *Journal of Computer Science*, 2007, 3.12: 934, pp. 934-938. doi: 10.3844/jcssp.2007.934.938

[68] Kumar, P.N.; Anandhi, S.; Perinbam, J.R.P., Evolving virtual reconfigurable circuit for a fault tolerant system. *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on* , vol., no., pp.1555-1561, 25-28 Sept. 2007 doi: 10.1109/CEC.2007.4424658

[69] Wang, J.; Chen, Q. S.; Lee, C.H. Design and implementation of a virtual reconfigurable architecture for different applications of intrinsic evolvable hardware. *Computers & Digital Techniques, IET*, vol.2, no.5, pp. 386-400, September 2008. doi: 10.1049/iet-cdt:20070124

[70] Sekanina, L. Evolutionary hardware design (Invited Paper), In: *VLSI Circuits and Systems V*, Bellingham, US, SPIE, 2011, pp. 1-11. ISBN 978-0-8194-8656-1

[71] Avalon Interface Specification. [online]. Altera Corp. [cited 20.7.2013] Available on: http://www.altera.com/literature/manual/mnl_avalon_spec.pdf?GSA_pos=1&WT.oss_r=1&WT.oss=avalon%20streaming%20spec

[72] Miller, J. F. Tutorial on Cartesian Genetic Programming. In: *Genetic and Evolutionary Computation Conference*. Amsterdam, Netherlands, 2013.

[73] Sekanina, L.; Friedl, Š. On Routine Implementation of Virtual Evolvable Devices Using C COMBO6, In: Proc. of the 2004 NASA/DoD Conference on Evolvable Hardware, Los Alamitos, US, ICSP, 2004, pp. 63-70. ISBN 0-7695-2145-2

[74] Martínek, T.; Sekanina, L. An Evolvable Image Filter: Experimental Evaluation of a Complete Hardware Implementation in FPGA, In: Moreno, J.M. (ed.) *Evolvable Systems: From Biology to Hardware*, Berlin, DE, Springer, 2005, pp. 76-85, ISBN 978-3-540-28736-0

[75] Lambert, C.; Kalganova, T.; Stomeo, E. FPGA-based systems for evolvable hardware. *Proceedings of World Academy of Science, Engineering and Technology*. 2006, vo. 12, pp. 123-129. ISSN 1307-688

[76] Memory Blocks in Cyclone IV Devices. In: Cyclone IV Device Handbook. [online], Altera Corp. [cited 22.6.2013]. Available on: http://www.altera.com

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

[78] Cyclone II Device Handbook, Altera Corp. [online]. [cited 17.4.2009]. Available on: http://www.altera.com

[79] WM8731 / WM8731L Datasheet, Wolfson Microelectronics. [online]. [cited 17.4.2009]. Available on: http://www.wolfsonmicro.com

[80] Nios II Processor Reference, Altera Corp. [online]. [cited 17.4.2009]. Available on: http://www.altera.com

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

[82] Austin, J.; RAM-based neural networks. *Progress in Neural Processing*. vol. 9, p.252, February1998, ISBN: 978-981-02-3253-5, ISSN: 2010-2895.

[83] Rohwer, R.J. Two Bayesian treatments of the n-tuple recognition method. In: *Fourth International Conference on Artificial Neural Networks 1995*. pp. 171-176, 26-28 Jun 1995, doi: 10.1049/cp:19950549.

[84] Altera Corp., Qsys Interconnect (chapter of Quartus II Handbook 11.1). [online]. [cited 7.9.2012]. Available on: http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf.

[85] Mitchell, R.J.; Bishop, J.M.; Minchinton P.R. Optimising memory usage in n-tuple neural networks. *Mathematics and Computers in Simulation*. Volume 40, Issues 5–6, May 1996, pp. 549–563, doi: 10.1016/0378-4754(95)00006-2.

AUTHOR'S PUBLICATIONS

List of author's publications related to thesis

Impact journal:

[A1] Burian, P.; Holota R. Fast image recognition based on n-tuple neural networks implemented in an FPGA. *Journal of Real-Time Image Processing. Springer, 2013. (accepted, available online, doi: 10.1007/s11554-013-0331-8)*

Book section:

[A2] Burian, P. Evolvable Hardware Implemented by FPGA. In: Nawrocki, R. (ed.) *Computer Applications in Electrical Engineering*. Poznan : COMPRINT, 2009, pp.100-117. ISBN: 978-83-89333-33-9

Journal:

[A3] Burian, P. Návrh číslicových obvodů za pomoci evolučních výpočetních technik. *Automatizace,* 2009, 52(3), pp. 178-180. ISSN: 0005-125X

Conference contributions:

[A4] Burian, P. Compact Version of Cartesian Genetic Programming. In: *21st Telecommunications Forum TELFOR 2013*, Beograd, Serbia, 2013. (the paper in review)

[A5] Burian, P. Reduction of Fitness Calculations in Cartesian Genetic Programming. In: *Applied Electronics (AE), 2013 International Conference on,* Pilsen: University of West Bohemia, 2013. pp.53-58. ISBN: 978-80-261-0166-6, ISSN: 1803-7232

[A6] Burian, P. Design Tool for Evolutionary Design of Digital Circuits. In: *Applied Electronics (AE), 2012 International Conference on,* Pilsen: University of West Bohemia, 2012. pp. 47-50. ISBN: 978-80-261-0038-6, ISSN: 1803-7232

[A7] Burian, P. Využití Evolučních výpočetních technik v elektronických systémech – přehled zajímavých aplikací. In: *Elektrotechnika a informatika 2011. Část 2., Elektronika.* Pilsen: University of West Bohemia, 2011. pp. 13-16. ISBN: 978-80-261-0015-7

[A8] Burian, P. Recombination Operators in Evolutionary FIR Filter Domain. In: *Computer Applications in Electrical Engineering - ZKwE'2010*. Poznan: Poznan University of Technology, 2010. pp. 57-58. ISBN: 978-83-89333-34-6

[A9] Burian, P. Cartesian Genetic Programming Using Bit Representation. In: *TELFOR 2010 : proceedings*. Belgrade: TELFOR, 2010. pp. 803-806. ISBN: 978-86-7466-392-9

[A10] Burian, P. Vlastnosti Kartézského genetického programování. In: *Elektrotechnika a informatika 2010. Část 2., Elektronika*. Pilsen: University of West Bohemia, 2010. pp. 21-24. ISBN: 978-80-7043-914-2

[A11] Burian, P. Evolutionary FIR filter. In: *Applied Electronics (AE), 2009 International Conference on,* Pilsen: University of West Bohemia, 2009. pp. 69-72. ISBN: 978-80-7043-781-0

[A12] Burian, P. Implementation of Small Evolvable Combinational Logic Circuit by FPGA. In: Computer *Applications in Electrical Engineering*. Poznan: University of Technology, Institute of Electrical Engineering and Electronics, 2009. pp. 207-208. ISBN: 978-83-89333-24-7

[A13] Burian, P. Implementace genetického algoritmu do obvodu FPGA. In: *Elektrotechnika a informatika 2008. Část 2., Elektronika*. Pilsen: University of West Bohemia, 2008. pp. 19-22. ISBN: 978-80-7043-701-8

APPENDICES

A.

Automatically generated source code of the evolved 5-bit parity circuit

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity parity is
          port
          (
          -- clock and reset port
            clk           : in std_logic;
            reset_n       : in std_logic;

            -- input ports
            input_data_0  : in std_logic;
            input_data_1  : in std_logic;
            input_data_2  : in std_logic;
            input_data_3  : in std_logic;
            input_data_4  : in std_logic;

            -- output port
            output_data_0 : out std_logic
          );
end;

architecture rtl of parity is

component func_or is
          generic
            (
            datawidth : positive := 1
            );
          port
            (
            input_data_A : in std_logic_vector(datawidth-1 downto 0);
            input_data_B : in std_logic_vector(datawidth-1 downto 0);
            output_data  : out std_logic_vector(datawidth-1 downto 0)
            );
end component;

component func_and is
          generic
            (
            datawidth : positive := 1
            );
          port
            (
            input_data_A : in std_logic_vector(datawidth-1 downto 0);
            input_data_B : in std_logic_vector(datawidth-1 downto 0);
            output_data  : out std_logic_vector(datawidth-1 downto 0)
            );
end component;

component func_xor is
          generic
            (
            datawidth : positive := 1
            );
          port
            (
            input_data_A : in std_logic_vector(datawidth-1 downto 0);
            input_data_B : in std_logic_vector(datawidth-1 downto 0);
            output_data  : out std_logic_vector(datawidth-1 downto 0)
            );
end component;

-- outputs of cells
signal interconnect_22 : std_logic_vector(0 downto 0);
signal interconnect_16 : std_logic_vector(0 downto 0);
signal interconnect_18 : std_logic_vector(0 downto 0);
signal interconnect_7  : std_logic_vector(0 downto 0);
signal interconnect_9  : std_logic_vector(0 downto 0);
signal interconnect_13 : std_logic_vector(0 downto 0);
signal interconnect_4  : std_logic_vector(0 downto 0);
signal interconnect_3  : std_logic_vector(0 downto 0);
signal interconnect_2  : std_logic_vector(0 downto 0);
signal interconnect_0  : std_logic_vector(0 downto 0);

-- input interconnect signals
signal input_data_interconnect_0 : std_logic_vector(0 downto 0);
signal input_data_interconnect_1 : std_logic_vector(0 downto 0);
signal input_data_interconnect_2 : std_logic_vector(0 downto 0);
signal input_data_interconnect_3 : std_logic_vector(0 downto 0);
signal input_data_interconnect_4 : std_logic_vector(0 downto 0);

-- output interconnect signals
signal output_data_interconnect_0 : std_logic_vector(0 downto 0);

begin
```

```
-- input interconnection
process(clk, reset_n)
begin
          if(reset_n = '0') then
            input_data_interconnect_0 <= (others => '0');
            input_data_interconnect_1 <= (others => '0');
            input_data_interconnect_2 <= (others => '0');
            input_data_interconnect_3 <= (others => '0');
            input_data_interconnect_4 <= (others => '0');
          elsif(clk='1' and clk'event) then
            input_data_interconnect_0(0) <= input_data_0;
            input_data_interconnect_1(0) <= input_data_1;
            input_data_interconnect_2(0) <= input_data_2;
            input_data_interconnect_3(0) <= input_data_3;
            input_data_interconnect_4(0) <= input_data_4;
          end if;
end process;

-- instances of cells
inst_cell_22      :   func_or       port map (interconnect_16, interconnect_18, interconnect_22);
inst_cell_16      :   func_and      port map (interconnect_7, interconnect_9, interconnect_16);
inst_cell_18      :   func_and      port map (input_data_interconnect_0, interconnect_13, interconnect_18);
inst_cell_7       :   func_or       port map (interconnect_4, interconnect_3, interconnect_7);
inst_cell_9       :   func_and      port map (interconnect_2, interconnect_0, interconnect_9);
inst_cell_13      :   func_xor      port map (interconnect_9, interconnect_7, interconnect_13);
inst_cell_4       :   func_and      port map (input_data_interconnect_3, input_data_interconnect_4, interconnect_4);
inst_cell_3       :   func_and      port map (input_data_interconnect_1, input_data_interconnect_2, interconnect_3);
inst_cell_2       :   func_or       port map (input_data_interconnect_3, input_data_interconnect_4, interconnect_2);
inst_cell_0       :   func_or       port map (input_data_interconnect_2, input_data_interconnect_1, interconnect_0);

-- output interconnection
output_data_interconnect_0 <= interconnect_22;

process(clk, reset_n)
begin
          if(reset_n = '0') then
            output_data_0 <= '0';
          elsif(clk= '1' and clk'event) then
            output_data_0 <= output_data_interconnect_0(0);
          end if;
end process;

end;
```

B.

Example of automatically generated source code – Virtual Reconfigurable Circuit

Entity: *vrc_top*

```
-- Virtual Reconfigurable Device
-- Entity name: vrc_top
--
-- VRC parameters:
-- Number of columns: 5
-- Number of rows: 5
-- L-back: 1
-- Input availability degree: 5
-- Output availability degree: 5
-- Config interface: Parallel
--
-- Generated by VHDL Designer - 4.5.2013 23:30:21

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.vhdl_designer.all;

entity vrc_top is

    port
        (
        -- primary inputs
        primary_input_0 : in std_logic_vector(7 downto 0);
        primary_input_1 : in std_logic_vector(7 downto 0);
        primary_input_2 : in std_logic_vector(7 downto 0);
        primary_input_3 : in std_logic_vector(7 downto 0);
        primary_input_4 : in std_logic_vector(7 downto 0);

        primary_input_valid : in std_logic;

        -- primary outputs
        primary_output_0 : out std_logic_vector(7 downto 0);
        primary_output_1 : out std_logic_vector(7 downto 0);
        primary_output_2 : out std_logic_vector(7 downto 0);
        primary_output_3 : out std_logic_vector(7 downto 0);
        primary_output_4 : out std_logic_vector(7 downto 0);

        primary_output_valid : out std_logic;

        -- config data port
        column_0_config_data : in std_logic_vector(39 downto 0);
        column_1_config_data : in std_logic_vector(49 downto 0);
        column_2_config_data : in std_logic_vector(49 downto 0);
        column_3_config_data : in std_logic_vector(49 downto 0);
        column_4_config_data : in std_logic_vector(49 downto 0);

        output_0_config_data : in std_logic_vector(4 downto 0);
        output_1_config_data : in std_logic_vector(4 downto 0);
        output_2_config_data : in std_logic_vector(4 downto 0);
        output_3_config_data : in std_logic_vector(4 downto 0);
        output_4_config_data : in std_logic_vector(4 downto 0);

        clock_config : in std_logic;

        config_column_0_wr : in std_logic;
        config_column_1_wr : in std_logic;
        config_column_2_wr : in std_logic;
        config_column_3_wr : in std_logic;
        config_column_4_wr : in std_logic;

        config_output_0_wr : in std_logic;
        config_output_1_wr : in std_logic;
        config_output_2_wr : in std_logic;
        config_output_3_wr : in std_logic;
        config_output_4_wr : in std_logic;

        -- other signals
        clk_vrc : in std_logic;
        clk_vrc_en : in std_logic;
        reset_n : in std_logic

        );

end;

architecture vrc_rtl of vrc_top is

-- components of columns
component cell_column_input is
    generic
        (
        datawidth : positive := 8
        );
    port
        (

        input_primary_data_port   : in  array_std_logic_vector(4 downto 0, datawidth-1 downto 0);
        cell_output_port          : out array_std_logic_vector(4 downto 0, datawidth-1 downto 0);

        clk     : in std_logic;
```

```
        clk_en  : in std_logic;

        column_config   : in std_logic_vector(39 downto 0);
        clk_config      : in std_logic;
        wr_config_en    : in std_logic

        );
end component;

component cell_column_type_0 is
    generic
        (
        datawidth : positive := 8
        );
    port
        (

        input_primary_data_port   : in  array_std_logic_vector(4 downto 0, datawidth-1 downto 0);
        input_cell_data_port      : in  array_std_logic_vector(4 downto 0, datawidth-1 downto 0);
        cell_output_port          : out array_std_logic_vector(4 downto 0, datawidth-1 downto 0);

        clk     : in std_logic;
        clk_en  : in std_logic;

        column_config   : in std_logic_vector(49 downto 0);
        clk_config      : in std_logic;
        wr_config_en    : in std_logic

        );
end component;

-- interconnect between columns
signal columns_data_bus_0 : array_std_logic_vector(4 downto 0, 7 downto 0);
signal columns_data_bus_1 : array_std_logic_vector(4 downto 0, 7 downto 0);
signal columns_data_bus_2 : array_std_logic_vector(4 downto 0, 7 downto 0);
signal columns_data_bus_3 : array_std_logic_vector(4 downto 0, 7 downto 0);
signal columns_data_bus_4 : array_std_logic_vector(4 downto 0, 7 downto 0);

type   shift_register is array( 0 to 3) of array_std_logic_vector(4 downto 0,7 downto 0);
signal input_shift_register   : shift_register;
signal input_data_vector      : array_std_logic_vector(4 downto 0, 7 downto 0);

signal valid_shift_register : std_logic_vector(0 to 5);

type   compensatory_register_chain is array (3 downto 0) of std_logic_vector(7 downto 0);
signal compensatory_registers_0 : compensatory_register_chain;
signal compensatory_registers_1 : compensatory_register_chain;
signal compensatory_registers_2 : compensatory_register_chain;
signal compensatory_registers_3 : compensatory_register_chain;
signal compensatory_registers_4 : compensatory_register_chain;

type   output_config_array is array (0 to 4) of std_logic_vector(4 downto 0);
signal output_config_storage : output_config_array;

signal selected_output_data_0 : std_logic_vector(7 downto 0);
signal selected_output_data_1 : std_logic_vector(7 downto 0);
signal selected_output_data_2 : std_logic_vector(7 downto 0);
signal selected_output_data_3 : std_logic_vector(7 downto 0);
signal selected_output_data_4 : std_logic_vector(7 downto 0);

signal register_latency_pointer_0 : std_logic_vector(3 downto 0);
signal register_latency_pointer_1 : std_logic_vector(3 downto 0);
signal register_latency_pointer_2 : std_logic_vector(3 downto 0);
signal register_latency_pointer_3 : std_logic_vector(3 downto 0);
signal register_latency_pointer_4 : std_logic_vector(3 downto 0);

signal primary_output_0_i : std_logic_vector(7 downto 0);
signal primary_output_1_i : std_logic_vector(7 downto 0);
signal primary_output_2_i : std_logic_vector(7 downto 0);
signal primary_output_3_i : std_logic_vector(7 downto 0);
signal primary_output_4_i : std_logic_vector(7 downto 0);

begin

column_0 : cell_column_input
    generic map
        (
        datawidth => 8
        )
    port map
        (
        input_primary_data_port    => input_data_vector,
        cell_output_port           => columns_data_bus_0,

        clk                        => clk_vrc,
        clk_en                     => clk_vrc_en,

        column_config              => column_0_config_data,
        wr_config_en               => config_column_0_wr,
        clk_config                 => clock_config
        );

column_1 : cell_column_type_0
    generic map
        (
        datawidth => 8
        )
    port map
        (
        input_primary_data_port    => input_shift_register(0),
        input_cell_data_port       => columns_data_bus_0,
        cell_output_port           => columns_data_bus_1,
```

```
        clk                         => clk_vrc,
        clk_en                      => clk_vrc_en,

        column_config               => column_1_config_data,
        wr_config_en                => config_column_1_wr,
        clk_config                  => clock_config
        );

column_2 : cell_column_type_0
    generic map
        (
        datawidth => 8
        )
    port map
        (
        input_primary_data_port     => input_shift_register(1),
        input_cell_data_port        => columns_data_bus_1,
        cell_output_port            => columns_data_bus_2,

        clk                         => clk_vrc,
        clk_en                      => clk_vrc_en,

        column_config               => column_2_config_data,
        wr_config_en                => config_column_2_wr,
        clk_config                  => clock_config
        );

column_3 : cell_column_type_0
    generic map
        (
        datawidth => 8
        )
    port map
        (
        input_primary_data_port     => input_shift_register(2),
        input_cell_data_port        => columns_data_bus_2,
        cell_output_port            => columns_data_bus_3,

        clk                         => clk_vrc,
        clk_en                      => clk_vrc_en,

        column_config               => column_3_config_data,
        wr_config_en                => config_column_3_wr,
        clk_config                  => clock_config
        );

column_4 : cell_column_type_0
    generic map
        (
        datawidth => 8
        )
    port map
        (
        input_primary_data_port     => input_shift_register(3),
        input_cell_data_port        => columns_data_bus_3,
        cell_output_port            => columns_data_bus_4,

        clk                         => clk_vrc,
        clk_en                      => clk_vrc_en,

        column_config               => column_4_config_data,
        wr_config_en                => config_column_4_wr,
        clk_config                  => clock_config
        );


-- Valid shift register
valid_shift_register_process : process(clk_vrc, reset_n)
begin

    if(reset_n='0') then

        valid_shift_register<=(others=>'0');

    elsif(clk_vrc='1' and clk_vrc'event) then

        if(clk_vrc_en='1') then
            valid_shift_register<= primary_input_valid & valid_shift_register(0 to 4);
        end if;

    end if;

end process;

primary_output_valid <= valid_shift_register(5);


-- Convert input data port
process(primary_input_0,primary_input_1,primary_input_2,primary_input_3,primary_input_4)
begin

    to_array_std_logic_vector(input_data_vector,primary_input_0,0);
    to_array_std_logic_vector(input_data_vector,primary_input_1,1);
    to_array_std_logic_vector(input_data_vector,primary_input_2,2);
    to_array_std_logic_vector(input_data_vector,primary_input_3,3);
    to_array_std_logic_vector(input_data_vector,primary_input_4,4);

end process;

input_data_shift_register_process : process(clk_vrc)
begin

    if(clk_vrc='1' and clk_vrc'event) then
```

```
        if(clk_vrc_en='1') then
            input_shift_register<= input_data_vector & input_shift_register(0 to 2);
        end if;
    end if;

end process;


output_config_write : process(clock_config)
begin

    if(clock_config='1' and clock_config'event) then

        if(config_output_0_wr='1') then
            output_config_storage(0) <= output_0_config_data;
        end if;
        if(config_output_1_wr='1') then
            output_config_storage(1) <= output_1_config_data;
        end if;
        if(config_output_2_wr='1') then
            output_config_storage(2) <= output_2_config_data;
        end if;
        if(config_output_3_wr='1') then
            output_config_storage(3) <= output_3_config_data;
        end if;
        if(config_output_4_wr='1') then
            output_config_storage(4) <= output_4_config_data;
        end if;
    end if;

end process;

with output_config_storage(0) select
    selected_output_data_0 <= to_std_logic_vector(columns_data_bus_0,0) when "00000",
    to_std_logic_vector(columns_data_bus_0,1) when "00001",
    to_std_logic_vector(columns_data_bus_0,2) when "00010",
    to_std_logic_vector(columns_data_bus_0,3) when "00011",
    to_std_logic_vector(columns_data_bus_0,4) when "00100",
    to_std_logic_vector(columns_data_bus_1,0) when "00101",
    to_std_logic_vector(columns_data_bus_1,1) when "00110",
    to_std_logic_vector(columns_data_bus_1,2) when "00111",
    to_std_logic_vector(columns_data_bus_1,3) when "01000",
    to_std_logic_vector(columns_data_bus_1,4) when "01001",
    to_std_logic_vector(columns_data_bus_2,0) when "01010",
    to_std_logic_vector(columns_data_bus_2,1) when "01011",
    to_std_logic_vector(columns_data_bus_2,2) when "01100",
    to_std_logic_vector(columns_data_bus_2,3) when "01101",
    to_std_logic_vector(columns_data_bus_2,4) when "01110",
    to_std_logic_vector(columns_data_bus_3,0) when "01111",
    to_std_logic_vector(columns_data_bus_3,1) when "10000",
    to_std_logic_vector(columns_data_bus_3,2) when "10001",
    to_std_logic_vector(columns_data_bus_3,3) when "10010",
    to_std_logic_vector(columns_data_bus_3,4) when "10011",
    to_std_logic_vector(columns_data_bus_4,0) when "10100",
    to_std_logic_vector(columns_data_bus_4,1) when "10101",
    to_std_logic_vector(columns_data_bus_4,2) when "10110",
    to_std_logic_vector(columns_data_bus_4,3) when "10111",
    to_std_logic_vector(columns_data_bus_4,4) when others;

with output_config_storage(1) select
    selected_output_data_1 <= to_std_logic_vector(columns_data_bus_0,0) when "00000",
    to_std_logic_vector(columns_data_bus_0,1) when "00001",
    to_std_logic_vector(columns_data_bus_0,2) when "00010",
    to_std_logic_vector(columns_data_bus_0,3) when "00011",
    to_std_logic_vector(columns_data_bus_0,4) when "00100",
    to_std_logic_vector(columns_data_bus_1,0) when "00101",
    to_std_logic_vector(columns_data_bus_1,1) when "00110",
    to_std_logic_vector(columns_data_bus_1,2) when "00111",
    to_std_logic_vector(columns_data_bus_1,3) when "01000",
    to_std_logic_vector(columns_data_bus_1,4) when "01001",
    to_std_logic_vector(columns_data_bus_2,0) when "01010",
    to_std_logic_vector(columns_data_bus_2,1) when "01011",
    to_std_logic_vector(columns_data_bus_2,2) when "01100",
    to_std_logic_vector(columns_data_bus_2,3) when "01101",
    to_std_logic_vector(columns_data_bus_2,4) when "01110",
    to_std_logic_vector(columns_data_bus_3,0) when "01111",
    to_std_logic_vector(columns_data_bus_3,1) when "10000",
    to_std_logic_vector(columns_data_bus_3,2) when "10001",
    to_std_logic_vector(columns_data_bus_3,3) when "10010",
    to_std_logic_vector(columns_data_bus_3,4) when "10011",
    to_std_logic_vector(columns_data_bus_4,0) when "10100",
    to_std_logic_vector(columns_data_bus_4,1) when "10101",
    to_std_logic_vector(columns_data_bus_4,2) when "10110",
    to_std_logic_vector(columns_data_bus_4,3) when "10111",
    to_std_logic_vector(columns_data_bus_4,4) when others;

with output_config_storage(2) select
    selected_output_data_2 <= to_std_logic_vector(columns_data_bus_0,0) when "00000",
    to_std_logic_vector(columns_data_bus_0,1) when "00001",
    to_std_logic_vector(columns_data_bus_0,2) when "00010",
    to_std_logic_vector(columns_data_bus_0,3) when "00011",
    to_std_logic_vector(columns_data_bus_0,4) when "00100",
    to_std_logic_vector(columns_data_bus_1,0) when "00101",
    to_std_logic_vector(columns_data_bus_1,1) when "00110",
    to_std_logic_vector(columns_data_bus_1,2) when "00111",
    to_std_logic_vector(columns_data_bus_1,3) when "01000",
    to_std_logic_vector(columns_data_bus_1,4) when "01001",
    to_std_logic_vector(columns_data_bus_2,0) when "01010",
    to_std_logic_vector(columns_data_bus_2,1) when "01011",
    to_std_logic_vector(columns_data_bus_2,2) when "01100",
    to_std_logic_vector(columns_data_bus_2,3) when "01101",
    to_std_logic_vector(columns_data_bus_2,4) when "01110",
    to_std_logic_vector(columns_data_bus_3,0) when "01111",
```

```
    to_std_logic_vector(columns_data_bus_3,1) when "10000",
    to_std_logic_vector(columns_data_bus_3,2) when "10001",
    to_std_logic_vector(columns_data_bus_3,3) when "10010",
    to_std_logic_vector(columns_data_bus_3,4) when "10011",
    to_std_logic_vector(columns_data_bus_4,0) when "10100",
    to_std_logic_vector(columns_data_bus_4,1) when "10101",
    to_std_logic_vector(columns_data_bus_4,2) when "10110",
    to_std_logic_vector(columns_data_bus_4,3) when "10111",
    to_std_logic_vector(columns_data_bus_4,4) when others;

with output_config_storage(3) select
    selected_output_data_3 <= to_std_logic_vector(columns_data_bus_0,0) when "00000",
    to_std_logic_vector(columns_data_bus_0,1) when "00001",
    to_std_logic_vector(columns_data_bus_0,2) when "00010",
    to_std_logic_vector(columns_data_bus_0,3) when "00011",
    to_std_logic_vector(columns_data_bus_0,4) when "00100",
    to_std_logic_vector(columns_data_bus_1,0) when "00101",
    to_std_logic_vector(columns_data_bus_1,1) when "00110",
    to_std_logic_vector(columns_data_bus_1,2) when "00111",
    to_std_logic_vector(columns_data_bus_1,3) when "01000",
    to_std_logic_vector(columns_data_bus_1,4) when "01001",
    to_std_logic_vector(columns_data_bus_2,0) when "01010",
    to_std_logic_vector(columns_data_bus_2,1) when "01011",
    to_std_logic_vector(columns_data_bus_2,2) when "01100",
    to_std_logic_vector(columns_data_bus_2,3) when "01101",
    to_std_logic_vector(columns_data_bus_2,4) when "01110",
    to_std_logic_vector(columns_data_bus_3,0) when "01111",
    to_std_logic_vector(columns_data_bus_3,1) when "10000",
    to_std_logic_vector(columns_data_bus_3,2) when "10001",
    to_std_logic_vector(columns_data_bus_3,3) when "10010",
    to_std_logic_vector(columns_data_bus_3,4) when "10011",
    to_std_logic_vector(columns_data_bus_4,0) when "10100",
    to_std_logic_vector(columns_data_bus_4,1) when "10101",
    to_std_logic_vector(columns_data_bus_4,2) when "10110",
    to_std_logic_vector(columns_data_bus_4,3) when "10111",
    to_std_logic_vector(columns_data_bus_4,4) when others;

with output_config_storage(4) select
    selected_output_data_4 <= to_std_logic_vector(columns_data_bus_0,0) when "00000",
    to_std_logic_vector(columns_data_bus_0,1) when "00001",
    to_std_logic_vector(columns_data_bus_0,2) when "00010",
    to_std_logic_vector(columns_data_bus_0,3) when "00011",
    to_std_logic_vector(columns_data_bus_0,4) when "00100",
    to_std_logic_vector(columns_data_bus_1,0) when "00101",
    to_std_logic_vector(columns_data_bus_1,1) when "00110",
    to_std_logic_vector(columns_data_bus_1,2) when "00111",
    to_std_logic_vector(columns_data_bus_1,3) when "01000",
    to_std_logic_vector(columns_data_bus_1,4) when "01001",
    to_std_logic_vector(columns_data_bus_2,0) when "01010",
    to_std_logic_vector(columns_data_bus_2,1) when "01011",
    to_std_logic_vector(columns_data_bus_2,2) when "01100",
    to_std_logic_vector(columns_data_bus_2,3) when "01101",
    to_std_logic_vector(columns_data_bus_2,4) when "01110",
    to_std_logic_vector(columns_data_bus_3,0) when "01111",
    to_std_logic_vector(columns_data_bus_3,1) when "10000",
    to_std_logic_vector(columns_data_bus_3,2) when "10001",
    to_std_logic_vector(columns_data_bus_3,3) when "10010",
    to_std_logic_vector(columns_data_bus_3,4) when "10011",
    to_std_logic_vector(columns_data_bus_4,0) when "10100",
    to_std_logic_vector(columns_data_bus_4,1) when "10101",
    to_std_logic_vector(columns_data_bus_4,2) when "10110",
    to_std_logic_vector(columns_data_bus_4,3) when "10111",
    to_std_logic_vector(columns_data_bus_4,4) when others;

latency_pointer_decoder_0 : process(output_config_storage(0))
begin

    if(unsigned(output_config_storage(0))<5) then
        register_latency_pointer_0 <= "0000";
    elsif(unsigned(output_config_storage(0))<10) then
        register_latency_pointer_0 <= "1000";
    elsif(unsigned(output_config_storage(0))<15) then
        register_latency_pointer_0 <= "-100";
    elsif(unsigned(output_config_storage(0))<20) then
        register_latency_pointer_0 <= "--10";
    else
        register_latency_pointer_0 <= "---1";
    end if;

end process;

latency_pointer_decoder_1 : process(output_config_storage(1))
begin

    if(unsigned(output_config_storage(1))<5) then
        register_latency_pointer_1 <= "0000";
    elsif(unsigned(output_config_storage(1))<10) then
        register_latency_pointer_1 <= "1000";
    elsif(unsigned(output_config_storage(1))<15) then
        register_latency_pointer_1 <= "-100";
    elsif(unsigned(output_config_storage(1))<20) then
        register_latency_pointer_1 <= "--10";
    else
        register_latency_pointer_1 <= "---1";
    end if;
end process;

latency_pointer_decoder_2 : process(output_config_storage(2))
begin

    if(unsigned(output_config_storage(2))<5) then
        register_latency_pointer_2 <= "0000";
    elsif(unsigned(output_config_storage(2))<10) then
```

```
        register_latency_pointer_2 <= "1000";
    elsif(unsigned(output_config_storage(2))<15) then
        register_latency_pointer_2 <= "-100";
    elsif(unsigned(output_config_storage(2))<20) then
        register_latency_pointer_2 <= "--10";
    else
        register_latency_pointer_2 <= "---1";
    end if;

end process;

latency_pointer_decoder_3 : process(output_config_storage(3))
begin

    if(unsigned(output_config_storage(3))<5) then
        register_latency_pointer_3 <= "0000";
    elsif(unsigned(output_config_storage(3))<10) then
        register_latency_pointer_3 <= "1000";
    elsif(unsigned(output_config_storage(3))<15) then
        register_latency_pointer_3 <= "-100";
    elsif(unsigned(output_config_storage(3))<20) then
        register_latency_pointer_3 <= "--10";
    else
        register_latency_pointer_3 <= "---1";
    end if;

end process;

latency_pointer_decoder_4 : process(output_config_storage(4))
begin

    if(unsigned(output_config_storage(4))<5) then
        register_latency_pointer_4 <= "0000";
    elsif(unsigned(output_config_storage(4))<10) then
        register_latency_pointer_4 <= "1000";
    elsif(unsigned(output_config_storage(4))<15) then
        register_latency_pointer_4 <= "-100";
    elsif(unsigned(output_config_storage(4))<20) then
        register_latency_pointer_4 <= "--10";
    else
        register_latency_pointer_4 <= "---1";
    end if;

end process;

latency_compensation_process : process(clk_vrc)
begin

    if(clk_vrc='1' and clk_vrc'event) then
        if(clk_vrc_en='1') then

                compensatory_registers_0(3) <= selected_output_data_0;
                compensatory_registers_1(3) <= selected_output_data_1;
                compensatory_registers_2(3) <= selected_output_data_2;
                compensatory_registers_3(3) <= selected_output_data_3;
                compensatory_registers_4(3) <= selected_output_data_4;

                for i in 2 downto 0 loop

                    if (register_latency_pointer_0(i+1)='1') then
                          compensatory_registers_0(i) <= selected_output_data_0;
                    else
                          compensatory_registers_0(i) <= compensatory_registers_0(i+1);
                    end if;
                    if (register_latency_pointer_1(i+1)='1') then
                          compensatory_registers_1(i) <= selected_output_data_1;
                    else
                          compensatory_registers_1(i) <= compensatory_registers_1(i+1);
                    end if;
                    if (register_latency_pointer_2(i+1)='1') then
                          compensatory_registers_2(i) <= selected_output_data_2;
                    else
                          compensatory_registers_2(i) <= compensatory_registers_2(i+1);
                    end if;
                    if (register_latency_pointer_3(i+1)='1') then
                          compensatory_registers_3(i) <= selected_output_data_3;
                    else
                          compensatory_registers_3(i) <= compensatory_registers_3(i+1);
                    end if;
                    if (register_latency_pointer_4(i+1)='1') then
                          compensatory_registers_4(i) <= selected_output_data_4;
                    else
                          compensatory_registers_4(i) <= compensatory_registers_4(i+1);
                    end if;

                end loop;
        end if;
    end if;

end process;

primary_output_0_i <= compensatory_registers_0(0) when register_latency_pointer_0(0)='0' else selected_output_data_0;
primary_output_1_i <= compensatory_registers_1(0) when register_latency_pointer_1(0)='0' else selected_output_data_1;
primary_output_2_i <= compensatory_registers_2(0) when register_latency_pointer_2(0)='0' else selected_output_data_2;
primary_output_3_i <= compensatory_registers_3(0) when register_latency_pointer_3(0)='0' else selected_output_data_3;
primary_output_4_i <= compensatory_registers_4(0) when register_latency_pointer_4(0)='0' else selected_output_data_4;

output_register_impl : process(clk_vrc, reset_n)
begin

    if(reset_n='0') then
        primary_output_0 <= (others => '0');
        primary_output_1 <= (others => '0');
```

```
        primary_output_2 <= (others => '0');
        primary_output_3 <= (others => '0');
        primary_output_4 <= (others => '0');

    elsif(clk_vrc='1' and clk_vrc'event) then

        primary_output_0 <= primary_output_0_i;
        primary_output_1 <= primary_output_1_i;
        primary_output_2 <= primary_output_2_i;
        primary_output_3 <= primary_output_3_i;
        primary_output_4 <= primary_output_4_i;

    end if;

end process;

end;
```

Entity: *cell_column_input*

```
library ieee;

use ieee.std_logic_1164.all;
use work.vhdl_designer.all;

entity cell_column_input is
    generic
        (
        datawidth : positive := 8
        );
    port
        (
        -- primary data input
        input_primary_data_port : in  array_std_logic_vector(4 downto 0, datawidth-1 downto 0);

        -- column data output
        cell_output_port        : out array_std_logic_vector(4 downto 0, datawidth-1 downto 0);

        -- clock ports
        clk           : in std_logic;
        clk_config    : in std_logic;
        clk_en        : in std_logic;

        -- config ports
        column_config : in std_logic_vector(39 downto 0);
        wr_config_en  : in std_logic
        );
end;

architecture column_block of cell_column_input is

component cell_alu_gates is

    generic
        (
        datawidth : positive := 8
        );

    port
        (
        cell_input_data_A : in std_logic_vector(datawidth-1 downto 0);
        cell_input_data_B : in std_logic_vector(datawidth-1 downto 0);

        cell_output_data : out std_logic_vector(datawidth-1 downto 0);

        config : in std_logic_vector(1 downto 0)
        );
end component;

signal cell_alu_0_input_A : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_0_input_B : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_0_output : std_logic_vector(datawidth-1 downto 0);

signal cell_alu_1_input_A : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_1_input_B : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_1_output : std_logic_vector(datawidth-1 downto 0);

signal cell_alu_2_input_A : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_2_input_B : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_2_output : std_logic_vector(datawidth-1 downto 0);

signal cell_alu_3_input_A : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_3_input_B : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_3_output : std_logic_vector(datawidth-1 downto 0);

signal cell_alu_4_input_A : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_4_input_B : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_4_output : std_logic_vector(datawidth-1 downto 0);

signal column_config_reg : std_logic_vector(39 downto 0);

alias cell_alu_4_config : std_logic_vector(1 downto 0) is column_config_reg (39 downto 38);
alias mux_selector_cell_4_B : std_logic_vector(2 downto 0) is column_config_reg (37 downto 35);
alias mux_selector_cell_4_A : std_logic_vector(2 downto 0) is column_config_reg (34 downto 32);

alias cell_alu_3_config : std_logic_vector(1 downto 0) is column_config_reg (31 downto 30);
alias mux_selector_cell_3_B : std_logic_vector(2 downto 0) is column_config_reg (29 downto 27);
alias mux_selector_cell_3_A : std_logic_vector(2 downto 0) is column_config_reg (26 downto 24);
```

```
alias cell_alu_2_config : std_logic_vector(1 downto 0) is column_config_reg (23 downto 22);
alias mux_selector_cell_2_B : std_logic_vector(2 downto 0) is column_config_reg (21 downto 19);
alias mux_selector_cell_2_A : std_logic_vector(2 downto 0) is column_config_reg (18 downto 16);

alias cell_alu_1_config : std_logic_vector(1 downto 0) is column_config_reg (15 downto 14);
alias mux_selector_cell_1_B : std_logic_vector(2 downto 0) is column_config_reg (13 downto 11);
alias mux_selector_cell_1_A : std_logic_vector(2 downto 0) is column_config_reg (10 downto 8);

alias cell_alu_0_config : std_logic_vector(1 downto 0) is column_config_reg (7 downto 6);
alias mux_selector_cell_0_B : std_logic_vector(2 downto 0) is column_config_reg (5 downto 3);
alias mux_selector_cell_0_A : std_logic_vector(2 downto 0) is column_config_reg (2 downto 0);

begin

-- instances of cells
cell_alu_0: cell_alu_gates
    generic map
            (
            datawidth
            )
    port map
            (
            cell_input_data_A => cell_alu_0_input_A,
            cell_input_data_B => cell_alu_0_input_B,
            cell_output_data => cell_alu_0_output,
            config => cell_alu_0_config
            );

cell_alu_1: cell_alu_gates
    generic map
            (
            datawidth
            )
    port map
            (
            cell_input_data_A => cell_alu_1_input_A,
            cell_input_data_B => cell_alu_1_input_B,
            cell_output_data => cell_alu_1_output,
            config => cell_alu_1_config
            );

cell_alu_2: cell_alu_gates
    generic map
            (
            datawidth
            )
    port map
            (
            cell_input_data_A => cell_alu_2_input_A,
            cell_input_data_B => cell_alu_2_input_B,
            cell_output_data => cell_alu_2_output,
            config => cell_alu_2_config
            );

cell_alu_3: cell_alu_gates
    generic map
            (
            datawidth
            )
    port map
            (
            cell_input_data_A => cell_alu_3_input_A,
            cell_input_data_B => cell_alu_3_input_B,
            cell_output_data => cell_alu_3_output,
            config => cell_alu_3_config
            );

cell_alu_4: cell_alu_gates
    generic map
            (
            datawidth
            )
    port map
        (
            cell_input_data_A => cell_alu_4_input_A,
            cell_input_data_B => cell_alu_4_input_B,
            cell_output_data => cell_alu_4_output,
            config => cell_alu_4_config
            );

with mux_selector_cell_0_A select
    cell_alu_0_input_A <= to_std_logic_vector(input_primary_data_port,0) when "000",
            to_std_logic_vector(input_primary_data_port,1) when "001",
            to_std_logic_vector(input_primary_data_port,2) when "010",
            to_std_logic_vector(input_primary_data_port,3) when "011",
            to_std_logic_vector(input_primary_data_port,4) when others;


with mux_selector_cell_0_B select
    cell_alu_0_input_B <= to_std_logic_vector(input_primary_data_port,0) when "000",
            to_std_logic_vector(input_primary_data_port,1) when "001",
            to_std_logic_vector(input_primary_data_port,2) when "010",
            to_std_logic_vector(input_primary_data_port,3) when "011",
            to_std_logic_vector(input_primary_data_port,4) when others;


with mux_selector_cell_1_A select
    cell_alu_1_input_A <= to_std_logic_vector(input_primary_data_port,0) when "000",
            to_std_logic_vector(input_primary_data_port,1) when "001",
            to_std_logic_vector(input_primary_data_port,2) when "010",
            to_std_logic_vector(input_primary_data_port,3) when "011",
            to_std_logic_vector(input_primary_data_port,4) when others;
```

```
with mux_selector_cell_1_B select
    cell_alu_1_input_B <= to_std_logic_vector(input_primary_data_port,0) when "000",
            to_std_logic_vector(input_primary_data_port,1) when "001",
            to_std_logic_vector(input_primary_data_port,2) when "010",
            to_std_logic_vector(input_primary_data_port,3) when "011",
            to_std_logic_vector(input_primary_data_port,4) when others;

with mux_selector_cell_2_A select
    cell_alu_2_input_A <= to_std_logic_vector(input_primary_data_port,0) when "000",
            to_std_logic_vector(input_primary_data_port,1) when "001",
            to_std_logic_vector(input_primary_data_port,2) when "010",
            to_std_logic_vector(input_primary_data_port,3) when "011",
            to_std_logic_vector(input_primary_data_port,4) when others;

with mux_selector_cell_2_B select
    cell_alu_2_input_B <= to_std_logic_vector(input_primary_data_port,0) when "000",
            to_std_logic_vector(input_primary_data_port,1) when "001",
            to_std_logic_vector(input_primary_data_port,2) when "010",
            to_std_logic_vector(input_primary_data_port,3) when "011",
            to_std_logic_vector(input_primary_data_port,4) when others;

with mux_selector_cell_3_A select
    cell_alu_3_input_A <= to_std_logic_vector(input_primary_data_port,0) when "000",
            to_std_logic_vector(input_primary_data_port,1) when "001",
            to_std_logic_vector(input_primary_data_port,2) when "010",
            to_std_logic_vector(input_primary_data_port,3) when "011",
            to_std_logic_vector(input_primary_data_port,4) when others;

with mux_selector_cell_3_B select
    cell_alu_3_input_B <= to_std_logic_vector(input_primary_data_port,0) when "000",
            to_std_logic_vector(input_primary_data_port,1) when "001",
            to_std_logic_vector(input_primary_data_port,2) when "010",
            to_std_logic_vector(input_primary_data_port,3) when "011",
            to_std_logic_vector(input_primary_data_port,4) when others;

with mux_selector_cell_4_A select
    cell_alu_4_input_A <= to_std_logic_vector(input_primary_data_port,0) when "000",
            to_std_logic_vector(input_primary_data_port,1) when "001",
            to_std_logic_vector(input_primary_data_port,2) when "010",
            to_std_logic_vector(input_primary_data_port,3) when "011",
            to_std_logic_vector(input_primary_data_port,4) when others;

with mux_selector_cell_4_B select
    cell_alu_4_input_B <= to_std_logic_vector(input_primary_data_port,0) when "000",
            to_std_logic_vector(input_primary_data_port,1) when "001",
            to_std_logic_vector(input_primary_data_port,2) when "010",
            to_std_logic_vector(input_primary_data_port,3) when "011",
            to_std_logic_vector(input_primary_data_port,4) when others;

-- implementation of column output registers
output_register : process(clk)
begin

    if(clk='1' and clk'event) then
        if(clk_en='1') then
            to_array_std_logic_vector(cell_output_port, cell_alu_0_output, 0);
            to_array_std_logic_vector(cell_output_port, cell_alu_1_output, 1);
            to_array_std_logic_vector(cell_output_port, cell_alu_2_output, 2);
            to_array_std_logic_vector(cell_output_port, cell_alu_3_output, 3);
            to_array_std_logic_vector(cell_output_port, cell_alu_4_output, 4);
        end if;
    end if;
end process;

-- implementation of config. registers
config_register : process(clk_config)
begin

    if(clk_config='1' and clk_config'event) then
        if(wr_config_en='1') then
            column_config_reg <= column_config;
        end if;
    end if;
end process;

end;
```

Entity: *cell_column_type_0*

```
library ieee;

use ieee.std_logic_1164.all;
use work.vhdl_designer.all;

entity cell_column_type_0 is
    generic
        (
        datawidth : positive := 8
        );
    port
        (
        -- primary data input
        input_primary_data_port : in  array_std_logic_vector(4 downto 0, datawidth-1 downto 0);

        -- column data input
        input_cell_data_port    : in  array_std_logic_vector(4 downto 0, datawidth-1 downto 0);
```

```
        -- column data output
        cell_output_port        : out array_std_logic_vector(4 downto 0, datawidth-1 downto 0);

        -- clock ports
        clk        : in std_logic;
        clk_config : in std_logic;
        clk_en     : in std_logic;

        -- config ports
        column_config : in std_logic_vector(49 downto 0);
        wr_config_en  : in std_logic
        );
end;

architecture column_block of cell_column_type_0 is

component cell_alu_gates is

    generic
        (
        datawidth : positive := 8
        );

    port
        (
        cell_input_data_A : in std_logic_vector(datawidth-1 downto 0);
        cell_input_data_B : in std_logic_vector(datawidth-1 downto 0);

        cell_output_data : out std_logic_vector(datawidth-1 downto 0);

        config : in std_logic_vector(1 downto 0)
        );
end component;

signal cell_alu_0_input_A : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_0_input_B : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_0_output : std_logic_vector(datawidth-1 downto 0);

signal cell_alu_1_input_A : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_1_input_B : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_1_output : std_logic_vector(datawidth-1 downto 0);

signal cell_alu_2_input_A : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_2_input_B : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_2_output : std_logic_vector(datawidth-1 downto 0);

signal cell_alu_3_input_A : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_3_input_B : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_3_output : std_logic_vector(datawidth-1 downto 0);

signal cell_alu_4_input_A : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_4_input_B : std_logic_vector(datawidth-1 downto 0);
signal cell_alu_4_output : std_logic_vector(datawidth-1 downto 0);

signal column_config_reg : std_logic_vector(49 downto 0);

alias cell_alu_4_config : std_logic_vector(1 downto 0) is column_config_reg (49 downto 48);
alias mux_selector_cell_4_B : std_logic_vector(3 downto 0) is column_config_reg (47 downto 44);
alias mux_selector_cell_4_A : std_logic_vector(3 downto 0) is column_config_reg (43 downto 40);

alias cell_alu_3_config : std_logic_vector(1 downto 0) is column_config_reg (39 downto 38);
alias mux_selector_cell_3_B : std_logic_vector(3 downto 0) is column_config_reg (37 downto 34);
alias mux_selector_cell_3_A : std_logic_vector(3 downto 0) is column_config_reg (33 downto 30);

alias cell_alu_2_config : std_logic_vector(1 downto 0) is column_config_reg (29 downto 28);
alias mux_selector_cell_2_B : std_logic_vector(3 downto 0) is column_config_reg (27 downto 24);
alias mux_selector_cell_2_A : std_logic_vector(3 downto 0) is column_config_reg (23 downto 20);

alias cell_alu_1_config : std_logic_vector(1 downto 0) is column_config_reg (19 downto 18);
alias mux_selector_cell_1_B : std_logic_vector(3 downto 0) is column_config_reg (17 downto 14);
alias mux_selector_cell_1_A : std_logic_vector(3 downto 0) is column_config_reg (13 downto 10);

alias cell_alu_0_config : std_logic_vector(1 downto 0) is column_config_reg (9 downto 8);
alias mux_selector_cell_0_B : std_logic_vector(3 downto 0) is column_config_reg (7 downto 4);
alias mux_selector_cell_0_A : std_logic_vector(3 downto 0) is column_config_reg (3 downto 0);


begin

-- instances of cells
cell_alu_0: cell_alu_gates
    generic map
            (
            datawidth
            )
    port map
            (
            cell_input_data_A => cell_alu_0_input_A,
            cell_input_data_B => cell_alu_0_input_B,
            cell_output_data => cell_alu_0_output,
            config => cell_alu_0_config
            );
cell_alu_1: cell_alu_gates
    generic map
            (
            datawidth
            )
    port map
            (
            cell_input_data_A => cell_alu_1_input_A,
            cell_input_data_B => cell_alu_1_input_B,
            cell_output_data => cell_alu_1_output,
            config => cell_alu_1_config
```

```
            );
cell_alu_2: cell_alu_gates
    generic map
            (
            datawidth
            )
    port map
            (
            cell_input_data_A => cell_alu_2_input_A,
            cell_input_data_B => cell_alu_2_input_B,
            cell_output_data => cell_alu_2_output,
            config => cell_alu_2_config
            );
cell_alu_3: cell_alu_gates
    generic map
            (
            datawidth
            )
    port map
            (
            cell_input_data_A => cell_alu_3_input_A,
            cell_input_data_B => cell_alu_3_input_B,
            cell_output_data => cell_alu_3_output,
            config => cell_alu_3_config
            );
cell_alu_4: cell_alu_gates
    generic map
            (
            datawidth
            )
    port map
            (
            cell_input_data_A => cell_alu_4_input_A,
            cell_input_data_B => cell_alu_4_input_B,
            cell_output_data => cell_alu_4_output,
            config => cell_alu_4_config
            );

with mux_selector_cell_0_A select
    cell_alu_0_input_A <= to_std_logic_vector(input_primary_data_port,0) when "0000",
            to_std_logic_vector(input_primary_data_port,1) when "0001",
            to_std_logic_vector(input_primary_data_port,2) when "0010",
            to_std_logic_vector(input_primary_data_port,3) when "0011",
            to_std_logic_vector(input_primary_data_port,4) when "0100",
            to_std_logic_vector(input_cell_data_port,0) when "0101",
            to_std_logic_vector(input_cell_data_port,1) when "0110",
            to_std_logic_vector(input_cell_data_port,2) when "0111",
            to_std_logic_vector(input_cell_data_port,3) when "1000",
            to_std_logic_vector(input_cell_data_port,4) when others;

with mux_selector_cell_0_B select
    cell_alu_0_input_B <= to_std_logic_vector(input_primary_data_port,0) when "0000",
            to_std_logic_vector(input_primary_data_port,1) when "0001",
            to_std_logic_vector(input_primary_data_port,2) when "0010",
            to_std_logic_vector(input_primary_data_port,3) when "0011",
            to_std_logic_vector(input_primary_data_port,4) when "0100",
            to_std_logic_vector(input_cell_data_port,0) when "0101",
            to_std_logic_vector(input_cell_data_port,1) when "0110",
            to_std_logic_vector(input_cell_data_port,2) when "0111",
            to_std_logic_vector(input_cell_data_port,3) when "1000",
            to_std_logic_vector(input_cell_data_port,4) when others;

with mux_selector_cell_1_A select
    cell_alu_1_input_A <= to_std_logic_vector(input_primary_data_port,0) when "0000",
            to_std_logic_vector(input_primary_data_port,1) when "0001",
            to_std_logic_vector(input_primary_data_port,2) when "0010",
            to_std_logic_vector(input_primary_data_port,3) when "0011",
            to_std_logic_vector(input_primary_data_port,4) when "0100",
            to_std_logic_vector(input_cell_data_port,0) when "0101",
            to_std_logic_vector(input_cell_data_port,1) when "0110",
            to_std_logic_vector(input_cell_data_port,2) when "0111",
            to_std_logic_vector(input_cell_data_port,3) when "1000",
            to_std_logic_vector(input_cell_data_port,4) when others;

with mux_selector_cell_1_B select
    cell_alu_1_input_B <= to_std_logic_vector(input_primary_data_port,0) when "0000",
            to_std_logic_vector(input_primary_data_port,1) when "0001",
            to_std_logic_vector(input_primary_data_port,2) when "0010",
            to_std_logic_vector(input_primary_data_port,3) when "0011",
            to_std_logic_vector(input_primary_data_port,4) when "0100",
            to_std_logic_vector(input_cell_data_port,0) when "0101",
            to_std_logic_vector(input_cell_data_port,1) when "0110",
            to_std_logic_vector(input_cell_data_port,2) when "0111",
            to_std_logic_vector(input_cell_data_port,3) when "1000",
            to_std_logic_vector(input_cell_data_port,4) when others;

with mux_selector_cell_2_A select
    cell_alu_2_input_A <= to_std_logic_vector(input_primary_data_port,0) when "0000",
            to_std_logic_vector(input_primary_data_port,1) when "0001",
            to_std_logic_vector(input_primary_data_port,2) when "0010",
            to_std_logic_vector(input_primary_data_port,3) when "0011",
            to_std_logic_vector(input_primary_data_port,4) when "0100",
            to_std_logic_vector(input_cell_data_port,0) when "0101",
            to_std_logic_vector(input_cell_data_port,1) when "0110",
            to_std_logic_vector(input_cell_data_port,2) when "0111",
            to_std_logic_vector(input_cell_data_port,3) when "1000",
            to_std_logic_vector(input_cell_data_port,4) when others;

with mux_selector_cell_2_B select
    cell_alu_2_input_B <= to_std_logic_vector(input_primary_data_port,0) when "0000",
            to_std_logic_vector(input_primary_data_port,1) when "0001",
            to_std_logic_vector(input_primary_data_port,2) when "0010",
            to_std_logic_vector(input_primary_data_port,3) when "0011",
```

```
            to_std_logic_vector(input_primary_data_port,4) when "0100",
            to_std_logic_vector(input_cell_data_port,0) when "0101",
            to_std_logic_vector(input_cell_data_port,1) when "0110",
            to_std_logic_vector(input_cell_data_port,2) when "0111",
            to_std_logic_vector(input_cell_data_port,3) when "1000",
            to_std_logic_vector(input_cell_data_port,4) when others;

with mux_selector_cell_3_A select
    cell_alu_3_input_A <= to_std_logic_vector(input_primary_data_port,0) when "0000",
            to_std_logic_vector(input_primary_data_port,1) when "0001",
            to_std_logic_vector(input_primary_data_port,2) when "0010",
            to_std_logic_vector(input_primary_data_port,3) when "0011",
            to_std_logic_vector(input_primary_data_port,4) when "0100",
            to_std_logic_vector(input_cell_data_port,0) when "0101",
            to_std_logic_vector(input_cell_data_port,1) when "0110",
            to_std_logic_vector(input_cell_data_port,2) when "0111",
            to_std_logic_vector(input_cell_data_port,3) when "1000",
            to_std_logic_vector(input_cell_data_port,4) when others;

with mux_selector_cell_3_B select
    cell_alu_3_input_B <= to_std_logic_vector(input_primary_data_port,0) when "0000",
            to_std_logic_vector(input_primary_data_port,1) when "0001",
            to_std_logic_vector(input_primary_data_port,2) when "0010",
            to_std_logic_vector(input_primary_data_port,3) when "0011",
            to_std_logic_vector(input_primary_data_port,4) when "0100",
            to_std_logic_vector(input_cell_data_port,0) when "0101",
            to_std_logic_vector(input_cell_data_port,1) when "0110",
            to_std_logic_vector(input_cell_data_port,2) when "0111",
            to_std_logic_vector(input_cell_data_port,3) when "1000",
            to_std_logic_vector(input_cell_data_port,4) when others;

with mux_selector_cell_4_A select
    cell_alu_4_input_A <= to_std_logic_vector(input_primary_data_port,0) when "0000",
            to_std_logic_vector(input_primary_data_port,1) when "0001",
            to_std_logic_vector(input_primary_data_port,2) when "0010",
            to_std_logic_vector(input_primary_data_port,3) when "0011",
            to_std_logic_vector(input_primary_data_port,4) when "0100",
            to_std_logic_vector(input_cell_data_port,0) when "0101",
            to_std_logic_vector(input_cell_data_port,1) when "0110",
            to_std_logic_vector(input_cell_data_port,2) when "0111",
            to_std_logic_vector(input_cell_data_port,3) when "1000",
            to_std_logic_vector(input_cell_data_port,4) when others;

with mux_selector_cell_4_B select
    cell_alu_4_input_B <= to_std_logic_vector(input_primary_data_port,0) when "0000",
            to_std_logic_vector(input_primary_data_port,1) when "0001",
            to_std_logic_vector(input_primary_data_port,2) when "0010",
            to_std_logic_vector(input_primary_data_port,3) when "0011",
            to_std_logic_vector(input_primary_data_port,4) when "0100",
            to_std_logic_vector(input_cell_data_port,0) when "0101",
            to_std_logic_vector(input_cell_data_port,1) when "0110",
            to_std_logic_vector(input_cell_data_port,2) when "0111",
            to_std_logic_vector(input_cell_data_port,3) when "1000",
            to_std_logic_vector(input_cell_data_port,4) when others;

-- implementation of output registers
output_register : process(clk)
begin
    if(clk='1' and clk'event) then
        if(clk_en='1') then
            to_array_std_logic_vector(cell_output_port, cell_alu_0_output, 0);
            to_array_std_logic_vector(cell_output_port, cell_alu_1_output, 1);
            to_array_std_logic_vector(cell_output_port, cell_alu_2_output, 2);
            to_array_std_logic_vector(cell_output_port, cell_alu_3_output, 3);
            to_array_std_logic_vector(cell_output_port, cell_alu_4_output, 4);
        end if;
    end if;
end process;

-- implementation of config. registers
config_register : process(clk_config)
begin
    if(clk_config='1' and clk_config'event) then
        if(wr_config_en='1') then
            column_config_reg <= column_config;
        end if;
    end if;
end process;

end;
```

Entity: *cell_alu_gates*

```
library ieee;

use ieee.std_logic_1164.all;

entity cell_alu_gates is
    generic
        (
        datawidth : positive := 8
        );
    port
        (
        -- cell/node inputs
        cell_input_data_A : in std_logic_vector(datawidth-1 downto 0);
        cell_input_data_B : in std_logic_vector(datawidth-1 downto 0);

        -- cell/node output
        cell_output_data : out std_logic_vector(datawidth-1 downto 0);
```

```
        -- cell/node configuration - function gene
        config : in std_logic_vector(1 downto 0)
        );
end;

architecture functions_multiplexer of cell_alu_gates is

-- components of fundamental functions
component func_xor is

    generic
        (
        datawidth : positive := 8
        );
    port
        (
        input_data_A : in std_logic_vector(datawidth-1 downto 0);
        input_data_B : in std_logic_vector(datawidth-1 downto 0);

        output_data : out std_logic_vector(datawidth-1 downto 0)
        );
end component;

component func_or is

    generic
        (
        datawidth : positive := 8
        );
    port
        (
        input_data_A : in std_logic_vector(datawidth-1 downto 0);
        input_data_B : in std_logic_vector(datawidth-1 downto 0);

        output_data : out std_logic_vector(datawidth-1 downto 0)
        );
end component;

component func_and is

    generic
        (
        datawidth : positive := 8
        );
    port
        (
        input_data_A : in std_logic_vector(datawidth-1 downto 0);
        input_data_B : in std_logic_vector(datawidth-1 downto 0);

        output_data : out std_logic_vector(datawidth-1 downto 0)
        );
end component;

component func_ident is

    generic
        (
        datawidth : positive := 8
        );
    port
        (
        input_data_A : in std_logic_vector(datawidth-1 downto 0);
        input_data_B : in std_logic_vector(datawidth-1 downto 0);

        output_data : out std_logic_vector(datawidth-1 downto 0)
        );
end component;

signal output_func_xor    : std_logic_vector(datawidth-1 downto 0);
signal output_func_or     : std_logic_vector(datawidth-1 downto 0);
signal output_func_and    : std_logic_vector(datawidth-1 downto 0);
signal output_func_ident  : std_logic_vector(datawidth-1 downto 0);

begin

-- instances of fundamentral functions
function_0 : func_xor

    generic map
        ( datawidth )

    port map
        (
        input_data_A => cell_input_data_A,
        input_data_B => cell_input_data_B,
        output_data  => output_func_xor
        );

function_1 : func_or

    generic map
        ( datawidth )

    port map
        (
        input_data_A => cell_input_data_A,
        input_data_B => cell_input_data_B,
        output_data  => output_func_or
        );

function_2 : func_and
```

```
    generic map
        ( datawidth )

    port map
        (
        input_data_A => cell_input_data_A,
        input_data_B => cell_input_data_B,
        output_data  => output_func_and
        );

function_3 : func_ident

    generic map
        ( datawidth )

    port map
        (
        input_data_A => cell_input_data_A,
        input_data_B => cell_input_data_B,
        output_data => output_func_ident
        );

with config select
    cell_output_data <=   output_func_xor     when "00",
                          output_func_or      when "01",
                          output_func_and     when "10",
                          output_func_ident   when others;

end;
```

C.

Source code – Column Detector

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.log2;
use ieee.math_real.ceil;

entity column_detect is

     generic
            (
             NUMBER_CELL            : positive;
             COLUMN_OUTPUT_EN       : boolean;
             COLUMN_PRIM_INPUT_EN   : boolean;
             NUMBER_PRIM_INPUT      : positive;
             ROUTE_CONFIG_WIDTH     : positive;
             FUNCT_CONFIG_WIDTH     : positive;
             LAST_COLUMN            : boolean
            );
     port
        (
    column_config : in std_logic_vector(((ROUTE_CONFIG_WIDTH*2*NUMBER_CELL)+(FUNCT_CONFIG_WIDTH*NUMBER_CELL))-1 downto 0);
    output_selected              : in  std_logic_vector(NUMBER_CELL-1 downto 0);
    previous_act_cell_propagation : in  std_logic_vector(NUMBER_CELL-1 downto 0);
    act_cell_propagation         : out std_logic_vector(NUMBER_CELL-1 downto 0);

    act_cell_flag                : out std_logic_vector(NUMBER_CELL-1 downto 0);
    number_act_cell              : out std_logic_vector(natural(ceil(log2(real(NUMBER_CELL))))-1 downto 0)
         );
end;

architecture rtl of column_detect is

-- type
type route_act_t  is array (0 to NUMBER_CELL-1) of std_logic_vector(NUMBER_CELL-1 downto 0);
type route_conf_t is array (0 to NUMBER_CELL-1) of unsigned(ROUTE_CONFIG_WIDTH-1 downto 0);

-- signals
signal routed_act_cell : route_act_t;
signal act_cell        : std_logic_vector(NUMBER_CELL-1 downto 0);
signal temp_0          : route_conf_t;
signal temp_1          : route_conf_t;
signal temp_limited_0  : route_conf_t;
signal temp_limited_1  : route_conf_t;

begin

input_act_cell_info_0 : if (COLUMN_OUTPUT_EN and (not LAST_COLUMN)) generate

   act_cell <= output_selected or previous_act_cell_propagation;

end generate;

input_act_cell_info_1 : if(COLUMN_OUTPUT_EN and LAST_COLUMN) generate

   act_cell <= output_selected;

end generate;

input_act_cell_info_2 : if((not COLUMN_OUTPUT_EN) and (not LAST_COLUMN)) generate

   act_cell <= previous_act_cell_propagation;

end generate;

route_act_primary_input_en : if(COLUMN_PRIM_INPUT_EN) generate

   route_cell_i:   for i in 0 to (NUMBER_CELL-1) generate

      temp_0(i) <=  unsigned((column_config(((i+1)*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH)))-1 downto
      FUNCT_CONFIG_WIDTH+ROUTE_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH))))));

      temp_1(i) <= unsigned((column_config((FUNCT_CONFIG_WIDTH+ROUTE_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+
      (2*ROUTE_CONFIG_WIDTH))))-1 downto FUNCT_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH))))));

      process(temp_0)
      begin

         if(temp_0(i)>(NUMBER_CELL+NUMBER_PRIM_INPUT-1)) then
            temp_limited_0(i) <= to_unsigned(NUMBER_CELL+NUMBER_PRIM_INPUT-1,ROUTE_CONFIG_WIDTH);
         else
            temp_limited_0(i) <= temp_0(i);
         end if;

      end process;

      process(temp_1)
      begin
         if(temp_1(i)>(NUMBER_CELL+NUMBER_PRIM_INPUT-1)) then
            temp_limited_1(i) <= to_unsigned(NUMBER_CELL+NUMBER_PRIM_INPUT-1,ROUTE_CONFIG_WIDTH);
         else
            temp_limited_1(i) <= temp_1(i);
         end if;
      end process;
```

```
      route_cell_j:   for j in 0 to (NUMBER_CELL-1) generate

         routed_act_cell(i)(j) <= act_cell(j) when (temp_limited_0(j)=(i+NUMBER_PRIM_INPUT)) or
                                  (temp_limited_1(j)=(i+NUMBER_PRIM_INPUT)) else '0';

      end generate;

   end generate;

end generate;

route_act_primary_input_not_en : if(not COLUMN_PRIM_INPUT_EN) generate

   route_cell_i:   for i in 0 to (NUMBER_CELL-1) generate

   temp_0(i) <= unsigned((column_config(((i+1)*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH)))-1 downto
   FUNCT_CONFIG_WIDTH+ROUTE_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH))))));

   temp_1(i) <= unsigned((column_config((FUNCT_CONFIG_WIDTH+ROUTE_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+
   (2*ROUTE_CONFIG_WIDTH))))-1 downto FUNCT_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH))))));

   process(temp_0)
   begin

      if(temp_0(i)>(NUMBER_CELL-1)) then
         temp_limited_0(i) <= to_unsigned(NUMBER_CELL-1,ROUTE_CONFIG_WIDTH);
      else
         temp_limited_0(i) <= temp_0(i);
      end if;

   end process;

   process(temp_1)
   begin

      if(temp_1(i)>(NUMBER_CELL-1)) then
         temp_limited_1(i) <= to_unsigned(NUMBER_CELL-1,ROUTE_CONFIG_WIDTH);
      else
         temp_limited_1(i) <= temp_1(i);
      end if;

   end process;

      route_cell_j:   for j in 0 to (NUMBER_CELL-1) generate
         routed_act_cell(i)(j) <= act_cell(j) when (temp_limited_0(j)=i) or (temp_limited_1(j)=i) else '0';
      end generate;

   end generate;

end generate;

propagation_out_generate : for i in 0 to (NUMBER_CELL-1) generate

   act_cell_propagation(i) <= '1' when unsigned(routed_act_cell(i))/=0 else '0';

end generate;

act_cell_flag <= act_cell;

process (act_cell)
   variable cnt_tmp : unsigned(natural(ceil(log2(real(NUMBER_CELL))))-1 downto 0);
begin

   cnt_tmp := (others => '0');

   for i in act_cell'range loop
      if(act_cell(i)='1') then
         cnt_tmp := cnt_tmp + 1;
      end if;
   end loop;

   number_act_cell <= std_logic_vector(cnt_tmp);

end process;

end;
```

D.

Source code – Column Detector L2

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.log2;
use ieee.math_real.ceil;

entity column_detect_l2 is
     generic
            (
             NUMBER_CELL            : positive;
             COLUMN_OUTPUT_EN       : boolean;
             COLUMN_PRIM_INPUT_EN   : boolean;
             NUMBER_PRIM_INPUT      : positive;
             ROUTE_CONFIG_WIDTH     : positive;
             FUNCT_CONFIG_WIDTH     : positive;
             LAST_COLUMN            : boolean
            );
     port
          (
    clk : in std_logic;

    column_config : in std_logic_vector(((ROUTE_CONFIG_WIDTH*2*NUMBER_CELL)+(FUNCT_CONFIG_WIDTH*NUMBER_CELL))-1 downto 0);
    output_selected                 : in  std_logic_vector(NUMBER_CELL-1 downto 0);
    previous_act_cell_propagation_l1 : in  std_logic_vector(NUMBER_CELL-1 downto 0);
    previous_act_cell_propagation_l2 : in  std_logic_vector(NUMBER_CELL-1 downto 0);
    act_cell_propagation_l1          : out std_logic_vector(NUMBER_CELL-1 downto 0);
    act_cell_propagation_l2          : out std_logic_vector(NUMBER_CELL-1 downto 0);

    act_cell_flag                   : out std_logic_vector(NUMBER_CELL-1 downto 0);
    number_act_cell                 : out std_logic_vector(natural(ceil(log2(real(NUMBER_CELL))))-1 downto 0)
         );
end;

architecture rtl of column_detect_l2 is

-- type
type route_act_t is array (0 to NUMBER_CELL-1) of std_logic_vector(NUMBER_CELL-1 downto 0);
type route_conf_t is array (0 to NUMBER_CELL-1) of unsigned(ROUTE_CONFIG_WIDTH-1 downto 0);

-- signals
signal routed_act_cell_l1     : route_act_t;
signal routed_act_cell_reg_l1 : route_act_t;
signal routed_act_cell_l2     : route_act_t;
signal routed_act_cell_reg_l2 : route_act_t;

signal act_cell : std_logic_vector(NUMBER_CELL-1 downto 0);

signal temp_0 : route_conf_t;
signal temp_1 : route_conf_t;

signal temp_limited_0 : route_conf_t;
signal temp_limited_1 : route_conf_t;

begin

input_act_cell_info_0 : if (COLUMN_OUTPUT_EN and (not LAST_COLUMN)) generate

   act_cell <= output_selected or previous_act_cell_propagation_l1 or previous_act_cell_propagation_l2;

end generate;

input_act_cell_info_1 : if(COLUMN_OUTPUT_EN and LAST_COLUMN) generate

   act_cell <= output_selected;

end generate;

input_act_cell_info_2 : if((not COLUMN_OUTPUT_EN) and (not LAST_COLUMN)) generate

   act_cell <= previous_act_cell_propagation_l1 or previous_act_cell_propagation_l2;

end generate;

route_act_primary_input_en : if(COLUMN_PRIM_INPUT_EN) generate

   route_cell_i:   for i in 0 to (NUMBER_CELL-1) generate

      temp_0(i) <= unsigned((column_config(((i+1)*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH)))-1 downto
                   FUNCT_CONFIG_WIDTH+ROUTE_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH))))));

      temp_1(i) <= unsigned((column_config((FUNCT_CONFIG_WIDTH+ROUTE_CONFIG_WIDTH+(i*
                   (FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH))))-1 downto
                    FUNCT_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH))))));

      process(temp_0)
      begin

         if(temp_0(i)>((NUMBER_CELL*2)+NUMBER_PRIM_INPUT-1)) then
            temp_limited_0(i) <= to_unsigned((NUMBER_CELL*2)+NUMBER_PRIM_INPUT-1,ROUTE_CONFIG_WIDTH);
         else
            temp_limited_0(i) <= temp_0(i);
         end if;
```

```
      end process;

      process(temp_1)
      begin

         if(temp_1(i)>((NUMBER_CELL*2)+NUMBER_PRIM_INPUT-1)) then
            temp_limited_1(i) <= to_unsigned((NUMBER_CELL*2)+NUMBER_PRIM_INPUT-1,ROUTE_CONFIG_WIDTH);
         else
            temp_limited_1(i) <= temp_1(i);
         end if;

      end process;

      route_cell_j:   for j in 0 to (NUMBER_CELL-1) generate

         routed_act_cell_l1(i)(j) <= '1' when (temp_limited_0(j)=(i+NUMBER_PRIM_INPUT+NUMBER_CELL)) or
                                     (temp_limited_1(j)=(i+NUMBER_PRIM_INPUT+NUMBER_CELL)) else '0';

         routed_act_cell_l2(i)(j) <= '1' when (temp_limited_0(j)=(i+NUMBER_PRIM_INPUT)) or
                                     (temp_limited_1(j)=(i+NUMBER_PRIM_INPUT)) else '0';

      end generate;

   end generate;

end generate;

route_act_primary_input_not_en : if(not COLUMN_PRIM_INPUT_EN) generate

   route_cell_i:   for i in 0 to (NUMBER_CELL-1) generate

      temp_0(i) <= unsigned((column_config(((i+1)*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH)))-1 downto
                   FUNCT_CONFIG_WIDTH+ROUTE_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH))))));

      temp_1(i) <= unsigned((column_config((FUNCT_CONFIG_WIDTH+ROUTE_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+(2*
                   ROUTE_CONFIG_WIDTH))))-1 downto FUNCT_CONFIG_WIDTH+(i*(FUNCT_CONFIG_WIDTH+(2*ROUTE_CONFIG_WIDTH))))));

      process(temp_0)
      begin

         if(temp_0(i)>((NUMBER_CELL*2)-1)) then
            temp_limited_0(i) <= to_unsigned((NUMBER_CELL*2)-1,ROUTE_CONFIG_WIDTH);
         else
            temp_limited_0(i) <= temp_0(i);
         end if;

      end process;

      process(temp_1)
      begin

         if(temp_1(i)>((NUMBER_CELL*2)-1)) then
            temp_limited_1(i) <= to_unsigned((NUMBER_CELL*2)-1,ROUTE_CONFIG_WIDTH);
         else
            temp_limited_1(i) <= temp_1(i);
         end if;

      end process;

      route_cell_j_1:   for j in 0 to (NUMBER_CELL-1) generate

         routed_act_cell_l1(i)(j) <= '1' when (temp_limited_0(j)=(i+NUMBER_CELL)) or (temp_limited_1(j)=(i+NUMBER_CELL))
                                     else '0';

         routed_act_cell_l2(i)(j) <= '1' when (temp_limited_0(j)=i) or (temp_limited_1(j)=i) else '0';

      end generate;

   end generate;

end generate;

process(clk)
begin

    if(clk='1' and clk'event) then
      routed_act_cell_reg_l1 <= routed_act_cell_l1;
      routed_act_cell_reg_l2 <= routed_act_cell_l2;
    end if;

end process;

propagation_out_generate_l2    : for i in 0 to (NUMBER_CELL-1) generate

   act_cell_propagation_l2(i) <= '1' when (unsigned(act_cell and routed_act_cell_reg_l2(i))/=0) else '0';

end generate;

propagation_out_generate_l1    : for i in 0 to (NUMBER_CELL-1) generate

   act_cell_propagation_l1(i) <= '1' when (unsigned(act_cell and routed_act_cell_reg_l1(i))/=0) else '0';

end generate;

act_cell_flag <= act_cell;

process (act_cell)
   variable cnt_tmp : unsigned(natural(ceil(log2(real(NUMBER_CELL))))-1 downto 0);
begin

   cnt_tmp := (others => '0');

   for i in act_cell'range loop
```

```
      if(act_cell(i)='1') then
         cnt_tmp := cnt_tmp + 1;
      end if;
   end loop;

number_act_cell <= std_logic_vector(cnt_tmp);

end process;

end;
```