# EVOLVABLE HARDWARE DESIGN OF COMBINATIONAL LOGIC CIRCUITS

By

Tatiana G. Kalganova

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
NAPIER UNIVERSITY
EDINBURGH, SCOTLAND
MAY 2000

## DECLARATION OF ORIGINALITY

I hereby declare that this thesis and the work contained herein was composed and originated entirely by myself, other than those items acknowledged in the text. The work was completed under a full-time supervised program at Napier University between September 1997 and May 2000.

Some materials presented in the thesis has been published as follows: Chapter 3 in [1]; Chapter 4 in [1], [2], [3]; Chapter 5 in [4], [5]; Chapter 6 in [6]; Chapter 7 in [7], [8], [9].

Some peripheral but related work, not described in this thesis, has been published in [2].

Dated: May 2000

Tatiana G. Kalganova

*To my family.*

# Abstract

Evolvable Hardware (EHW), as an alternative method for logic design, became more attractive recently, because of its algebra-independent techniques for generating self-adaptive self-reconfigurable hardware. This thesis investigates and relates both evaluation and evolutionary processes, emphasizing the need to address problems arising from data complexity.

Evaluation processes, capable of evolving cost-optimised fully functional circuits are investigated. The need for an extrinsic EHW approach (software models) independent of the concerns of any implementation technologies is emphasized. It is also shown how the function description may be adapted for use in the EHW approach. A number of issues of evaluation process are addressed: these include choice of optimisation criteria, multi-objective optimisation techniques in EHW and probabilistic analysis of evolutionary processes.

The concept of self-adaptive extrinsic EHW method is developed. This approach emphasizes the circuit layout evolution together with circuit functionality. A chromosome representation for such system is introduced, and a number of genetic operators and evolutionary algorithms in support of this approach are presented. The genetic operators change the genetic material at the different levels of chromosome representation. Furthermore, a chromosome representation is adapted to the function-level EHW approach. As a result, the modularised systems are evolved using multi-output building blocks. This chromosome representation overcomes the problem of long string chromosome.

Together, these techniques facilitate the construction of systems to evolve logic functions of large number of variables. A method for achieving this using bidirectional incremental evolution is documented. It is demonstrated that the integration of a dynamic evaluation process and self-adaptive function-level EHW approach allows the bidirectional incremental evolution to successfully evolve more complex systems than traditionally evolved before. Thereby it provides a firm foundation for the evolution of complex systems.

Finally, the universality of these techniques is proved by applying them to multi-valued combinational logic design. Empirical study of this application shows that there is no fundamental difference in approach for both binary and multi-valued logic design problems.

# Acknowledgments

First and foremost I wish to express my thanks to and acknowledge the assistance of all those people without whom this thesis would not exist.

Firstly thanks are due to my two supervisors, Prof. T.C. Fogarty (South Bank University) and Prof. A. Almaini (Napier University) who supervised the work reported here.

Thanks are due to the School of Computing at Napier University and EvoNet office for providing support for this work under my PhD student grant. I would also like to give special thanks to Dr. J. Miller who originated the approach that I extended and investigated.

I would also like to thank the following who helped me in various ways during my life as a PhD student: L. Miramontes Hercog (South Bank University), N. Gonet (University of Edinburgh), S.-H. Choi (Napier University), J. Bautista Ortega (Rosslin Institute, University of Edinburgh), A. Stoica (NASA), J. Willies (EvoNet), Marjory and Geoff Horne, Christine and Paul Deponio. Thanks also to my parents for support, Galina Kalganova (my mother) and Gennady Kalganov (my father) for emotional support during the writing of this dissertation, and my nephews, Sasha and Pasha Lipnitskie, for providing such a wonderful distraction during my visit to Belarus. Special thanks to my sister, Natalia Lipnitskaya for our long chats and discussion, for emotional support that I was getting from her during time of writing this work. Finally and especially I would like to thank Olivier Fouquet for his encouragement and understanding.

# Table of Contents

# List of Tables

# List of Figures

# List of Symbols

There follows the lists of symbols and abbreviations used in this thesis.

## Abbreviations

| | |
|---|---|
| BIE | Bidirectional Incremental Evolution |
| BGM | Boundary Geometry Mutation |
| CL | Circuit Layout |
| CMOS | Complementary metal oxide semiconductor |
| DMOS | Dynamic CMOS |
| EA | Evolutionary Algorithm |
| EHW | Evolvable Hardware |
| ES | Evolutionary Strategy |
| Eq. | Equation |
| FA | Full Adder |
| Fig. | Figure |
| FET | Field-Effect Transistor |
| FPGA | Field Programmable Gate Array |
| HW | Hardware |
| GA | Genetic Algorithm |

| | |
|---|---|
| GGM | Global Geometry Mutation |
| GM | Geometry Mutation |
| GP | Genetic Programming |
| HA | Half Adder |
| LGM | Local Geometry Mutation |
| mult2 | a 2-bit multiplier |
| mult3 | a 3-bit multiplier |
| MVL | Multi-Valued Logic |
| PLA | Programmable Logic Array |
| SW | Software |
| VES | Variable-length chromosome ES |
| VGA | Variable-length chromosome GA |

## Symbols

| | |
|---|---|
| $y(X)$ | the logic function |
| $r$ | the radix of logic; $r = 2$ corresponds to the Boolean logic |
| $n$ | the number of variables in the logic function |
| $m$ | the number of outputs in the logic function |
| $p$ | the number of input combinations in the logic function; $p = r^n$ corresponds to the completely specified logic function; $p < r^n$ defines that the function is incompletely specified |
| $X$ | the set of input variables, $X = \{x_0, x_1, ..., x_{n-1}\}$ |
| $Y$ | the set of output variables, $Y = \{y_0, y_1, ..., y_{m-1}\}$ |
| $\mathcal{F}$ | the fitness function |
| $\mathcal{F}_1 + \mathcal{F}_2$ | the dynamic fitness function |

| | |
|---|---|
| $F_1$ | the percentage of correct output bits in evolved circuit |
| $F_2$ | the number of active primitive logic gates in the circuit |
| $F_3$ | the percentage of correct output combinations in evolved circuit |
| $F_4$ | the cost of circuit in terms of the number of used transistors, resistors, capacitors, etc. |
| $F_i^{bc}$ | the $i$-th fitness criteria of the best evolved chromosome |
| $\overline{F_i^{bc}}$ | the mean $i$-th fitness criteria of the best chromosome |
| $\overline{F_2(\mathcal{N}_f)}$ | the mean fitness function $F_2$ of fully functional designs evolved during R runs |
| $\mathbb{FS}$ | the functional set of logic gates |
| $\|\mathcal{FS}\|$ | the number of elements in the functional set of logic gates, $\mathbb{FS}$ |
| $\mathcal{N}$ | the circuit or network |
| $\mathcal{N}_f$ | the fully functional circuit |
| $cost(\mathcal{N})$ | the cost or size of the circuit $\mathcal{N}$ |
| $cost(\mathcal{N}^{max})$ | the maximum cost of the circuit $\mathcal{N}$ |
| $\mathcal{B}_z$ | the building block $\mathcal{B}$ labeled $z$ |
| $cost(\mathcal{B}_z)$ | the cost or size of the the building block $\mathcal{B}_z$ |
| $\mathcal{B}(c_{col}, c_{row})$ | the building block $\mathcal{B}$ located in column $c_{col}$ and row $c_{row}$ |
| $\mathbb{B}$ | the rectangular array of the logic building blocks $\{\mathcal{B}_{c_{col}c_{row}} : \{\mathcal{B}_{c_{col}c_{row}} \in \mathbb{B},\ c_{col} = \{0, \cdots, N_{cols} - 1\},\ c_{row} = \{0, \cdots, N_{rows} - 1\}\}\}$ |
| $N_{cols}^{max}, N_{rows}^{max}$ | the maximum number of columns and rows in the rectangular array respectively |
| $N_{cols}, N_{rows}$ | the number of columns and rows in rectangular array, respectively, $N_{cols} \in \{1, \cdots, N_{cols}^{max}\}$ and $N_{rows} \in \{1, \cdots, N_{rows}^{max}\}$ |
| $N_{cols} \times N_{rows}$ | the circuit layout |
| $N_{connect}$ | the connectivity parameter representing the number of columns on the |

| | |
|---|---|
| | left to which a cell in a particular column $c_{col}$ or an output may be connected and $N_{connect} \in \{1, \cdots, N_{cols}\}$ |
| $N_{in}^{max}, N_{out}^{max}$ | the maximum number of inputs and outputs in any building block respectively |
| $N_{in}(\mathcal{B}), N_{out}(\mathcal{B})$ | the number of inputs and outputs in building block $\mathcal{B}$ respectively |
| $\lambda$ | the population size |
| R | the number of algorithm runs |
| R($\mathcal{N}_f$) | the number of fully functional circuits evolved |
| $N_{gen}$ | the number of generations |
| $p_{mc}$ | the circuit mutation rate |
| $p_{mg}$ | the circuit layout mutation rate |
| $p_c$ | the crossover rate |

# Chapter 1

# Introduction

This thesis relates to the work carried-out in the School of Computing, at Napier University, between September 1997 and June 2000. The theme of this thesis is an extrinsic Evolvable Hardware applied to the combinational logic design.

This chapter states the objectives of the research.

## 1.1 Objectives

The goal of the research was to develop Evolvable Hardware approach capable of evolving *practical* digital logic circuits implementing the logic functions of large number of variables. In order to achieve this goal it was necessary to:

1. Investigate the evaluation process in the extrinsic gate-level Evolvable Hardware approach;

2. Develop a high-level self-adaptive EHW approach;

3. Design an extrinsic EHW approach, that is able to evolve circuits of large number of inputs and outputs.

## 1.2 The outline of the dissertation

### Chapter 2.

This is the introduction to the thesis. The need for an extrinsic evolvable hardware approach to design cost-optimised combinational digital circuits of large number of variables is discussed. In this chapter we will consider the main concept of Evolvable Hardware and discuss some work that has been done in this discipline. In spite of being a new research area, Evolvable hardware already incorporates a large set of applications that can be classified according to five important properties summarised in Evolvable Hardware of the taxonomy properties classification. We attempt to cover as many as possible aspects of research done in the field of evolvable hardware and especially in the field of extrinsic Evolvable Hardware. This includes research carried out on how the different genetic algorithm parameters, chromosome representation and Evolvable Hardware parameters influence the performance of algorithm. The incremental evolution that has been introduced to overcome the problem of "stalling" effect of the evolution for a large number of generations is discussed as well. The outline of this dissertation and the motivation for this research are also presented in this chapter.

### Chapter 3.

In this chapter we consider some aspects of evaluation process that include fitness function, optimisation criteria and probability analyses of the evolution processes. In order to improve the quality of evolved circuits we propose to use a *dynamic fitness* technique. The circuits are evolved in two phases. Initially the genome fitness in a given chromosome is evaluated by the percentage of output bits that are correct.

Once fully functional circuits have been evolved, the quality of evolved circuit is taken into the fitness function. This allows us to evolve circuits with 100% functionality *and* minimise the circuit by some of the criteria given in advance. We showed that the algorithm performs better when the dynamic fitness function strategy is used. The quality of evolved circuit can be defined by a number of criteria. We propose to evaluate the evolved circuit in terms of the number of basic blocks used in the implementation technologies. Thus, the FPGA-based circuit can be estimated by the number of primitive active logic cells or by the number of basic building block used in the FPGA design. The MOS-based circuit can be evaluated in terms of the number of transistors. Other optimisation criteria such as the circuit delay, circuit area, connectivity restrictions can be applied, but are not considered in this section. Therefore, using these criteria we are able to evolve efficient FPGA- as well as MOS-based circuits. The proposed fitness function technique has been compared with such multi-objective function approaches as MIN-MAX formulation, method of objective weighting and method of distance function. Also, we investigate how the circuit evolution is carried out. This allows us to define first what type of genes influences the most on algorithm performance and second, effectiveness of gate allocation in circuit design.

## Chapter 4.

In this chapter we have discussed one of the possible ways to improve the quality of evolved circuits. The choice of suitable circuit geometry is a very complicated task and is intimately linked with the complexity of the function implemented. So, in order to avoid this we have investigated the possibility of evolving the circuit layout at the same time as trying to evolve the fully functional circuits. The circuit geometry dictates the length of the chromosome, thus we worked with chromosomes of variable length. In this scheme, mutation was carried out in two ways. First, we can mutate genes associated with a circuit in a fixed geometry, and secondly, we can by mutation choose the circuit geometry. The main purpose of circuit layout evolution was to try to evolve the best circuit layout together with circuit functionality. However during the algorithm execution we found the interesting result that actually using a flexible circuit geometry allows us to reduce the number of active gates in circuit. This was unexpected. We have defined several strategies for the GA. We have investigated cases where we use homogeneous, heterogeneous or partially heterogeneous (heterogeneous only at the initialisation stage of algorithm) circuit layouts during algorithm execution and determined the algorithm performance as a function of both fitness measures.

## Chapter 5.

In this chapter we propose the function-level extrinsic Evolvable Hardware approach. This approach allows us to evolve circuits using primitive logic cells as well as multi-input multi-output logic functions specified in advance or generated during evolution. The connectivity of complex building blocks used in evolution is specified in advance and the functionality of these blocks is outlined during evolution. The behaviour of complex building blocks can be defined by one- or multi-output logic functions. The proposed chromosome representation allows us to reduce the size of chromosome significantly, that is essential in the complex problem solving area. The experimental results show that the proposed function-level extrinsic Evolvable Hardware evolves the fully functional circuits easier than the similar gate-level extrinsic Evolvable Hardware.

## Chapter 6.

Evolvable Hardware has been proposed as a new technique to design complex systems. However, complex systems turn out to be very difficult to evolve. The problem is that a general strategy is too difficult for the evolution system to discover directly. This chapter proposes a new approach that performs incremental evolution in two directions: from complex system to sub-systems with sufficient complexity and from sub-systems to a complex system. In this approach, first, the complex problem is gradually decomposed into some sub-tasks during evolution, and, then, the problem is evolved incrementally, by starting with simpler behaviour and gradually making the task more challenging and general. The system discovers the evaluation tasks, their sequence as well as dimensions of circuit layout automatically. Since the complexity of the evaluation tasks is unknown, the self-adaptive Evolvable Hardware approach has to be used in order to define the initial parameters of a system automatically. Hence, the method combining the circuit layout evolution together with circuit functionality described in Chapter 4 is applied to evolve sub-tasks. Additional to this method the function-level chromosome representation is used (see Chapter 5) in order to improve the algorithm performance. Thus, the defined sub-tasks are evolved using the self-adaptive function-level extrinsic Evolvable Hardware approach. The evaluation of each sub-tasks is performed using a dynamic fitness function introduced in Chapter 3. The method is tested in the digital circuit task, and compared with direct evolution. The bidirectional incremental approach evolves more effective and more general circuits and should also scale up to harder tasks.

Chapter 7.

In this chapter we apply the extrinsic Evolvable Hardware approach to multi-valued combinational logic design. We consider the possibility of evolving the multi-valued logic functions using both the gate- and the function- level extrinsic Evolvable Hardware described in Chapter 3 and Chapter 5 and applied previously to binary circuit design problem. We discussed the optimal parameters of both approaches. We show that the behaviour of the proposed extrinsic Evolvable Hardware is similar in both multi-valued and binary logic cases.

We end the thesis with some concluding remarks, which review the contribution of this work and include topics for further research.

## 1.3 Some remarks about terminology used in the dissertation

In this work we refer to some terminology that should be explained.

A *logic circuit is evolved* means that the evolutionary algorithm started with randomly generated non-functional circuit that, through evolution, finds a fully functional circuit. The circuit is evolved based on the test-and-assemble method.

A *functional set of logic gates*, $\mathbb{FS}$ is a set of logic gates, from which the circuit can be assembled. For example, considering the AND-OR PLA, the functional set of logic gates contains AND, OR and NOT logic gates. In evolvable hardware *any* set of logic gates can be chosen. Each logic gate in a functional set of logic gates is encoded with an integer. Thus, $\mathbb{FS} : \{5, 7, 8\}$ means that the logic gates, encoded by integers 5, 7 and 8, are used to evolve the logic circuit.

A *fully functional circuit* is a circuit that correctly implements the desired truth table. In other words the functionality of this circuit is 100%.

In this dissertation all *minimisation techniques* are performed using evolutionary algorithms, unless otherwise cited in the text.

The *uncommitted* or *redundant* logic gate is a logic gate that is in chromosome genotype but is not actually used in the circuit structure. The *active* logic gate is a logic gate that is not redundant.

The AND/OR/NOT and the AND/EXOR/NOT approach (commonly known as Reed-Muller) are used. We will refer to logic gate implementing a primitive logic operation from any of these algebras as *primitive logic gate*. So, from the algebraic point of view, the circuit can be evaluated in terms of the number of primitive logic gates. The cost of the circuit can be defined by the number of primitive active logic gates.

# Chapter 2

# Evolutionary design of electronic circuits

This chapter provides an overview of the background to Evolvable Hardware and the need to investigate the extrinsic Evolvable Hardware (EHW) is identified. In spite of being a new research area, Evolvable hardware already incorporates a large set of applications that can be classified according to five important properties summarised in EHW taxonomy classification and first proposed in 1996 [12]. Special attention will be paid to the progress made in the area of extrinsic Evolvable Hardware (EHW) applied to digital circuit design. The needs of the EHW realm are discussed and the research area of this work is drawn.

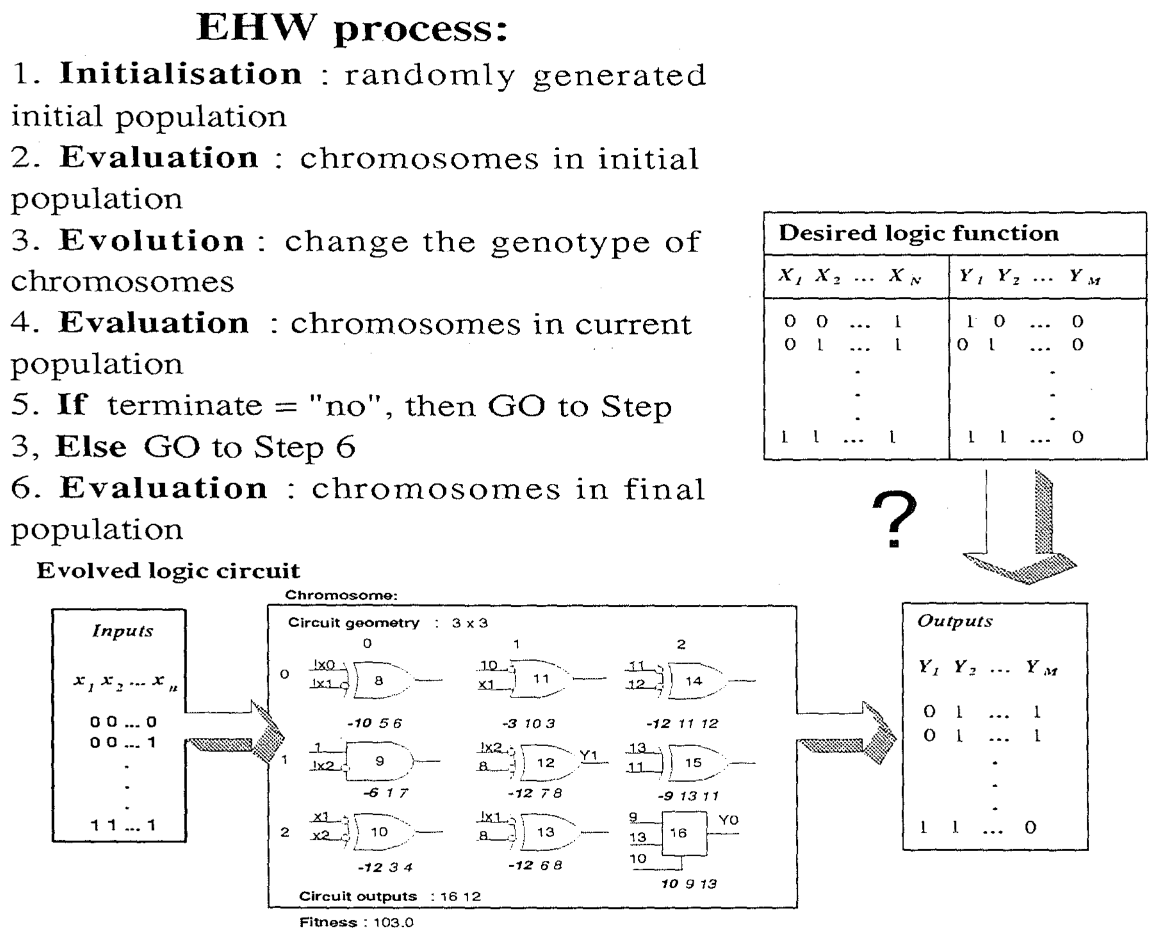
## 2.1 Circuit design problem and evolutionary algorithms

Evolutionary algorithms are employed in different application problems. One of them is circuit design problem, where a traditional circuit is implemented once the representation of the function is optimised.

There are two main approaches for the synthesis of combinational logic circuits using evolutionary algorithms.

1. The first approach optimises the formal representation of the function and designs the circuit based on the optimal function representation. In this case a functionally complete basis is chosen and the genetic algorithm is applied to optimise the form of the function representation. For example, variable ordering of Binary Decision Diagrams for multi-level functions using evolutionary algorithm has been discussed in [13], [14], [15]. Using the obtained optimised representation the circuit structure is synthesised. It is clear that the circuit design is obtained by the application of algebraic rules associated with the relevant algebra. Further, the synthesised circuit can be tested using evolutionary algorithms. The verification of circuits by means of evolutionary algorithms has been reported in [16].

2. The second approach namely *Evolvable Hardware* (EHW) begins from randomly connected and randomly chosen gates and gradually evolves the target functionality. So, there is no specific knowledge about circuits generated in the initial population. The particular set of gates used is fixed in advance, but whether or not any particular gate is used, or, how many times a gate is used, is entirely free. The advantage of this approach is that it allows us to synthesize the circuit using *any* set of logic gates. Consequently, it permits the synthesis of compact and unusual circuit structures. In this way we can abandon the restrictions associated with conventional design [9]. The basic idea of this approach is demonstrated in Fig. 2.1.



**EHW process:**

1. **Initialisation** : randomly generated initial population
2. **Evaluation** : chromosomes in initial population
3. **Evolution** : change the genotype of chromosomes
4. **Evaluation** : chromosomes in current population
5. **If** terminate = "no", then GO to Step 3, **Else** GO to Step 6
6. **Evaluation** : chromosomes in final population

| **Desired logic function** | |
|---|---|
| $X_1$ $X_2$ ... $X_N$ | $Y_1$ $Y_2$ ... $Y_M$ |
| 0 0 ... 1 | 1 0 ... 0 |
| 0 1 ... 1 | 0 1 ... 0 |
| . | . |
| . | . |
| . | . |
| 1 1 ... 1 | 1 1 ... 0 |

Figure 2.1: Circuit design problem in EHW. In Evolvable Hardware approach, the circuits in initial population are generated *randomly* and, therefore, *do not implement* correctly the desired logic function. In EHW, an evolutionary algorithm *designs* a circuit that correctly implements a given logic function and *optimises* to obtain a fully functional circuit. In other words, evolutionary algorithms *evolves* a logic circuit.

## 2.2 Main concept of Evolvable Hardware

*Evolvable Hardware* (EHW) is technique to synthesise and optimise electronic circuits using evolutionary algorithms.

In the context of electronic synthesis the configuration of evolved circuits as well as connecting elements inside circuits are represented by chromosomes. A circuit can be implemented in hardware or simulated in software. An evolutionary process is employed to the population of circuits in order to synthesise a target circuit. An initial population is generated randomly. Genetic operators are applied to the chromosome and obtained new circuits are compared with the target logic function. Connecting elements can represent any electronic element or device. Connecting elements are linked with each other according to the connectivity restrictions applied to the target implementation technology. One of the main tasks of the evaluation process is to define how the functionality of the evolved circuit approaches the target function. Once the fully functional circuit is evolved, a number of optimisation criteria can be applied in order to obtain the circuit with desired efficiency. The process in usually ended after a given number of generations or when the closeness to the target response has been reached. If the connecting element is represented by a primitive logic function, a *gate-level EHW approach* is applied. In the *function-level EHW approach*, high level hardware functions such as adders, multipliers, etc. rather than simple logic functions are used as primitive functions in evolution [17], [18]. Therefore, the connecting element, or so called *building block*, implements the one-output or multi-output logic function. Fig. 2.2 depicts the basic idea of EHW in electronics.

Taken as a design methodology, evolvable hardware offers a major advantage over

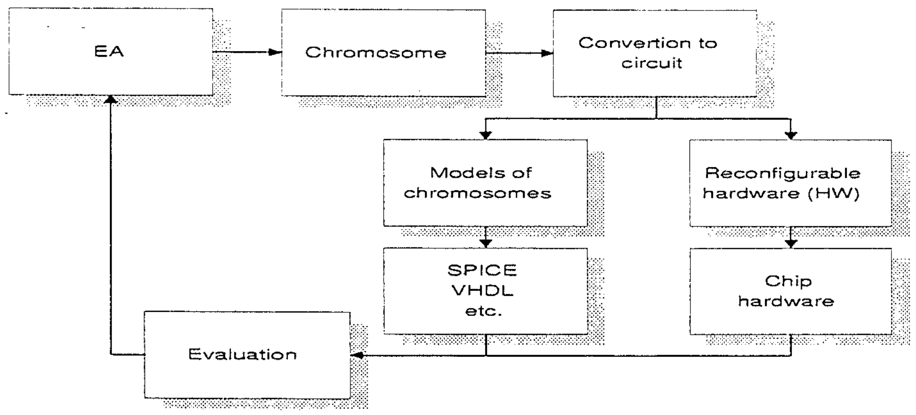


Figure 2.2: EHW in electronics.

classical methods; the designer's job is reduced to that of specifying the circuit requirements and the basic elements, whereupon evolution "takes over" to "design" the circuit [19]. The Evolvable Hardware exploits the search space that has been unreachable before by classical design methodologies. EHW conquers new application areas, and currently the future of EHW is considered as self-adaptive, self-reconfigurable hardware, capable to adjust to new problem without the designer's help.

## 2.3 Taxonomy of Evolvable Hardware

In this section the main directions of EHW and their development will be considered. The classification of current EHW can be roughly derived in accordance with the following five key properties specified as a taxonomy of evolvable hardware in [12] (see Fig. 2.3):

1. Evaluation process;

2. Evolutionary process;

3. Evolutionary programming approach;

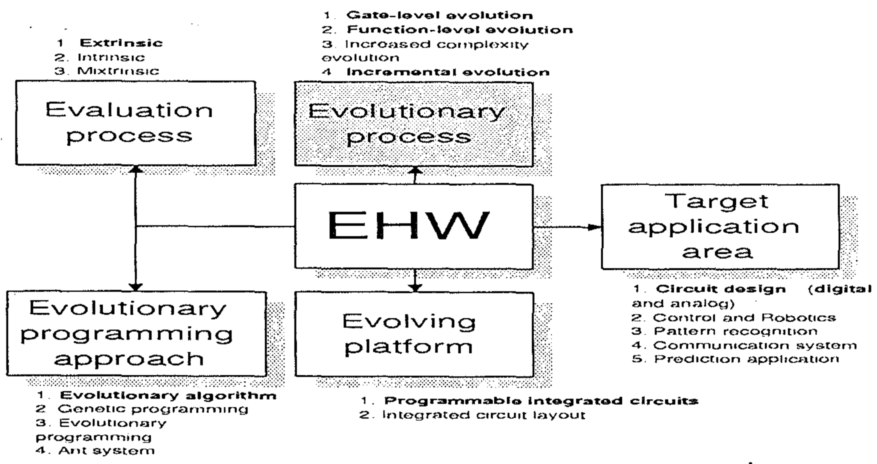

1 Extrinsic
2. Intrinsic
3. Mixtrinsic
1. Gate-level evolution
2. Function-level evolution
3. Increased complexity evolution
4 Incremental evolution
Evaluation process
Evolutionary process
EHW
Target application area
1. Circuit design (digital and analog)
2. Control and Robotics
3. Pattern recognition
4. Communication system
5. Prediction application
Evolutionary programming approach
Evolving platform
1. Evolutionary algorithm
2 Genetic programming
3. Evolutionary programming
4. Ant system
1. Programmable integrated circuits
2. Integrated circuit layout

Figure 2.3: Taxonomy of evolvable hardware.

4. Target application area;

5. Evolving platform.

### 2.3.1 Evaluation process

In EHW, the evaluation of chromosomes can be performed on software or hardware. Depending on how the evaluation process is implemented, three types of evolutionary processes can be defined:

1. Extrinsic evolution, [20], [21];

2. Intrinsic evolution, [22], [23], [21];

3. Mixtrinsic evolution, [24];

**Extrinsic EHW.** In *extrinsic* evolution, the candidate solutions are evaluated as software (SW) models and evaluations are done using a simulator. Extrinsic EHW is schematically illustrated in Fig 2.4. The population is homogeneous and consists of

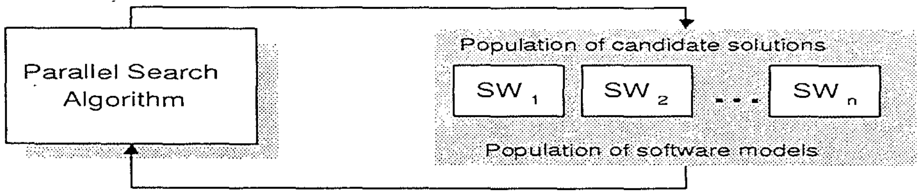

Parallel Search Algorithm
Population of candidate solutions
SW 1
SW 2
SW n
Population of software models

Figure 2.4: Extrinsic EHW: evaluations of software solutions.

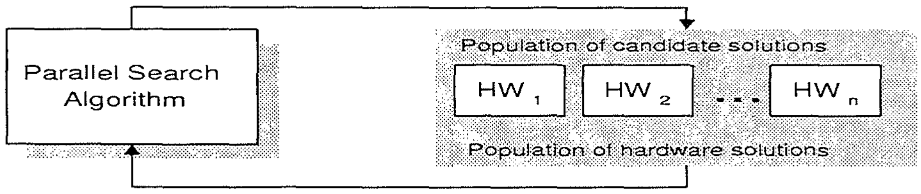

Parallel Search Algorithm
Population of candidate solutions
HW 1
HW 2
HW n
Population of hardware solutions

Figure 2.5: Intrinsic EHW: evaluations of hardware solutions.

SW models (e.g. SPICE netlists, VHDL, etc.) that describe an electronic circuit to a certain degree of accurancy. The extrinsic evolution is widely applied to the digital circuit design problems since it operates with logic gates and can always produce a correct solution that is replicable to HW [25], [26], [27], [1]. Extrinsic EHW has been applied to evolve such analogue circuit design problems as

1. low pass filter (Miller et al. [28], [29], [30]);
2. passive filter (Koza et al. [31]);
3. high gain amplifier (Koza et al. [32]);
4. 60 decibel amplifier (Bennett III et al. [33]);
5. operational amplifier (Bennett et al. [34]).

These problems can be solved only with some approximation in extrinsic EHW.

**Intrinsic EHW.** In intrinsic EHW, the candidate solution are implemented on physical HW configurations on programmable devices/architectures, which are evaluated using some test/evaluation equipment (see Fig. 2.5). There are major advantages in using this approach: the speed of the evolutionary process and the fact that no simulation models are needed to evaluate the circuit [22], [12], [35]. In intrinsic EHW individual candidates influence each other because each candidate performs on the same chip as its predecessors. Intrinsic EHW has been applied in order to evolve

1. tone discriminator (Thompson et al. [36], [37]);
2. switched capacitor circuits (Zebulum et al. [21]);
3. oscillator circuits (Huelsbergen et al. [38]);
4. electronic stethoscope circuit (Lohn et al. [39]);
5. electric filters (Zebulum et al. [40], [41]);
6. intermediate frequency filters (Murakawa et al. [42]);
7. Butterworth low-pass filter (Lohn et al. [39]);
8. non-liner digital filter (Perkins et al. [43]);
9. band-pass filter (Zebulum et al. [44]).

**Mixtrinsic EHW.** Mixtrinsic evolution relates to evolving on mixed (heterogeneous) populations, composed partly of models and partly of real HW. This constrains the evolution to a solution that jointly simulates well in SW and performs well in HW. In other words, it produces a solution that exploits only HW characteristics included in the SW model for producing the desired behaviour (Fig. 2.6). Mixtrinsic EHW

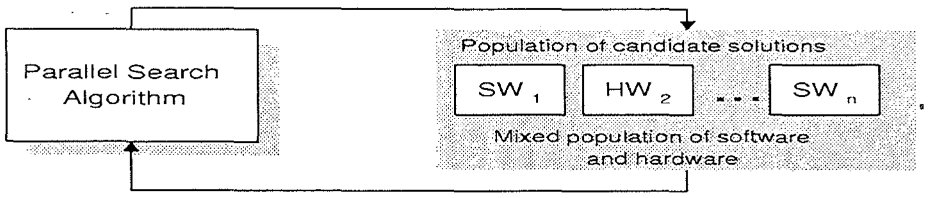

Parallel Search Algorithm
Population of candidate solutions
SW 1
HW 2
SW n
Mixed population of software and hardware

Figure 2.6: Mixtrinsic EHW: evaluations of mixed populations comprised of both hardware and software solutions.

has been proposed to solve a portability problem, since for several reasons (including mismatches between models and physical HW, limitations of the simulator and testing system, etc.) circuits evolved in SW may not perform the same way when implemented in HW, and vice-versa.

Two types of mixtrinsic EHW have been proposed [24]:

1. Complementary;

2. Combined.

In *complementary* mixtrinsic EHW, candidate solutions are evaluated after being alternatively reassigned to either a HW or a SW platform (subject to random or deterministic choice).

In *combined* mixtrinsic EHW, each individual is evaluated both in HW and SW, and a combined fitness function is calculated. In the simplest case, this can be a simple average of the two components or may involve adjustable weights, etc.

Mixtrinsic evolution has been applied by Stoica et al. [24] to evolving AND gate using one field programmable transistor array (FPTA).

In all EHW methods mentioned above genetic algorithm is implemented in software. Recently, *complete hardware evolution* has been introduced, where the evolutionary process itself is implemented in hardware [45], [43]. Instead of having all

(Extrinsic) or part (Intrinsic) of the evolutionary process in the host processor, a hardware implementation of the evolutionary process is used to drive evolution. So far, two works has been reported in this direction:

- Tufte and Haddow [45] proposed a GA pipeline where the evolutionary process together with the evolving design are implemented on the same chip. They propose two approaches to overcome the problem of reconfiguring the chip once a new individual is generated: (1) to set-off an area on the FPGA chip to be used as re-configurable memory that may be reconfigured internally within the chip or (2) to use partially reconfigurable FPGA technology which allows selection of parts of the chip for configuration. The method has been tested on evolving an one-control multiplexer. One of the disadvantages of this method is that this prototype limits the possibility of evolving larger more complex designs, where complexity may be measured in terms of the complexity of both the genotype required to represent the design and the fitness function required to evaluate the design. Limited chip size also affects the population size which limits the amount of genetic information available to help the evolutionary process towards an acceptable fitness level. Limited genetic information in the form of smaller populations may cause the evolutionary process to evolve towards a non-optimal solution.

- Perkins et al. [43] developed a self-contained FPGA-based implementation of a spatially-structured evolutionary algorithm that provides a significant speedup over conventional serial processing in three ways: (a) efficient hardware-pipelined fitness evaluation of individuals, (b) evaluation of an entire population of individuals in parallel and (c) elimination of slow off-chip communication. They find a speedup factor of over 1000 compared to a C implementation of the same system.

### 2.3.2 Evolutionary process

Evolution in EHW can be performed at different levels:

1. Gate-level EHW;
2. Function-level EHW;
3. Increased complexity evolution;
4. Incremental evolution.

**Gate-level EHW.** The hardware evolution employing the primitive gates such as AND and OR gates is called *gate-level evolution* [18]. One of the most common problem in gate-level EHW is that this approach is only suitable for small circuits. So, the gate-level evolution is not powerful for the use in industrial applications. Nevertheless, gate-level EHW has been applied for a number of real applications. A myoelectric artificial hand, which is operated by muscular control signals has been designed using a gate-level EHW chip [46]. In this particular application the EHW performance is slightly better than with neural networks. At the same time the learning time in this application is considerable reduced. An evolutionary robot navigation system developed using a gate-level EHW has been introduced in [47]. Applications with gate-level EHW perform better than with neural networks [46]. Experimental results show that gate-level EHW operates very well in solving relatively easy application tasks.

**Function-level EHW.** The function-level evolution has been introduced to be employed in more practical applications than the gate-level EHW. In the *function level evolution*, high level hardware functions, such as addition, subtraction, sine, etc., rather than simple logic functions are used as primitive building blocks in the evolution [48], [49], [18]. Much more powerful circuits can be evolved using the function evolution [50], [49]. As an alternative to the function-level EHW there has been work in Genetic Programming for evolving the functions [51]. The method is called Automatically Defined Functions (ADF) and is used in software evolution. Function level of evolution have been applied to real applications. Simulations of data compression using function level evolution indicates performances comparable to other compression methods like JPEG compression [52]. The scheme is designed for implementation in a custom ASIC device. A function based FPGA has been proposed for applications like ATM cell scheduling [50] and adaptive equalizer in digital mobile communication [49]. Function-level EHW shows better performance than similar gate-level EHW [52], [49], [50]. In all applications mentioned above, function-level EHW is implemented using intrinsic evolution. The most complex connecting element used in extrinsic EHW is an one-control multiplexer [53], [9], [25]. No extrinsic EHW has been designed that uses multi-input multi-output connecting elements. One of possibilities of evolving a digital circuit using function-level extrinsic EHW will be discussed in Chapter 5. An FPGA model consisting of 100 programmable floating processing units (PFUs) has been applied to such well-known problems as the two interwined spirals, the Iris data set, 2-D image rotation, synthesis of a 4-state automation [48], [18], [17].

**Increased complexity evolution.** The idea of *increased complexity evolution* is to evolve a system gradually as a kind of divide-and-conquer method [54], [55]. Evolution is first applied individually to a large number of simple cells. The evolved functions are the basic blocks used in further evolution or assembly of a larger and more complex system. This may continue until a final system is at a sufficient level of complexity. The main advantage of this method is that evolution is not performed in one operation on the complete evolvable hardware unit, but rather in a bottom-up way. The increased complexity evolution has been applied to the problem of recognizing characters of 5x6 pixels size, where each pixel can be 0 or 1 [54]. As a result of increased complexity evolution, the number of generations can be drastically reduced by evolving sub-systems instead of a complete system.

The divide-and Conquer method can be successfully applied to such application problems where the interaction within complex systems is understood. Torresen [54], [55] proposed to divide a system into functionally independent sub-systems, that can be evolved independently. It has been proposed to divide a system without defining the optimal partition points. This lack of a method can lead to increased complexity of evolved solution. In order to avoid it, well-known decomposition methods can be applied together with divide-and-conquer method. This approach will be discussed in detail later in Chapter 6.

As has been mentioned above, only the relatively easy problems can be handled successfully using the divide-and-conquer method [56]. This method is not suitable to problems, where interactions within a complex system are so many, or so little understood, that we cannot understand how to divide it into smaller comprehensible pieces. It is for these kinds of problems that artificial evolution is put forward as a means to develop complex systems that work even when they are too complex to comprehend.

**Incremental evolution.** Incremental evolution is used for these purposes. It has been applied successfully to a number of application problems such as control of robotic system [56], [57], [58] and task of prey capture [59] that are stochastic, dynamic complex tasks. The incremental approach evolves more effective and more general behaviours, and should also scale up to harder tasks [59]. No attempts to apply incremental evolution to extrinsic EHW have been made before. This issue will be raised in Chapter 6. An incremental evolution applied to EHW differs from similar methods applied in neuro-evolution. This method incorporates the interaction knowledge of complex systems and knowledge of successful evolution. The evolution is first applied to the randomly generated population of chromosomes. Then at some point the evolved material is analysed on the genetical significance. Genetically important parts of the chromosome are further evolved in order to complete sub-tasks. Once the fully functional solution is obtained the system is gradually optimised through evolution. The incremental evolution is applied to evolve digital circuits extrinsically.

### 2.3.3 Evolutionary programming approach

In order to exploit the search space and design circuits, EHW uses artificial evolution. There are basically four approaches to implementing the artificial evolution:

1. Evolutionary algorithms;
2. Genetic programming;
3. Evolutionary programming;

4. Ant System.

Evolutionary algorithms have the following basic features:

- Binary encoding to form chromosomes;
- Random initialisation of a population;
- Recombination operators (crossover, mutation);
- Generational replacement of population with elitism.

The evolutionary algorithms include

- genetic algorithms;
- evolutionary strategy.

The evolutionary strategy differs from genetic algorithms mainly in the way recombination is implemented. Instead of using both crossover and mutation operators, the evolutionary strategy utilises the mutation operator only.

In the genetic programming method chromosomes are represented as trees with ordered branches, in which the internal nodes are functions and the leaves are terminals (variables and operands).

In evolutionary programming the chromosomes are encoded using integer or real numbers. Mainly, evolutionary programming adopts the same operators as in evolutionary algorithms.

The ant system is a multi-agent system, where low level interactions between single agents (i.e. artificial ants) result in a complex behaviour of the whole ant colony, The idea was inspired by colonies of real ants, which deposit a chemical substance on the ground called pheromone. This substance influences the behaviour of the ants: they will tend to take those paths where there is a larger amount of pheromone [60], [61], [62].

Combination of these four basic methods produces a large variety of other alternative evolutionary programming methods.

## 2.3.4 Target application area

Recently evolvable hardware has been applied to a wide range of application areas, including:

- Digital and analogue circuit design;
- Control and robotics;
- Communication systems;
- Pattern recognition;
- Prediction application.

#### Circuit design

Some research has been done in the area of analogue and digital circuit design.

**Analog circuit design.** EHW has been applied to evolve different parts of communication system that includes

- adaptive equalizer (Murakawa et al. [49]);
- low-pass filter (Miller [28]);
- intermediate frequency filter (Murakawa et al. [42]);

- Butterworth low-pass filter (Lohn and Colombano [39]);
- linear filter (Lohn et al. [63]);
- non-linear filter (Lohn et al. [63]);
- electronic stethoscope circuit (Lohn and Colombano [39]);
- microwave circuit (Kasai et al. [64]).

Let us consider in more detail some of applications to EHW.

- Layzell [65] evolved a NOT gate using specially designed motherboard at transistor level.
- Huelsbergen et al. [38] used an evolutionary search to find automatically electronic circuits that toggle an output line at, or close to, a given target frequency. They found empirically that oscillating circuits can be evolved that closely approximate some of the supplied target frequencies.
- Lohn and Colombano [39] evolve a low pass analogue filter suitable for use in an electronic stethoscope using a linear representation, a simple unfolding technique and a parallel genetic algorithm. Circuit size, circuit topology and device values are evolved in their system. This method helps to minimise the computer time required to evolve circuits by keeping the decoding and repairing processes shorter. At the same time this technique is topology-limited.
- Zebulum et al. [21] consider both intrinsic and extrinsic evolution of amplifiers and oscillators. They use different techniques for analogue circuit representation. They show that in analogue circuit design some precautions can result in

imposing constraints to the evolutionary algorithm due to physical limitations of semiconductor devices.

**Digital circuit design.** Some work has been done in the area of evolving digital circuits. Most of them use the extrinsic EHW approach, that will be discussed later in detail. At the same time some applications have been implemented using intrinsic EHW.

- Manovit et al. [66] synthesized different types of synchronous sequential logic circuits such as a counter, serial adder, frequency divider, modulo-5 detector and parity checker using registered PAL.
- Levi and Guccione [67] evolved at the gate and flip-flop levels an 8-bit counter and several digital frequency dividers. They show that the evolved digital circuits are stable. A genetic FPGA uses Xilinx's JBits interface to control the generation of bitstream configuration data and XHWIF portable hardware interface to communicate with a variety of commercially available FPGA-based hardware.
- Damiani et al. [68] evolve a digital circuit which computes a simple hash function mapping a 16-bit address space into an 8-bit one.
- Hollingworth et al. [11] evolve two-bit adder using Virtex devices through internet reconfigurable logic.

### Control and Robotics

EHW can change its own hardware structure to adapt to the environment whenever environmental changes occur through genetic learning. This feature of EHW can be used in some control applications. The process of genetic learning consists of seeking the best hardware configuration given a particular environmental condition. Some overview of evolutionary techniques in physical robotics can be found in [69].

The following systems have been developed using EHW:

1. control system [70], [71], [72];
2. evolutionary robot navigation system [47];
3. learning system for autonomous robot [73], [74].

One of the basic elements in a robotic control system is the controller. Evolving controllers for autonomous mobile robots becomes one of the attractive application areas for EHW.

- A. Thompson [75] evolved a real hardware robot controller for wall-avoidance behaviour. For the hardware evolution, architecture bits (also called "configuration memory") of the EHW controller, which was implemented in FPGAs, were used as genotypes. In this work, the whole function and behaviour of the EHW controller have been determined in advance.

- In [71] the genetic evolution in the environment was employed to determine the connections among the elements, thus types of reflexes realised by logic circuits and the ways of their combinations being obtained automatically by the interactions between the environment and the system itself.

- T. Higuchi et al [23] developed a V-shape ditch tracer with fault-tolerant circuit that is used as a prototypical welding robot. EHW works as the backup of the control logic circuit for the tracing, although the EHW is not given any information about circuit. As mentioned in [23] the learned result can be directly translated into a new hardware system, making it suitable for real time applications.

- Ebner [76] demonstrated the possibility of evolving a hierarchical control architecture using genetic programming on a large physical mobile robot.
- The evolution of a connectionist structure where each node has an associated program, evolved using genetic programming has been proposed by Silva in [72]. It has been reported that this approach requires less effort to find a solution in comparison with other known genetic programming based approaches.
- Both topology and tuning of controller with a free variable can be evolved using extrinsic EHW implemented by means of genetic programming [77].
- An evolved robotic controller for a four-legged real robot enabling it to walk dynamically has been proposed in [78]. The evolution has been performed on-line by a linear machine code GP system. The robot has eight degrees of freedom. It has been shown that the evolving system is robust against mechanical failures.
- Harnby et al. [79] evolve a ball-chasing behavior that successfully transfers to an actual AIBO (a registered trademark of Sony Corporation). Their approach differs from others in that they model an intermediate software layer which passes processed sensor data to the controller and receives high-level control commands, instead of simulated ray sensor values. They reported that in this interpretation a simulator runs over 11000 times faster than real time.

**Pattern recognition**

- T. Higuchi et al [23] designed the pattern recognition system that has been tested on recognition of three patterns consisting 64 pixels (black and white, 8x8). The results show that EHW works as a hard-wired pattern recognizer with such robustness as neural networks.

- Hollingworth et al. [80] proposed highly parallel image processing tool to detect edges within a wide range of conventional grey-scale.

- Dumoulin et al. [81] introduced a method of inventing linear edge enhancement operators using evolution and reconfigurable hardware in order to design a totally automated object recognition system. A technique proposed by them builds an edge enhancement operator using evolutionary methods and implements and tests each generation using the Xilinx 6200 family FPGA. Images of 640x640 pixels have been processed.

- Yasunaga et al. [82] proposed a logic circuit design methodology for pattern recognition chips using genetic algorithms. In the proposed method, pattern data are transformed into the truth tables and the truth tables are generalized to adapt the unknown pattern data. The generalized, or evolved truth tables are then synthesized to logic circuits. In this method no floating point numerical circuits are required and the intrinsic parallelism in the data is embedded into the circuits. Consequently, high speed recognition systems can be realized with acceptable small circuit sizes. The face image and sonar spectrum of 8x8 pixels are chosen as examples.

### Data compression

Tanaka et al. [83] describe a data compression system using Evolvable Hardware for digital colour electrophotographic printers. They show that EHW can change the compression method according to the characteristics of the image. The proposed EHw-based compression system can compress approximately twice as much data as JBIG, the current international standard.

Stoica et al. [84] demonstrated the evolution of compression algorithms with results better than the best-known compression algorithms. Their system was used to automatically generate a hardware-based image compression algorithm specially adapted for the class of images captured by the spacecraft. A nonlinear model used by the compression algorithm is evolved using genetic programming technique, which can be then compiled to an FPGA configuration, and finally downloaded (up-link to the spacecraft) to the real FPGA.

### Prediction Application

In a prediction application, EHW gets information from the environment in which it is embedded and tries to predict the next state of the environment [85]. EHW has been used in such prediction applications as

- scheduling real-time traffic in Asynchronous Transfer Mode (ATM) networks (Liu et al. [50]);
- data compression (Salami et al. [86]).

## 2.3.5 Evolving platform

Different evolving platforms can be used in EHW:

1. Programmable integrated circuits;

2. Integrated circuit layout.

**Programmable integrated circuits.** Programmable integrated circuits can be divided into three categories [19]: (1) Memories; (2) Microprocessors; (3) Logic circuits. EHW focuses mainly on the Programmable logic circuits. Logic circuits can be divided into three sub-categories [19]: PLD (Programmable Logic Device); CPLD (Complex Programmable Logic Device); FPGA (Field Programmable Gate Array).

A PLD circuit consists of an array of AND gates, which generates product terms from the system's inputs, and an array of OR gates, which generates the output of the system. According to their degree of programmability, PLDs can be classified into PROM (Programmable Read Only Memory), PAL (Programmable Array Logic) and PLA (Programmable Logic Array) [19].

A CPLD can be seen as a combination of programmable cells consisting usually of multiplexers or memories and an interconnection network that selects the inputs of the programmable cells.

FPGAs are the most popular reconfigurable devices in EHW [19], [12]. They consist of an array of logic cells and I/O cells. Each logic cell consists of universal function [19] (multiplexer, demultiplexer, memory), which can be programmed to realize a certain function. FPGAs are highly versatile devices that offer the designer a wide range of design choice.

Instead of using FPGAs, there is also the possibility to use a memory as the evolving platform. In this case, the memory contents will be genetically programmed [12], [75].

**Integrated circuit layout.** In this method, integrated circuit layers, such as metal, oxide and silicon, as well as complete devices, such as transistors or transmission gates, are handled by the evolutionary process to create a complete circuit layout [12]. VLSI design is an example of this approach. In this case, the intrinsic approach is the only viable scheme to evaluate evolved circuits.

Recently, a number of chips developed especially for EHW has been introduced in the past.

- Kajitani et al. [46] introduced an integrated EHW LSI chip that consists of GA hardware, reconfigurable hardware logic, a chromosome memory, a training data memory and a 16-bit CPU core. They showed that the EHW performs slightly better than a neural network and that the learning time is considerably reduced.

- Laysell [65] presented a test platform designed specifically to tackle the difficulties of using FPGAs. The motherboard has been designed to investigate many important issues arising in current EHW research, including analysis, fault tolerance, genotype encoding, portability, basic elements and evolved topologies.

- Murakawa et al. [42] proposed an analogue EHW chip for Intermediate Frequency Filters, which are widely used in cellular phones. There are two advantages of the proposed chip, namely, (1) improved yield rates and (2) smaller circuits, which can lead to cost reductions and efficient implementation.

- Hamilton et al. [87] developed a demonstration board to implement analogue and mixed-signal arrays. The rich mix of analogue and digital functionality provided by Palmo systems combined with the fact that they may accept random configuration bit streams them most attractive as platforms for evolvable hardware. They demonstrate the ability of proposed tool to evolve circuits using different technologies and standard electronic chips such as Motorola device, Zetex device.

- Langeheine et al. [88] suggested a hardware system devoted to perform evolution of analogue VLSI circuits. The system contains a CMOS chip providing an array of 16x16 transistors programmable in their channel dimensions and their connectivity. A genetic algorithm is executed on a PC connected to one or more programmable transistor arrays. Individuals are represented by a given configuration of this array. They demonstrate the feasibility of proposed method.

- Stoica et al. [89], [44] proposed a fine grained reconfigurable transistor array (FPTA) that is integrated on CMOS chip. The FPTA has advantageous features for hardware evolutionary experiments when compared to programmable circuits with a coarse level of granularity. They demonstrated the flexibility and versatility for the implementation of a variety of circuits in comparison with other models of re-configurable circuits.

A number of EHW design issues concerning robust electronics [37], [90], fault tolerance [91], [92], [93], fault recognition [94], self-repairing features of hardware [95], [96], dynamic fitness schedules [63] have been actively discussed in the past, but have not been covered in this overview.

## 2.4 An extrinsic EHW in digital circuit design

EHW has been applied to a wide range of complicated tasks. The development of extrinsic EHW approach is very important for the following reasons:

1. The portability problem in EHW can be solved using mixtrinsic EHW, where some of solutions are evaluated in software;

2. The EHW exploits the new search space, that is not limited by traditional logic algebra [9];

3. The circuit evolutionary process is very similar in both intrinsic and extrinsic EHWs.

Because of reasons mentioned above it is very important to understand the nature of evolving circuits using extrinsic EHW.

Most extrinsic EHW approaches have been designed to evolve both the functionality and connectivity of interconnected primitive logic gates such as AND, OR, NOT [97]. The extrinsic EHW approaches have been adopted to synthesize the circuits intended use on a programmable logic array (PLA) devices [97], Xilinx 6000 Field Programmable Gate Arrays (FPGAs) [25], Xilinx XC6216 FPGA [98], [99], Virtex FPGA [100], [101]. Note that the PLA is a device which has an input field of AND gates whose outputs are fed to an output field of OR gates to form sum-of-product logic solutions.

A variety of extrinsic EHW methods have been applied to synthesise digital circuits (see Table 2.1). In most cases, primitive functions or multiplexers have been considered as connecting elements.

Table 2.1: Summary of extrinsic EHW approaches for digital circuit design VGA is a variable-length chromosome GA; GP and EP are a Genetic and an Evolutionary Programming; CGP is a Cartesian GP; AS is an Ant System; $f(n, m, r)$ is an $n$-input $m$-output $r$-valued logic function; $f_{mux}(3, 1, 2)$ is a logic function describing the behaviour of multiplexer; $\mathcal{F}$ is a fitness function; $F_1$ defines a correctness of outputs of network evolved; $F_2$ is the minimal number of logic cells used; $F_3$ is a correctness of input combinations; $F_4$ is an error based fitness, $\mathcal{F}_1 + \mathcal{F}_2$ is a dynamic fitness function strategy with execution of criteria $F_1$ at first stage of evolutionary process and criteria $\overline{F_2}$ at the second stage; $p$ is the number of input combinations in the logic function.

| **Author** | **Building block** | $\mathcal{F}$ | **EA** | **Application** |
|---|---|---|---|---|
| Kitano H., 1996 [20] | $f(2,1,2)$ | $F_1$ | EP | 6-1 multiplexer ($f(6,1,2)$, $p = 64$); Multiple XOR |
| Zebulum R.S. et al, 1996 [12] | $f(2,1,2)$ | $F_1$ | GA | Digital string simulator; 4-1 multiplexer; 3x8 decoder; 1-bit full adder |
| Murakawa M. et al, 1996 [49] | $f(n,1,2)$ | $F_1$ | VGA | Adaptive equalization |
| Thompson A., 1996 [22] | $f(2,1,2)$ | $F_1$ | GA | Oscillator |
| Miller J. et al, 1997 [25] | $f(2,1,2)$ $f_{mux}(3,1,2)$ | $F_1$ | CGP | Arithmetic binary circuit design |
| Miller J., 1998 [102] | $f(2,1,2)$ $f_{mux}(3,1,2)$ | $F_1$ | GA | Real-valued function design |
| **Kalganova T. et al, 1998** [7] | $f(2,1,r)$ $f_{mux}(r+1,1,r)$ | $F_1$ | CGP | **3-valued half adder** |
| Hernandez-Aguirre A. et al, 1999 [103] | $f_{mux}(3,1,2)$ | $\mathcal{F}_1 + \mathcal{F}_2$ | GP | Logic function: $f(6,1,2)$, $p = 64$ |
| **Kalganova T. et al, 1999** [1] | $f(2,1,2)$ $f_{mux}(3,1,2)$ | $\mathcal{F}_1 + \mathcal{F}_2$ | CGP | 2-bit multiplier ($f(4,4,2)$, $p = 16$) |
| Coello C.A. et al, 1999 [26] | $f(2,1,2)$ | $\mathcal{F}_1 + \mathcal{F}_2$ | GA | 2-bit multiplier ($f(4,4,2)$, $p = 16$) |
| Masher J. et al, 1999 [98] | $f(2,1,2)$ | $F_1$, $F_3$ | GA | Sorting network design |
| Miller J., 1999 [104] | $f(2,1,2)$ $f(3,1,2)$ | $F_4$ | CGP | Low pass filter design |
| **Kalganova T., 2000** [4] | $f(2,1,r)$ $\mathbf{f(n,m,r)}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | CGP | 3-bit multiplier ($f(6,6,2)$, $p = 64$); 2-bit full adder ($f(5,3,2)$, $p = 32$); **3-valued 1.5 digit multiplier; 3-valued 1-digit full adder** |
| Coello C.A. et al, 2000 [105] | $f(2,1,2)$ | $F_1$, $F_2$ | AS | 2-bit multiplier ($f(4,4,2)$, $p = 16$) |
| **Kalganova T., 2000** [6] | $f(2,1,r)$ $\mathbf{f(n,m,r)}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | BIE | **sqn_d.pla** ($f(7,10,2)$, $p = 128$), **m1_d.pla** ($f(6,12,2)$, $p = 32$), **z5xp1_d.pla** ($f(7,3,2)$, $p = 84$) |

Despite the fact that EHW has been actively studied during the last decade, a lot of work still has to be done in the area of improving the quality of evolved circuits, understanding the nature of circuit evolutionary process, and discovering new approaches that allow us to apply EHW to more complex tasks. As it can be seen from Table 2.1, extrinsic EHW has been applied to relatively easy problems. For instance, evolving logic functions of small number of variables. Mostly, research in the area of extrinsic EHW is directed to exploit the possible application areas and test how variations of evolutionary programming approaches influence the algorithm performance. Standard GA, evolutionary strategy and ant systems have been proposed for use in extrinsic EHW.

At the earlier stage of EHW investigation, the main purpose of EHW was to evolve fully functional circuits starting with randomly generated populations. The requirements to EHW change with time and currently researches intend not only evolve fully functional solution but also optimise the obtained solution by a number of optimisation criteria. In respect of these requirements a dynamic fitness function has been propose by Kalganova et al. [1] and Hernandez-Aguirre et al. [103] at the same time.

Hernandez-Aguirre et al. [103] proposed to use a WIRE function in the functional set of logic gates. Maximisation of the number of wires in the circuit lead to the minimisation of the number of logic gates in the circuits. Unfortunately in this interpretation redundant logic gates with another functionality than WIRE are not taken into account, which leads to the unaccurate estimation of the quality of evolved circuits. The dynamic fitness function proposed by Kalganova in [1] avoids this disadvantage. The quality of evolved circuit is estimated in terms of the number

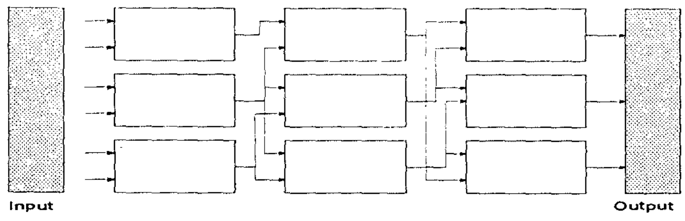


Figure 2.7: Matrix used to represent a circuit to be processed. Each gate gets its inputs from either of the gates in the previous column.

of logic gates actually used in circuit.

There are a lot of issues that can be investigated in extrinsic EHW. They include the circuit layout evolution, function-level evolution and incremental evolution. This dissertation addresses all of these problems.

The extrinsic EHW approaches proposed in the past are considered in details in the following sub-sections. Chromosome representations have a rectangular array structure in all extrinsic EHW approaches discussed before, since most of evolving platforms used in EHW have rectangular grid structure.

### 2.4.1 Louis's EHW approach

Louis [106] is one of earliest sources that report the use of GAs to design combinational logic circuits. The chromosome representation introduced by Louis [106] is currently relatively popular and is used by a number of researchers, mostly by research group supervised by Coello [26], [107], [108], [105], [27], [109].

**Chromosome representation.** The circuit is represented as a matrix in which each matrix element is a gate (there are 5 types of gates: AND, NOT, OR, EXOR and WIRE) that receives its 2 inputs from any gate at the previous column as shown in Fig. 2.7. More formally, we can say that any circuit can be represented as a

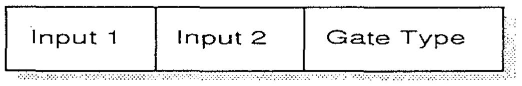


Figure 2.8: Encoding used for each of the matrix elements that represent a circuit.

bidimensional array of gates $S_{i,j}$, where $j$ indicates the *level* of a gate, so that those gates closer to the inputs have lower values of $j$. (Level values are incremented from left to right in Figure 2.7). For a fixed $j$, the index $i$ varies with respect to the gates that are "next" to each other in the circuit, but without being necessarily connected.

A chromosomic string encodes the matrix shown in Fig. 2.7 by using triplets in which the 2 first elements refer to each of the inputs used, and the third is the corresponding gate as shown in Fig. 2.8.

**Fitness function.** The goal of circuit design as stated in [26] is to produce a fully functional design (i.e., one that produces all the expected outputs for any combination of inputs according to the truth table given for the problem) which maximises the number of WIREs. This is happened because WIRE indicates a null operation, or in other words, the absence of gate, and it is used just to keep regularity in the representation used by the GA that otherwise would have to use variable-length stings [27]. Therefore, higher the number of WIREs in the circuit, lower the number of logic gates. So the dynamic fitness function has been introduced [26]. At the beginning of the search, only the validity of the circuit outputs is taken into account, and the GA is basically exploring the search space. Once a functional solution appears, then the fitness function is modified such that any valid designs produced are rewarded for each WIRE gate that they include, so that the GA tries to find the circuit with the minimum number of gates that performs the function required. It is at this stage that the GA is actually exploiting the search space, trying to optimise the solutions found (in terms of their number of gates) as much as possible.

**Application.** The complexity of solved problem depends on the circuit layout, the number of primitive logic gates in the functional set, the connectivity restrictions and the complexity of evolved circuit. Only relatively simple problems, such as implementation of a 3-variable logic function, a 2-bit multiplier, have been solved using this method [107], [26], [105].

### 2.4.2 Cartesian GP

In this dissertation, an extrinsic EHW (also called Cartesian GP [53]) proposed by J. Miller and applied to the digital logic circuits design will be extended [25]. The problem of interest consists of designing a digital circuit that performs a desired logic function (specified by a truth table). I refer to primitive logic gate if this gate implements any primitive one- or two-input logic function. The circuit evolution is performed using a rudimentary $(1 + \lambda)$ evolutionary strategy (ES) with uniform mutation [29] and an elite genetic algorithm (GA) [25]. In this case a population of random chromosomes is generated and the fittest chromosome is selected. The new population is then filled with mutated versions of this.

**Chromosome representation.** The following chromosome representation has been reported in [102]. The best way to explain the chromosome representation is to use a simple example. Fig. 2.9 shows the genotype and phenotype of a gate array with 3x3 circuit geometry ($N_{cols} \times N_{rows}$). The gate array implements a full adder. This function has 3-inputs and 2-outputs, therefore, $x_0, x_1, x_2$ represent the primary inputs of circuit and $y_0, y_1$ are the primary outputs of the circuit. The gate genotype which describes the behavioral features of logic cell is defined as follows: $< c_f \; i_0 \; i_1 \; i_2 >$. The function of each cell is expressed as the first gene $c_f$ associated with each cell

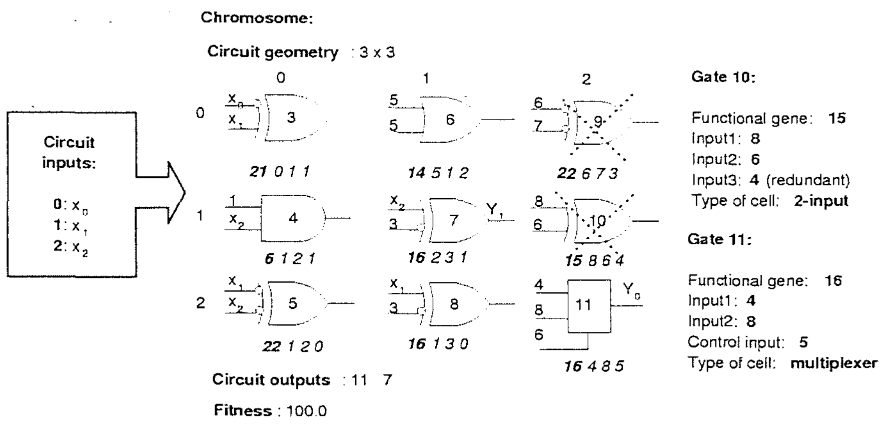


Figure 2.9: An example of the phenotype and corresponding genotype of a chromosome with 3x3 circuit layout used in cartesian GP.

(shown in bold typeface). Each cell is assumed to possess three input connections $i_0, i_1, i_2$. If the cell function does not require inputs then the corresponding genes are ignored. For example, the cell with output labeled 8 has input connections 1, 3 and 0. This means that the first input is connected to the primary input $x_1$, the second input is connected to the output of cell labeled 3 located in 0th column 0th row, and the third input is connected to the primary input $x_0$. The cell 8 is described by function of 2 variables. This means that third input gene is ignored. The primary outputs of the gate array are also expressed as connections. For example, $y_0$ is connected to the output of the cell labeled 11. The gate array is envisaged as being divided into vertical columns of cells. In order to define the feed-forward nature of circuit the representation is so constrained that columns of cells may only have their inputs connected to connection points on their left. The number of columns (including the primary inputs) to which the inputs of cell in question can be connected is assigned by connectivity parameter $N_{connect}$. The allowed cell functions can be chosen to be any

Table 2.2: Cell gate functionality.

| $c_f$ | $f$ | $c_f$ | $f$ | $c_f$ | $f$ | $c_f$ | $f$ | $c_f$ | $f$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | "0" | 5 | $\overline{i_1}$ | 10 | $i_0 \oplus i_1$ | 15 | $\overline{i_0} \vee \overline{i_1}$ | 20 | $i_0 \oplus (i_1 \wedge i_2)$ |
| 1 | "1" | 6 | $i_0 \wedge i_1$ | 11 | $i_0 \oplus \overline{i_1}$ | 16 | $(i_0 \wedge \overline{i_2}) \vee (i_1 \wedge i_2)$ | 21 | $\overline{i_0} \oplus i_1$ |
| 2 | $i_0$ | 7 | $i_0 \wedge \overline{i_1}$ | 12 | $i_0 \vee i_1$ | 17 | $(i_0 \wedge \overline{i_2}) \vee (\overline{i_1} \wedge i_2)$ | 22 | $i_0 \oplus \overline{i_1}$ |
| 3 | $i_1$ | 8 | $\overline{i_0} \wedge i_1$ | 13 | $i_0 \vee \overline{i_1}$ | 18 | $(\overline{i_0} \wedge \overline{i_2}) \vee (i_1 \wedge i_2)$ | 23 | $\overline{i_0 \wedge i_1}$ |
| 4 | $\overline{i_0}$ | 9 | $\overline{i_0} \wedge \overline{i_1}$ | 14 | $\overline{i_0} \vee i_1$ | 19 | $(\overline{i_0} \wedge \overline{i_2}) \vee (\overline{i_1} \wedge \overline{i_2})$ | 24 | $\overline{i_0 \vee i_1}$ |

subset of those shown in Table 2.2, where $\wedge, \vee, \oplus$ represent AND, OR and exclusive-OR (EXOR) operations respectively, $\overline{i_0}$ indicates NOT $i_0$. Functions 0-15, 21-22 are the basic binary functions. Functions 16-19 are all binary multiplexers with different inputs inverted. The multiplexer (MUX) implements a simple (IF-THEN) statement (i.e. IF $i_2 = 0$ THEN $i_0$ ELSE $i_1$).

For many circuits there will be cells that are not actually connected to any of the outputs, as indeed is the case for cells with outputs 9 and 10 in the example above. These are redundant for the circuit they define, and may be removed when the chromosome is analysed. There are other forms of the cell redundancy possible which cause a particular cells outputs to be stuck at either one or zero even though the inputs to the cell are not fixed. This happens when the inputs change together. There is also logic redundancy. This is the case when the cell can be removed from the circuit and the functionality of circuit has not being changed. For example, the cell labeled 6 can be redundant, because the inputs of this cell are connected to the output of the cell 5 and as a result the functions describing the behaviour of logic cells 5 and 6 are equal. Hence, cell 6 can be removed from the circuit.

**Objective Function and Fitness.** The fitness function defines the percentage of output bits which are correct. A fitness function, that is equal to 100%, defines the fully functional design. The algorithm immediately terminates on achieving 100%

functionality. For example, the fitness function $\mathcal{F}$ for the one-bit adder with carry (Fig. 2.9) is 100.0. This means that this circuit is fully functional and this implements the full adder.

**Application** The cartesian GP has been applied to such problems as binary circuit design [25], [110], [28], [53], real-valued function design [102], multi-valued logic design [7], [8], low pass filter design [29], [104] (see Table 2.3). The progress of this approach is given in Table 2.3.

### 2.4.3 Assemble and Test

The principles of Assemble and Test in Cartesian GP have been considered in detail in [9]. Every binary and multiple-valued function is specified by a truth table. The truth table specifies what values the outputs of a function are for all values taken by the function inputs. There are certain special collections of operators that act on a binary or multiple-valued function that have the property that *any* function can be represented by expressions involving these operators and the input variables. The collection of these operators and the sets they operate on is often referred to as an *algebra*. In the case of binary functions there are two well-known algebras: Boolean which uses AND, OR and NOT and Reed-Muller (R-M) which uses AND, EX-OR and NOT. Multiple-valued logic also has its own algebras and often are referred to as *functionally complete bases*. When these algebras are used, a given function can only be represented by a particular class of expressions. The basic concept is shown in Fig. 2.10.

The unknown region in Fig. 2.10 depicts all the representations of logic functions which are written as an expression which does not use operations taken from the set [NOT, AND, OR, EX-OR]. Any expression in this region, *once known*, could be

Table 2.3: Development of cartesian GP. IE is an incremental evolution; ES is an evolutionary strategy; GA is a genetic algorithm; VES and VGA are the variable-length chromosome ES and GA respectively; $f(n,m,r)$ is an $n$-input $m$-output $r$-valued logic function; $f_{mux}(3,1,2)$ is an logic function described the behaviour of multiplexer; $F_1$ defines the correctness of outputs of logic circuit evolved; $F_2$ is the minimal number of logic cells used; $F_3$ is the correctness of input combinations; $F_4$ is an error based fitness; $\mathcal{F}_1 + \mathcal{F}_2$ is a dynamic fitness function.

| **Author** | **Building block** | **Fitness function** | **Circuit layout** | **EA** | **Application** |
|---|---|---|---|---|---|
| Miller J. et al, 1997 [25], [111], [110], [112] | $f(2,1,2)$ $f_{mux}(3,1,2)$ | $F_1$ | Fixed | GA | Arithmetic binary circuit design |
| Miller J., 1998 [102] | $f(2,1,2)$ $f_{mux}(3,1,2)$ | $F_1$ | Fixed | GA | Real-valued function design |
| Kalganova T. et al 1998 [8], [7], [9] | $f(2,1,2)$, $f_{mux}(r-1,1,2)$ | $F_1$ | Fixed | GA | Multi-valued circuit design |
| Kalganova T. et al 1999 [1], [3], [2] | $f(2,1,2)$, $f_{mux}(3,1,2)$ | $\mathcal{F}_1 + \mathcal{F}_2$ | Flexible | VGA | Binary circuit design |
| Miller J., 1999 [104], [28], [104] | $f(2,1,2)$ $f_{mux}(3,1,2)$ | $F_4$ | Fixed | ES | Low pass filter design |
| Miller J., 1999 [53] | $f(2,1,2)$ $f_{mux}(3,1,2)$ | $F_1$ | Fixed | ES | Parity function design |
| Kalganova T., 2000 [4] | $f(m,n,2)$ | $\mathcal{F}_1 + \mathcal{F}_2$ | Fixed | VES | Binary circuit design |
| Kalganova T., 2000 [4] | $f(m,n,r)$ | $\mathcal{F}_1 + \mathcal{F}_2$ | Fixed | VES | Multi-valued circuit design. |
| Miller J. et al 2000 [113] | $f(3,1,2)$ | $F_1$ | Fixed | GA | Symbolic regression problem |
| Miller J. et al 2000 [113] | $f(3,1,2)$ | $F_1$ | Fixed | ES | Santa Fe Ant Trail problem |
| Kalganova T., 2000 [6] | $f(m,n,r)$ | $\mathcal{F}_1 + \mathcal{F}_2$ | Fixed | IE | Binary circuit design |

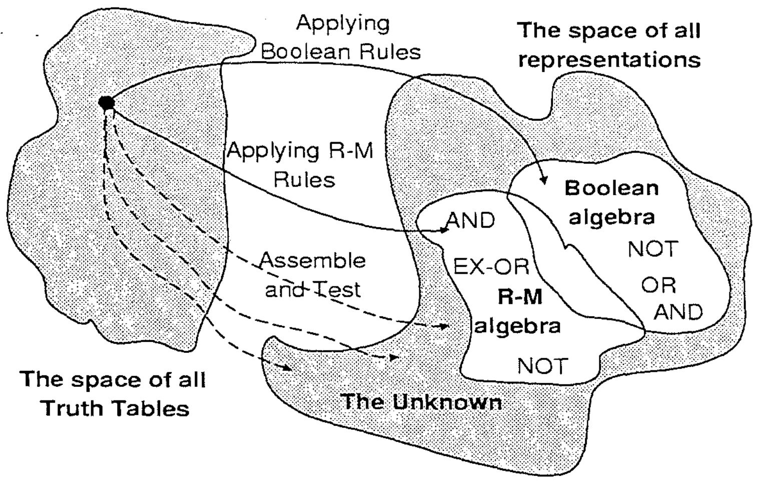


Figure 2.10: How *assemble-and-test* reaches the unknown regions of the space of all representations.

manipulated to become an expression in either the R-M or Boolean regions.

### 2.4.4 Advantages of an extrinsic EHW

One of the main advantages of extrinsic EHW methods proposed in the past is that evolution explores the whole search space without restriction to the function representation forms and logic algebras used. This allows us to synthesise new, more optimal logic circuits, principles [9]. For example, the evolved three-bit multiplier is more efficient in term of the number of primitive active logic gates, than a conventional one [9]. Scrupulous analysis of these circuits can produce some new principles to synthesise logic circuits and discover new logic algebra relations. For example, logic algebra involves four logic operators. For instance, some attempts to expose the new principles of construction the multipliers have been reported in [9]. In Chapter 7 some circuits that are not equal from point of view well-known logic algebra are discussed. More careful analysis of these circuits can produce new logic algebra relations. Although the extrinsic EHW methods have been actively studied in terms of their possible application areas, no work directed to investigate these methods in depth has been reported before.

### 2.4.5 Disadvantages of an extrinsic EHW

The extrinsic EHW approaches mentioned above are not capable of solving the complex design problems because of obstacles that appeared in the evaluation process.

- Direct evolution is inefficient at evolving a long chromosome string. This problem has been solved by introducing the variable length chromosome [114], function-level evolution [17], [4], automatically defined functions [51] and the divide-and-conquer approach (as a increased complexity evolution) [54], [55].

- Computation time requirements. The computation time required to process one chromosome increases exponentially with increasing the number of inputs in the implemented logic function. The problem can be solved by using a cube representation of logic function instead of a binary truth table. This issue will be discussed in Chapter 3. Cube representation has less number of input/output combinations in comparison with the truth table. Also, this problem can be diminished by using the parallel calculations [115].

- The number of generations required to perform a task can increase drastically with the complexity of the task. This is often called a stalling effect. For example, a two-bit multiplier (4 inputs, 4 outputs) can be easily evolved after 5, 000 generations [9]. Similarly, evolution of a fully functional three-bit multiplier (6 inputs, 6 outputs) at gate-level requires between 3, 000, 000 [9] and 10, 000, 000 generations [116]. According to this data we can predict that billions of generations are required to evolve a fully functional 4-bit multiplier

(8 inputs, 8 outputs). A similar trend can be observed in evolutionary robotics. In order to overcome this problem several researchers in the field of robotics have demonstrated that incremental evolution can be successfully applied to stochastic dynamic problems when implemented using neural networks [117], [59], [118]. In incremental evolution, neural networks learn complex general behaviour by starting with simple behavior and incrementally making the task more general.

- It should be mentioned that although at first sight the size of the search space for some instances of this problem may seem too small to even attempt to use a heuristic function, that is not true [26]. For the representation used in this dissertation, if we assume that a number of gates, to which the logic gate can be connected is $N_{connect}N_{rows}$, a chromosomic length of connectivity genes is $N_{inputs}N_{cols}N_{rows}$ and a number of logic gates in the functional set of logic gates is $|\mathbb{FS}|$, the size of intrinsic search space is $|\mathbb{FS}|^{N_{connect}N_{rows}^{N_{inputs}N_{cols}N_{rows}}}$. So it can be seen the size of search space depends on the number of columns and rows in the rectangular array, connectivity parameter and the number of logic gates in the functional set of logic gates.

The contribution of this work is to overcome the difficulties of extrinsic EHW mentioned above.

## 2.5 Incremental Evolution

While a number of researchers have used incremental evolution with different evolutionary learning methods, there seems to be no general comparison of different implementations of the technique. The incremental evolution can be applicable to the tasks where often a natural hierarchy of behaviours from simple to complex exists. As it can be seen below, incremental evolution is actively applied to synthesize controllers and closely incorporates with neuro-evolution.

Harvey et al. [119], [120] proposed strategy of incremental evolution. They use incremental evolution to train robots to move towards a white triangle on a dark background, but not towards a similarly-sized white rectangle. The robots are first trained to orient themselves to face a large white rectangle. Then the target is changed to a smaller white rectangle and robot pursuit is tested as this target moved. Finally, training focuses on the actual task of interest, which includes not moving towards the very rectangles the robots have already been trained to pursue. Comparisons made to those trained from search find incremental evolution provides more robust solutions.

Floreano et al. [57] discussed some hardware solutions that can support investigations in incremental evolution, namely modularity and cross-platform compatibility. Experiments demonstrate that the time required to generate interesting solutions (which is comparable or shorter than the time required by other learning techniques, such as reinforcement learning) does not imply that the approach is not viable. They discovered that it is possible for incremental evolution to be successful, when the intermediate task is more difficult than the final task.

Nolfi et al. [121] used incremental evolution to train controllers for robots. Maneuvering physical hardware in real time makes training with robots prohibitively expensive. However, any simulation of the robot uses an approximate model, which simplifies the noise and environment and can influence learning. This research finds significant degradation in performance when solutions trained in simulation are transferred to physical hardware. Training the simulation-evolved controllers for a few generations with the actual robots raises the real-world performance to simulation levels.

Winkeler et al. [122] present a study of the different methods of incremental evolution described in [121] and [120], as well as some alternative methods. Mixed incremental training differs significantly from the methods in theses previous works. Standard incremental evolution begins with a population already trained on a single task that is simpler than, but related to the task at hand. In mixed increments, multiple populations trained on a number of simplified tasks are combined to create a starting point. This new technique is shown to be robust against poor choices of simplified tasks.

Chavas et al. [123] used an incremental evolution to simulate the evolution of neural controllers for robust obstacle-avoidance in a Khepera robot and proved to be more efficient than a direct approach. During a first evolutionary stage, obstacle-avoidance controllers in medium-light conditions are generated. During the second evolutionary stage, controllers avoiding strongly-lighted regions, where the previously acquired obstacle-avoidance capacities would be impaired, are obtained.

Gomez and Miikkulainen [59] proposed to apply incrementally neuro-evolution to achieve the desired complex behaviour. In this approach, enforced sub-populations allow evolution of recurrent networks which are necessary for tasks that require memory. The Delta-Coding technique allows the evolution of transitions between tasks even when the population has lost diversity during the previous task.

King and Novak [124] advocated incremental evolution of the database layer (based on integrating only those pieces of the schema that are somehow semantically interrelated). Making persistence evolution, an incremental process does not only reduce the cost of doing this evolution, but it, in many instances, turns intractable processes into tractable ones.

Ramakrishnan [125] modeled business rules in resource allocation jobs and shows how this resource allocation model allows an incremental evolution of the system by enabling the behaviour of resource allocation classes to be extended through reuse mechanisms, and how this facility of incremental evolution of systems can be extended by capturing the dynamic components of the system explicitly as a second level of constrains.

Fukunaga et al. [58] reported some experiments to show that incremental evolution can be used as a technique for improving the performance of genetic programming.

Kintano [126] tried to establish a methodology for building very large complex system which has functional structures. He presented a method of developing very complex structures based on a grammar-based approach. The introduction of novel meta-node and associated operations is the essential feature of the method. The performance of the method has been demonstrated by actually generating neural network topologies in human neural subsystems; receptive field of skin somatic sensors and cerebeller cortex.

Artificial evolution with genetically converged populations occurs under the following circumstances [56]:

1. Incremental evolution, when the problem itself changes over time, for instance a sequence of related problems of increasing complexity may be tackled by a continuing evolutionary algorithm.

2. This may include cases where the genotype length may change in the course of

evolution, particularly where increasingly complex solutions require increasing genotypes in order to specify them.

3. It is becoming increasingly recognized that even in standard GA optimisation problems, an initially random population typically becomes genetically convergent after very few generations.

4. This may include cases where the computation requirements to evaluate one chromosome, describing a complex problem, are significantly high and therefore requires a lot of computational efforts in order to proceed the task using standard GA.

All these four principles can be directly applied to the EHW that solves the digital circuit design problem.

The first case occurs in digital logic design with EHW when the quality of evolved circuit can be changed during evolution, even if the functionality of the circuit remains the same. For example, in [103], [1] a dynamic fitness function is used to evolve a fully functional circuit with optimal structure.

The size of genotype in extrinsic EHW depends on the number of logic gates involved in the evolution. The second case appears, when in order to solve the complex problem, a large number of logic gates is required. This increases the size of genotype.

The third case has been studied at extrinsic EHW in the context of evolving arithmetic digital functions such as two-bit and three-bit multipliers. The number of generations required to perform a task increases drastically with increasing the complexity of task. Thus, a two-bit multiplier can be easily evolved after 5 000 generations [9]; at the same time the evolution of fully functional three-bit multiplier at gate-level requires from 3 000 000 generations [9] up to 10 000 000 generations [116]. According to this data we can predict that billions of generations are required to evolve fully functional 4-bit multiplier. In this case, it becomes practically impossible to perform this task.

The last case can be studied on the example of implementing logic function of large number of variables. The number of input-output combinations in logic function increases exponentially with increasing the number of variables. At the same time the complexity of logic function expands with enlarging the number of variables. This means that it requires a large number of logic gates to implement the desired logic function. All these facts mentioned above lead to increasing the computational time required to evaluate one chromosome. In this case, application of GA to digital circuit design becomes practically impossible.

Therefore this makes artificial evolution not only possible but also essential when the EHW is applied to solve complicated task.

## 2.6 Motivation of presented research

In the previous chapters it has been shown despite the fact that the EHW is very new research, a lot of work has been done by exploring the new application areas of this new method. The empirical study of this approach shows that the extrinsic and intrinsic EHW perform similar tasks. Using the extrinsic EHW approach at the first stage of investigation allows designer reduce the cost and efforts of investigation of the basic behaviour of EHW. Therefore, in order to understand the circuit evolution in general, it is important to investigate the comportment of extrinsic EHW specifically. Overview of research that has been achieved in this area shows that most of researches are concentrated on attempts to demonstrate applicability of extrinsic EHW to different problems rather then have a look more deeper inside the circuit evolutionary process. Understanding of this process allows us to improve its performance and evolve more optimal circuits.

From this point of view, we outline our research in the area of evaluation and evolutionary processes performed in the extrinsic EHW. It could form the basis of a dynamic fitness function and evaluation process in general and of a self-adaptive function-level EHW method. The first is important because the correct evaluation process benefits the evolutionary process and, further, allows to evolve cost-optimised circuits. The second is important because there is a lack of an existing extrinsic EHW approach to design of combinational logic circuits. This is especially true in the case of those systems where the logic function is given by a truth table and direct evolution is used to synthesise the circuit. The number of input combinations in the truth table increases exponentially with increasing the number of variables in logic function. Therefore, the computational efforts to estimate a circuit increases exponentially as well. Furthermore, the "stalling" effect is occurred in direct evolution when the complex problem is evolved. By these two reasons it becomes practically impossible to evolve complex circuits using existing EHW methods.

The bidirectional incremental evolution presented in Chapter 6 will be a step towards evolving more complex circuits, basic to the variety of EHW approaches being developed for digital circuit design problems. This approach combines such discovered features of extrinsic EHW as dynamic circuit evaluation (Chapter 3), self-adaptability of EHW parameters (Chapter 4), higher-level evolution (Chapter 5). The discovered characteristics of extrinsic EHW are proved to be universal at least for digital circuit design problems, since the EHW behaviour has been explored for binary and multi-valued circuit design problems (see Chapter 7).

## 2.7 Scope of this Dissertation

This work is concerned with the latter of the two goals attributed to extrinsic EHW above, namely evolving circuits that implement logic functions of large number of variables and that are optimised by a number of criteria chosen by designer. I will be content to demonstrate that it is possible to evolve logic functions of large number of variables using extrinsic EHW. The actual implementation of the extrinsic EHW on actual chip is left for future work.

Understanding the nature of the extrinsic circuit evolution process will help a researcher to comprehend the behaviour of the circuits evolved on real hardware. Instead of focusing on exploring the application area of extrinsic EHW approach, I focused on "strengthening" of existing extrinsic gate-level EHW approach, investigation the behaviour of this method and possibilities to design fully automatic EHW that will solve relatively complex problem efficiently.

So, I concentrated on inquisition of evaluation and evolution processes in extrinsic EHW. First process defines the quality of evolved circuits and the second one responds on the performance of EHW. It is very important to comprehend how the evaluation process affects the problem statement. The choice of optimisation parameters can change the implementation technology and the quality of evolved tasks (see Chapter 3).

The set of EHW parameters chosen can drastically influence the EHW performance. Automatic choice of these parameters can avoid this problem. Analysing the experimental results obtained for gate-level EHW and understanding their nature, we can improve and speed up the evolutionary process as well as to solve more complex tasks. Thus, empirical knowledge of circuit layout behaviour in gate-level EHW gives us some idea about the necessity of evolving the circuit layout together with circuit functionality in order to define the correct circuit layout suitable for a considered problem (see Chapter 4). Automatic choice of EHW parameters can simplify the life of designer, but it will not be able to solve the complex problems.

There are a number of limitations, that we have to overcome to improve existing EHW approach. One of them is the chromosome length. In extrinsic EHW at hand the chromosome length depends on the complexity of solving task. More complex task is represented by larger genotype. In order to overcome this problem, the building blocks of higher complexity can be used (see Chapter 5).

Another problem that impedes the successful evolution of complex circuits is the computation time. The number of generations required to solve the problem successfully increases drastically with increasing the complexity of problem. This obstacle can be overcome by evolving the logic circuit incrementally. By this reason, the emergency of introducing the mixed incremental evolution, that combines the standard functional decomposition methods with evolutionary approach, can be explained (see Chapter 6). During incremental evolution we can not only evolve fully functional circuit, by also optimise it by given criteria. The techniques to improve the EHW performance can be used at each incremental step, that only improve the EHW performance in terms of the number of generations required and simplify the choice of the circuit layout necessary to perform each incremental task.

While the opportunistic nature of these methods presents difficulties in fully interpreting the results of an experiments, it may ultimately pose an advantage for automatic problem solving.

## 2.8 Contribution to knowledge

This work introduces a novel method, namely bidirectional incremental evolution that allows us to evolve complex combinational circuits. The method overcomes such disadvantages of previously proposed extrinsic EHW approaches, as long string chromosome, computational requirements and the "stalling" effect in evolution. Automatisation of this method requires designers to develop an extrinsic EHW approach that self-adapts to initial parameters, such as circuit layout, connectivity restrictions. Essential reductions of chromosome strings can be made using the function-level EHW approach.

An additional contribution of this work is the identification of requirements in evaluation process and optimisation criteria of evolved circuits.

The application of the extrinsic EHW approach to evolve combinational multi-valued logic circuits serves to demonstrate the similarities in EHW behaviour for both binary and multi-valued logic design.

## 2.9 Summary

In this chapter, the foundation has been laid for understanding a basic principles of evolvable hardware. Previous research has shown that evolvable hardware is efficient as it adapts to wide ranging environments both in analytical and empirical studies. It has been shown that in almost all applications EHW have some advantages over well-known classical and leaning design approaches. Despite the fact that Evolvable hardware area has been established only during the last decade a huge amount of applications have been found already, that include application areas from pattern recognition to control of complex robotic systems. The detailed extrinsic EHW methods and their advantages and disadvantages have been presented. And finally, the research area of this dissertation is justified.

# Chapter 3

# Analysis and verification of evolved circuits

In this chapter some specific features of the evaluation process in EHW are considered. Such aspects of the circuit evaluation as function representation, optimisation criteria, fitness function strategy are discussed. Probabilistic analysis of the genotypes and dynamic fitness function are also introduced in this Chapter.

## 3.1 Introduction

Previously the number of logic gates has been used as one of the design metrics to be minimised in a circuit, so that the goal was to generate fully functional circuits that required the smallest possible number of gates (chosen from a certain set defined by user) [103], [1], [26]. This metric is applied very well if the produced circuit is implemented using FPGA technology [25]. However, the use of this metric may not be realistic in VLSI systems design, where the emphasis is to decrease the whole manufacturing cost rather than reducing the total number of components used [127], [128], [129]. Therefore, the circuit design problem for VLSI systems design has to be restated in such a way that the issues previously mentioned are taken into account.

Therefore, we rephrase our goal so that now we are interested in generating fully functional circuits in which the whole manufacturing cost is minimum. In order to achieve this goal we introduce a *dynamic fitness function*, that contains two stages. At the beginning of the search, only compliance with the truth table is taken into account. Once the first fully functional solution appears, the evaluation process switches to a new fitness function in which fully functional circuits that have less manufacturing cost are rewarded. Each of these two fitness functions can optimise the circuit by a number of criteria. For example, the second fitness function can be represented using any multi-objective function in which a number of criteria are taken into account.

FPGA, CMOS, NMOS, PMOS, dynamic MOS have been chosen as target implementation technologies. The manufacturing cost of a MOS circuit is estimated by the number of transistors. The number of basic logic gates employed is used as an optimisation criteria for the FPGA-based circuit. Note that such criteria as the circuit area, the circuit delay, the circuit composition, etc. can be chosen as optimisation criteria of the manufacturing cost. Such multi-objective methods as the method of objective weighting, the method of distance functions, the MIN-MAX formulation can be employed to combine any of these criteria in one objective fitness function, that is used at the second stage of the dynamic fitness function.

Clearly, utilising the dynamic fitness function, we manipulate with two different evolutionary processes: (1) evolution toward a fully functional circuit and (2) evolution toward an optimised circuit. The structures of these two evolutionary processes can be different. In order to investigate how different types of genes located differently in genotype participate in both evolution processes, a probabilistic approach is introduced in first time. This approach analyses the circuits that bring some improvements in the evolutionary process, i.e. their fitness functions have been ameliorated in comparison with the previous one.

## 3.2 An extrinsic gate-level EHW

In this section we will describe an extrinsic evolvable hardware approach applied to digital circuit design. The problem of interest to us consists of designing a digital cost-optimised circuit that performs a desired logic function (specified by a truth table). The circuit can be optimised according to any chosen manufacturing cost, such as the number of primitive active logic gates, the number of transistors, circuit delay, etc.. We refer to a gate as a primitive logic gate if this gate implements any primitive one- or two-input logic function. The circuit evolution has been performed using a rudimentary $(1 + \lambda)$ evolutionary strategy with uniform mutation [29]. In this case a population of random chromosomes is generated and the fittest chromosome is selected. The new population is then filled with mutated versions of this. The basic concept of the chromosome representation has been considered in Section 2.4.2, [25], [110], [1].

## 3.3 Function representation in an extrinsic EHW

The evaluation process in most EHW approaches, applied to the digital circuit design, contains the testing of evolved network on correctness. In this case, the function is defined using truth table [98], [25], [115]. This function representation is not suitable for evolving circuits having a large number of inputs, because the computational efforts increase exponentially with increasing the number of inputs in logic functions.

The evaluation process can be speeded up by using the so called sub-machine-code GP technique [115]. This approach exploits the internal parallelism of sequential CPUs and allows to speed up the program performance, but it will not help to solve the problem of evolving logic functions of large number of variables. In order to overcome this problem, we propose to use the ternary function representation. The input combinations are given by cubes. The logic function can be given by both complete and incomplete set of input combinations. Hence, both completely and incompletely specified logic functions can be evolved. In this section we will consider the specific features of the evaluation process performed for logic functions specified by truth tables, minterm tables and cubes.

### 3.3.1 Boolean functions specified by truth and minterms tables

The test as to whether the evolved circuit performs the desired logic translation of inputs to outputs is achieved by running *all* test inputs through the network, and comparing the results with the desired functionality in a bit-wise fashion. The number of input combinations for the $n$-input completely specified Boolean logic function is $2^n$. Therefore, this approach can be applied to design the logic functions of small number of variables, because the number of input combinations, needed to be evaluated, increases exponentially with increasing the number of inputs.

A PLA file (truth table of logic function) contains the target function, and this is used as a basis for comparison. The percentage of total correct outputs in response to appropriate inputs is then used as the *fitness* measure for the evolutionary algorithm. In other words, the nearer the evolved circuit comes to performing the desired functionality, the fitter it is deemed to be. Therefore, the fitness function is defined

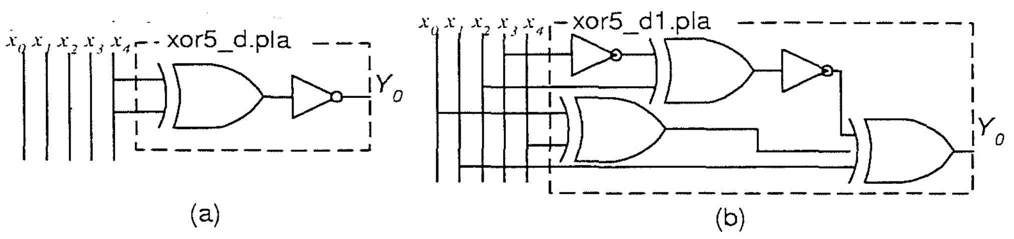


Figure 3.1: A 5-digit parity circuit evolved using (a) minterms table; (b) truth table. Functional set: $\mathbb{FS} : \{2, 7, 8, 9\}$.

as follows:

$$\mathcal{F} = \frac{\sum_{f_c=0}^{p-1} \sum_{i=0}^{m-1} 2^{i-1} \cdot |y_i - d_i|}{m * p} * 100. \qquad (3.3.1)$$

where $n$ and $m$ are the number of inputs and outputs in logic function respectively; $p$ is the number of ON and OFF sets (ignoring DON'T CARE conditions) in the Boolean logic function (if $p = 2^n$, then the Boolean function is completely specified, else the function is incompletely specified); $\{y_0, y_1, ..., y_{m-1}\}$ are the $m$ digits of the output combination produced by the evaluation of the circuit, $\{d_0, d_1, ..., d_{m-1}\}$ are the desired outputs (for the fitness case $f_c$), $|y_i - d_i|$ is the absolute difference between the actual output and the desired output.

Table 3.1: A 5-digit even parity function given by minterms table (xor5_d.pla) and by truth table (xor5_d1.pla). $X = \{x_0, x_1, x_2, x_3, x_4\}$, $Y = \{y_0\}$

| **xor5_d.pla** | | | | **xor5_d1.pla** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $X$ | $Y$ | $X$ | $Y$ | $X$ | $Y$ | $X$ | $Y$ | $X$ | $Y$ | $X$ | $Y$ |
| 00001 | 1 | 10000 | 1 | 00000 | 0 | 01000 | 1 | 10000 | 1 | 11000 | 0 |
| 00010 | 1 | 10011 | 1 | 00001 | 1 | 01001 | 0 | 10001 | 0 | 11001 | 1 |
| 00100 | 1 | 10101 | 1 | 00010 | 1 | 01010 | 0 | 10010 | 0 | 11010 | 1 |
| 00111 | 1 | 10110 | 1 | 00011 | 0 | 01011 | 1 | 10011 | 1 | 11011 | 0 |
| 01000 | 1 | 11001 | 1 | 00100 | 1 | 01100 | 0 | 10100 | 0 | 11100 | 1 |
| 01011 | 1 | 11010 | 1 | 00101 | 0 | 01101 | 1 | 10101 | 1 | 11101 | 0 |
| 01101 | 1 | 11100 | 1 | 00110 | 0 | 01110 | 1 | 10110 | 1 | 11110 | 0 |
| 01110 | 1 | 11111 | 1 | 00111 | 1 | 01111 | 0 | 10111 | 0 | 11111 | 1 |

Note, that the desired function has to be given on *all ON and OFF sets.* In some cases the function is defined only on the input combinations where the value of function is "1". In this case we have to complete the truth table and specify all ON and OFF conditions. If the truth table is not complete, the EHW assumes that the function is incompletely specified. In other words the input conditions that are not included in the truth table are considered as DON'T CARES. Hence, the generated circuit correctly implements only the input-output combinations given in the desired truth table. The following example demonstrates how the circuit structure is changed with altering the set of input combinations for given function.

Let us consider the 5 digit even parity function. This function can be represented by the minterms table given in xor5_d.pla and the truth table defined in xor5_d1.pla (see Table 3.1). The xor5_d.pla describes the input-output combinations of the 5-digit even parity logic function, where this function takes the value "1". This representation is often used in *.pla files for one-output logic functions. The point is that the function representation defined by the *.pla file is employed to implement the function on PLA structure. Therefore, the specific features of this regular structure are taken into account in the representation form. In order to avoid it during evolution of the logic function, this has to be defined by the truth table given in xor5_d.pla. The xor5_d1.pla minterms table contains all possible input-output combinations for the 5-digit even parity function. The gate-level extrinsic EHW has been used to evolve the circuits implementing the logic function given by xor5_d.pla and xor_d1.pla files. The evolved circuit structures are shown in Fig. 3.1. The design depicted in Fig. 3.1(a) implements the logic function given by the xor5_d.pla file and Fig. 3.1(b) shows the generated circuit performed the function defined by the xor_d1.pla file. It is clear that the circuit shown in Fig. 3.1(a) does not realize the 5-digit even parity function. The point is that the xor5_d.pla file describes only the input combinations where the function is "1" and the input combinations where the function is "0" are considered as DON'T CARE and, hence, are not taken into account. Therefore, the circuit shown in Fig. 3.1(a) implements correctly the function given in file xor5_d.pla. The circuit utilises only the input $x_4$ and the circuit output is "1" for any output combinations. In other words, this circuit implements the constant "1". In this case the input combinations, where the function is "0", are considered as DON'T CARE. In this case, the algorithm defines the variables $x_0$, $x_1$, $x_2$ and $x_3$ as DON'T CARE and generates the constant output equal to "1". In the case of defining the logic function on *all ON and OFF sets* (xor5_d1.pla), the circuit given in Fig. 3.1(b) is generated. This circuit fully implements the 5-digit even parity function, because during evaluation process the input combinations where the function is "0" are taken into account. All inputs are employed in the circuit design.

So, we can conclude that the function *has to* be defined on *all ON and OFF conditions.* Evolution of the logic function given by minterms table will produce wrong functionality circuit. Therefore, the representation of logic function using minterms table is not suitable if the EHW approach is used to implement the logic function.

### 3.3.2 Boolean functions specified by cubes

In order to apply the ternary representation of logic function to the extrinsic EHW, the ternary logic has to be introduced.

## Notions from ternary algebra

In this section we present for later use some concepts and notation from ternary algebra. For more details see [130].

We use 0 and 1 to denote the usual logic values, and $\Phi$ to denote a third value, which will have several interpretations. The *uncertainty partial order* $\sqsubseteq$ on the set $\{0, \Phi, 1\}$ is defined as follows:

$$0 \sqsubseteq 0, \quad \Phi \sqsubseteq \Phi, \quad 1 \sqsubseteq 1, \quad 0 \sqsubseteq \Phi, \quad 1 \sqsubseteq \Phi,$$

and no other pairs are related by $\sqsubseteq$. The value $\Phi$ is considered *uncertain* whereas 0 and 1 are *certain.*

The smallest ternary algebra is $\mathbb{T} =< \{0, \Phi, 1\}, \cdot, \vee, \neg, 0, \Phi, 1 >$, where the set has only three elements, and $\cdot, \vee$ and $\neg$ are the ternary OR, AND and NOT operations as defined in Table 3.2.

Table 3.2: The ternary operations.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $x_1$ | 0 | 0 | 0 | $\Phi$ | $\Phi$ | $\Phi$ | 1 | 1 | 1 |
| $x_2$ | 0 | $\Phi$ | 1 | 0 | $\Phi$ | 1 | 0 | $\Phi$ | 1 |
| $\overline{x_1}$ | 1 | 1 | 1 | $\Phi$ | $\Phi$ | $\Phi$ | 0 | 0 | 0 |
| $x_1 \vee x_2$ | 0 | $\Phi$ | 1 | $\Phi$ | $\Phi$ | 1 | 1 | 1 | 1 |
| $x_1 \wedge x_2$, or $x_1 x_2$ | 0 | 0 | 0 | 0 | $\Phi$ | $\Phi$ | 0 | $\Phi$ | 1 |
| $x_1 \oplus x_2$ | 0 | $\Phi$ | 1 | $\Phi$ | $\Phi$ | $\Phi$ | 1 | $\Phi$ | 0 |

A *ternary function* of $n$ variables is any function $f$ from $\{0, \Phi, 1\}^n$ to $\{0, \Phi, 1\}$, for $n \geq 0$. To each $n$-tuple $\mathbf{a} = (\mathrm{a}_1, ..., \mathrm{a}_n) \in \{0, \Phi, 1\}^n$, the function $f$ assigns an unique value $\mathbf{f}(\mathrm{a}) \in \{0, \Phi, 1\}$. The Boolean function can be described using ternary representation as follows: To each $n$-tuple $\mathbf{a} = (\mathrm{a}_1, ..., \mathrm{a}_n) \in \{0, \Phi, 1\}^n$, the function $f$ assigns an unique value $\mathbf{f}(\mathrm{a}) \in \{0, 1\}$. The $n$-tuple describes the cube of $2 * N_\Phi$ input combinations, where $N_\Phi$ is the number of uncertain values in the cube. For example, the $n$-tuple $(11\Phi0\Phi)$ represents 4 input combinations: $\{11000, 11001, 11100, 11101\}$.

Any Boolean function can be represented using ternary logic.

Table 3.3: Boolean logic function given by truth table.

| $x_0$ | $x_1$ | $x_2$ | $y_0$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

A three-input one-output logic function, $y$ is given by the truth table shown in Table 3.3. This function of $x_1$, $x_2$ and $x_3$ variables can be also defined using ternary representation:

$$\begin{bmatrix} \Phi & \Phi & 1 \\ 0 & 1 & \Phi \\ \Phi & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

and analytical representation as follows:

$$y_0 = \overline{x}_0\overline{x}_1x_2 \vee \overline{x}_0x_1\overline{x}_2 \vee \overline{x}_0x_1x_2 \vee x_0\overline{x}_1x_2 \vee x_0x_1x_2.$$

Note that one Boolean function can be represented by different ternary matrices. The optimal representation of logic function is the representation using ternary matrices with minimal number of rows in it. The $n$-input $m$-output logic function, given as a truth table in Table 3.4 (binary_test2.pla), can be described using ternary representation as follows.

$$X = \begin{bmatrix} 1 & 0 & \Phi & 1 \\ \Phi & 1 & 0 & \Phi \\ 1 & \Phi & 1 & 0 \\ 0 & 0 & \Phi & \Phi \\ 0 & \Phi & 1 & \Phi \end{bmatrix}, \; Y = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}.$$

This function is incompletely specified since it is not given on the input combinations $< 1000 >$ and $< 1111 >$. This function can be also defined analytically as:

$$y_0 = x_0\overline{x}_1x_3 \vee \overline{x}_0\overline{x}_1 \vee \overline{x}_0x_2; \tag{3.3.2}$$

$$y_1 = x_1\overline{x}_2 \vee \overline{x}_0\overline{x}_1 \vee \overline{x}_0x_2;$$

$$y_2 = x_0\overline{x}_1x_3 \vee x_1\overline{x}_2 \vee x_0x_2\overline{x}_3.$$

The implementation of this function on PLA in shown in Fig. 3.2(a).

Table 3.4: A tested function given by the cube representation (ternary_test2.pla) and by the truth table (binary_test2.pla). $X = \{x_0, x_1, x_2, x_3\}$, $Y = \{y_0, y_1, y_2\}$

| **ternary_test2.pla** | | **binary_test2.pla** | | | | | |
|---|---|---|---|---|---|---|---|
| $X$ | $Y$ | $X$ | $Y$ | $X$ | $Y$ | $X$ | $Y$ |
| 10Φ1 | 101 | 0000 | 110 | 0101 | 011 | 1011 | 101 |
| Φ10Φ | 011 | 0001 | 110 | 0110 | 110 | 1100 | 011 |
| 1Φ10 | 001 | 0010 | 110 | 0111 | 110 | 1101 | 011 |
| 00ΦΦ | 110 | 0011 | 110 | 1001 | 101 | 1110 | 001 |
| 0Φ1Φ | 110 | 0100 | 011 | 1010 | 001 | | |

## Fitness function

The ternary representation of logic function can be used in evaluation process of the extrinsic EHW approach if the functional set of logic gates contains the logic functions described by the ternary logic operations shown in Table 3.2. A PLA file contains the

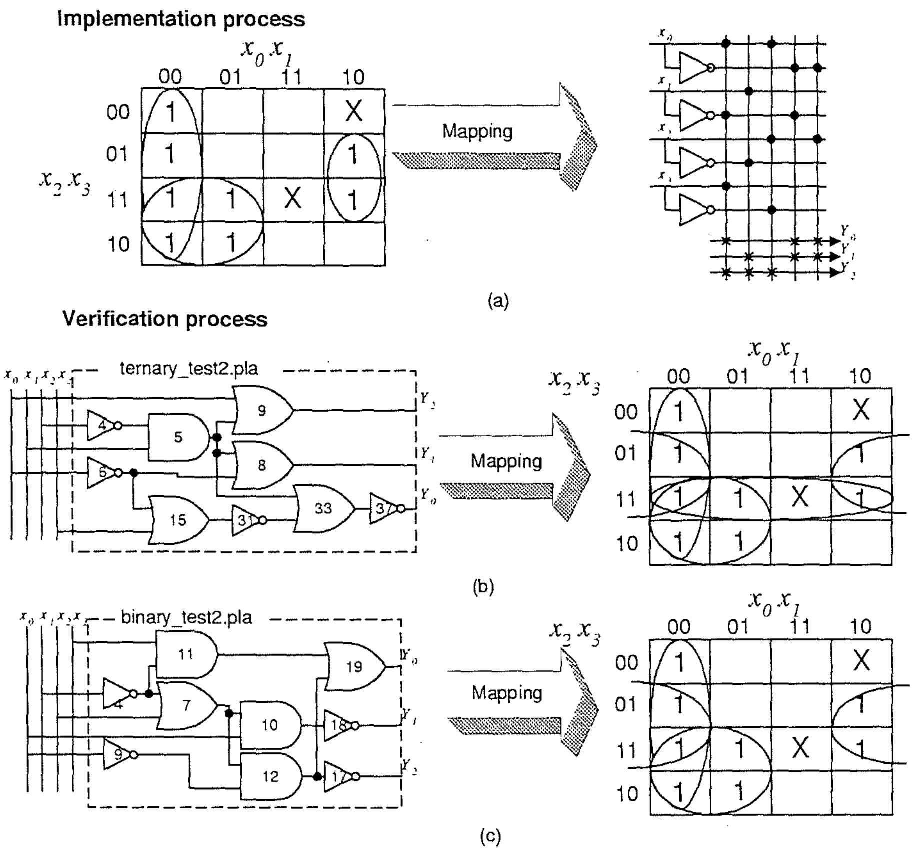


Figure 3.2: Implementation of logic function given in Eq. 3.3.2 (a) AND-OR PLA; In the standard logic design, the logic function is implemented according to the optimised representation, for example Karnaugh map. In given case, the PLA mapping is generated based on Karnaugh map. (b) circuit is evolved using cube representation; (c) circuit is evolved using the truth table. Unlike in the standard logic design, in EHW the circuit is synthesised independently from the representation form. In this process the representation form can be used in verification of the circuit correctness. In given case the Karnaught map is used to verify the correctness of evolved circuits.

Table 3.5: Initial data: Evolving logic functions using gate-level extrinsic EHW. $N_{in}^{max}(\mathcal{B})$ is the maximum number of inputs in the building block $\mathcal{B}$. $\mathcal{F}_1 + \mathcal{F}_2$ is the dynamic fitness function. The *truth table* representation of logic function corresponds to applying binary logic in EHW and the *cube representation* - to ternary logic.

| **Circuit, *.pla file** | **xor5_d** | **test2** | **add2c** | **mult2** |
|---|---|---|---|---|
| | **EHW parameters** | | | |
| **Circuit layout,** $N_{cols} \times N_{rows}$ | $1 \times 10$ | $1 \times 35$ | $1 \times 10$ | $1 \times 10$ |
| **Connectivity parameter,** $N_{connect}$ | 15 | 35 | 15 | 10 |
| **Functional set,** $\mathbb{FS}$ | $\{2, 6, 8\}$ | | $\{2, 6, 7, 8\}$ | |
| **Function representation** | Truth table<br>Cube representation | | Truth table | |
| **Gate distribution** | Proportional | | Proportional | |
| **Type of layout** | Fixed | | Fixed | |
| $N_{in}^{max}(\mathcal{B})$ | 4 | | 4 | |
| | **EA parameters** | | | |
| **Type of EA** | $(1 + \lambda)$ ES | | $(1 + \lambda)$ ES | |
| **Number of generations,** $N_{gen}$ | 5000 | 10000 | 15000 | 5000 |
| **Population size,** $\lambda$ | 5 | | 5 | |
| **Number of algorithm runs** | 100 | | 100 | |
| **Circuit mutation rate,** $p_{cm}$ | 0.05 | | 0.05 | |
| **Fitness strategy** | $(\mathcal{F}_1 + \mathcal{F}_2)$ | | $(\mathcal{F}_1 + \mathcal{F}_2)$<br>MIN-MAX formulation<br>Method of distance functions<br>Method of objective weighting | |

target function that is described by the ternary representation. In other words, the input combinations of desired logic function are given by cubes. This representation is used as a basis for comparison. The percentage of total correct outputs in response to appropriate input cubes is then used as the *fitness* measure for the evolutionary algorithm. The fitness function is defined as follows:

$$\mathcal{F} = \frac{\sum_{f_c=0}^{c-1} \sum_{i=0}^{m-1} 2^{i-1} \cdot |y_i - d_i|}{m * c} * 100. \tag{3.3.3}$$

where $n$ and $m$ are the number of inputs and outputs in the logic function respectively; $c$ is the number of cubes defining the input combinations of the desired Boolean logic function; $\{y_0, y_1, ..., y_{m-1}\}$ are the $m$ digits of the output combination produced by the evaluation of the circuit, $\{d_0, d_1, ..., d_{m-1}\}$ are the desired outputs (for the fitness case $f_c$), $|y_i - d_i|$ is absolute difference between the actual output and the desired output (the Hamming distance between $y_i$ and $d_i$).

The functional set of logic gates allowed to be used in evolution is defined by the primitive ternary logic operations such as AND, OR, NOT, NAND, EXOR, NOT with primary and inverted inputs and outputs given in Table 3.2.

**Experimental results**

Let us consider a 4-input 3-output incompletely specified logic function described in previous section and given in Table 3.4. Ternary_test2.pla file contains the cube representation of given logic function and binary_test2.pla file includes the truth table of this function (see Table 3.4). The truth table of this logic function contains no $< 1000 >$ and $< 1111 >$ input combinations. Therefore, the logic circuits are not tested on these input combinations. The gate-level extrinsic EHW approach has been used to evolve the circuits given by ternary_test2.pla and binary_test2.pla files. If the logic function is specified in cubes, the primitive logic gates in the functional set are defined according to the Table 3.2. If the logic function is given by truth table, the primitive logic operations discussed in Chapter A are used in the functional set.

The circuit designs synthesized by EHW given by cube and truth table are illustrated in Fig. 3.2(b), (c) respectively. These designs have been evolved using initial data given in Table 3.5. Calculating the circuit for *all* input combinations we can notice, that the circuits are not equal. Thus, the output $Y_0$ is 1 on the input combination

$< 1111 >$ for the circuit illustrated in Fig. 3.2(b) and is 0 on the input combination $< 1111 >$ for the circuit shown in Fig. 3.2(a). Therefore, from algebraic point of view the outputs ($Y_0$) in both circuits are not equal. But, since the logic function is incompletely specified, and this input combination is considered as DON'T CARE, the circuits correctly implement the given logic function. The designs depicted in Fig. 3.2(b) and (c) have been tested using both ternary and binary function representations. The result of test proves that both designs implement the function given either by the truth table or by the ternary representation. Thus, this exemplifies that the *correct* fully functional design can be evolved if the logic function is given by the truth table or by cubes.

## 3.4 Fitness function strategies

In this section we will introduce two-stage fitness function ($\mathcal{F}_1 + \mathcal{F}_2$), also called *dynamic fitness function* [1]. This fitness function strategy allows us to evolve the fully functional circuit and optimise the circuit structure by a number of criteria. The algorithm performance with the dynamic fitness function is compared with performances of similar algorithms utilising the multi-objective techniques such as the MIN-MAX formulation, the method of distance functions and the method of objective weighting. We choose these methods, because the optimisation of the single objective may guarantee a Pareto-optimal solution [131]. These methods work effectively if the objectives are clearly specified or have no discontinuous variable space [131]. This is applicable for our problem. We achieved relatively good results using these methods because we had knowledge of the priority of each objective before forming the single objective from a set of objectives and the knowledge of the value of optimal solution. In the following section we will consider some optimisation criteria used in an extrinsic EHW.

## 3.4.1 Pareto Optimum

The Pareto-Optimal solution can be defined as follows. Let find the vector $\overline{x}^* = [x_1^*, x_2^*, ..., x_n^*]^T$, which will satisfy the $m$ inequality constraints [107], [132]:

$$g_i(\overline{x}) \geq 0, \;\; i = 1, 2, ..., m \tag{3.4.1}$$

the $p$ equality constraints

$$h_i(\overline{x}) = 0 \;\; i = 1, 2, ..., p \tag{3.4.2}$$

and optimises the vector function

$$\overline{f}(\overline{x}) = [f_1(\overline{x}), f_2(\overline{x}), ..., f_k(\overline{x})]^T \tag{3.4.3}$$

where $\overline{x} = [x_1, x_2, ..., x_n]^T$ is the vector of decision variables. In other words, the particular set $x_1^*, x_2^*, ..., x_k^*$ which yields the optimum values of all the objective functions has to be determined from the set **F** of all numbers which satisfy Eq. 3.4.1 and Eq. 3.4.2.

A point $\overline{x}^* \in \mathbf{F}$ is *Pareto optimal* is for every $\overline{x} \in \mathbf{F}$ either,

$$\bigwedge_{i \in I}(f_i(\overline{x}) = f_i(\overline{x}^*) \tag{3.4.4}$$

or, there is at least one $i \in I$ such that

$$f_i(\overline{x}) > f_i(\overline{x}^*) \tag{3.4.5}$$

In words, this definition says that $\overline{x}^*$ is Pareto optimal if there exists no feasible vector $\overline{x}$ which would decrease some criterion without causing a simultaneous increase in at least one other criterion. The Pareto optimum almost always gives not a single solution, but rather a set of solutions called *non-inferior* or *non-dominated* solutions.

### 3.4.2 Criteria used in an extrinsic EHW

The main purpose of the extrinsic EHW approaches is to evolve the fully functional circuit with optimal parameters. Any type of circuit parameters can be chosen to define the quality of evolved circuit. It can be the number of primitive logic cells in the circuit, the number of transistors in evolved circuit, the circuit delays, etc.. In this work the following optimisation criteria are investigated:

1. the percentage of correct output *bits*, $F_1$;

2. the number of active primitive logic gates in the circuit, $F_2$;

3. the percentage of correct output *combinations*, $F_3$;

4. the cost of the circuit in terms of the number of used transistors, resistors, capacitors, etc., $F_4$.

One of the objective of the circuit design is to construct a fully functional circuit optimised by given criteria. In EHW approaches the search for the desired circuit begins with the random generated circuits. This means that initially the chosen circuits are not fully functional. Hence, there are two main objectives in EHW:

1. to evolve a fully functional circuit;

2. to optimise the evolved circuit by given criteria.

The first objective of EHW can be achieved by checking the actual circuit for correctness. It can be provided by criteria $F_1$ and $F_3$ mentioned above. These criteria show how close the actual circuit functionality to the requested one. The $F_1$ and $F_3$ criteria can be calculated as follows:

$$F_1 = \frac{\sum_{f_c=0}^{p-1} \sum_{i=0}^{m-1} 2^{i-1} \cdot |y_i - d_i|}{m * p} * 100; \tag{3.4.6}$$

$$F_3 = \frac{\sum_{f_c=0}^{p-1} \cdot |\mathbf{y}_{\mathbf{f_c}} - \mathbf{d}_{\mathbf{f_c}}|}{p} * 100; \tag{3.4.7}$$

where $p$ is the number of input combinations in the given logic function; $\{y_0, y_1, ..., y_{m-1}\}$ are the $m$ digits of the output combination produced by the evaluation of the circuit; $\mathbf{y}_{\mathbf{f_c}}$ is the circuit output vector corresponding to the $f_c$ input combination; $\{d_0, d_1, ..., d_{m-1}\}$ are the desired outputs (for the fitness case $f_c$); $\mathbf{d}_{\mathbf{f_c}}$ is the desired output vector corresponding to the $f_c$ input combination; $|y_i - d_i|$ is the absolute difference between the actual output and the desired *outputs*; $|\mathbf{y}_{\mathbf{f_c}} - \mathbf{d}_{\mathbf{f_c}}|$ is the absolute difference between the actual and the desired *output vectors.*

Depending on the circuit design task the evolved circuit can be optimised by different criteria. These allows us to evolve the optimal circuit structures optimised by given criteria and, therefore, achieve the second objective of EHW. Note that any optimisation criteria can be applied, such as delay in the circuit, connectivity restrictions, the number of primitive logic cells in the circuit, the circuit area, the number of transistors used etc.. In this work we limit our investigation to evolving the circuits with the minimal number of active primitive logic cells used, $F_2$ (FPGA design) and the minimal number of transistors in the circuit, $F_4$ (MOS design). It is clear that these criteria have to be minimised during the evolutionary process and the criteria $F_1$ or $F_3$ have to be maximised. In order to perform the maximisation process, the inverted criteria, such as the number of non-active primitive logic gates in the circuit, $(F_2^{max} - F_2)$ and the cost of the circuit in terms of the number of non-used transistors, resistors, capacitors, etc., $(F_4^{max} - F_4)$, are applied, where $F_2^{max}$ and $F_4^{max}$ are the maximum possible number of logic gates and transistors in the circuit respectively. These two criteria can be calculated as follows. Let $cost(\mathcal{N}_f)$ be the size or cost of the fully functional circuit $\mathcal{N}_f$ that can be defined as follows:

$$cost(\mathcal{N}_f) = \sum_{j=0}^{j<N_{cols}*N_{rows}-1} cost(\mathcal{B}_j) \tag{3.4.8}$$

and the cost of the building block $\mathcal{B}_j$ is calculated as

$$cost(\mathcal{B}_j) = \begin{cases} N_j^p, & \mathcal{B}_j \text{ is committed building block} \\ 0, & \mathcal{B}_j \text{ is uncommitted building block.} \end{cases} \tag{3.4.9}$$

where $N_{cols}$ and $N_{rows}$ are the number of columns and rows in rectangular array respectively, $N_j^p$ is the minimal implementation cost of the building block $\mathcal{B}_j$. If criteria $F_2$ is applied, then $N_j^p$ defines the minimum number of primitive logic cells in the building block $\mathcal{B}_j$ and $cost(\mathcal{N}_f)$ represents the criteria $F_2$. If criteria $F_4$ is activated, then $N_j^p$ determines the minimal number of transistors required to implement the building block $\mathcal{B}_j$ using chosen implementation technology such as CMOS, NMOS, PMOS or dynamic CMOS and $cost(\mathcal{N}_f)$ describes the criteria $F_4$.

The maximum cost of network, $\mathcal{N}_f$ can be calculated as follows:

$$cost(\mathcal{N}_f^{max}) = \sum_{j=0}^{j<N_{cols}*N_{rows}-1} cost(\mathcal{B}_{complex}) \tag{3.4.10}$$

where $\mathcal{B}_{complex}$ is the most complex logic function from the functional set of logic gates used in evolutionary process; $cost(\mathcal{B}_{complex})$ is the minimal (optimal) implementation cost of the building block $\mathcal{B}_{complex}$. If criteria $F_2$ is employed, then $cost(\mathcal{B}_{complex})$ defines the minimum number of primitive logic cells required to implement the building block $\mathcal{B}_{complex}$. If criteria $F_4$ is activated, then $cost(\mathcal{B}_{complex})$ determines the number of transistors in the circuit implementation of the building block $\mathcal{B}_{complex}$.

The number of primitive unused logic cells $(F_2^{max} - F_2)$ and the number of unused transistors $(F_4^{max} - F_4)$ can be calculated as follows:

$$F_{2,4}^{max} - F_{2,4} = cost(\mathcal{N}_f^{max}) - cost(\mathcal{N}_f). \tag{3.4.11}$$

where $cost(\mathcal{N}_f^{max})$ and $cost(\mathcal{N}_f)$ are calculated according to the given criteria $F_2$ or $F_4$. The number of transistors in the circuit is defined according to the type of implementation technology used (for more details see Appendix A).

We consider the building block $\mathcal{B}_j$ as sub-circuit with the structure that is not allowed to be changed. So, the cost of the building block does not take into account how many outputs of building block have been involved. This means that both the uncommitted and committed gates inside the building block are taken into account.

So, according to the objectives of EHW, all optimisation circuit criteria can be divided into two main categories. The first category includes the criteria which describe the circuit functionality ($F_1$, $F_3$) and the second category contains any criteria which define the quality of evolved circuit (in our case: $F_2$, $F_4$). Since the criteria $F_2$ and $F_4$ are defined for different implementation technologies, there is impossible to use both at the same time in the evaluation process.

### 3.4.3 Dynamic fitness function, $\mathcal{F}_1 + \mathcal{F}_2$

Our goal is to produce a fully functional design (i.e., one that produces the expected behaviour stated by its truth table) and produces the optimal structure by chosen criteria. Therefore, we decided to use the *dynamic fitness function* $\mathcal{F}_1 + \mathcal{F}_2$. At the beginning of the search, only compliance with the truth table, defined by fitness function $\mathcal{F}_1$, is taken into account, and the evolutionary approach is basically exploring the search space. Once the first functional solution appears, we switch to a new fitness function $\mathcal{F}_2$ in which fully functional circuits with the best optimised parameters are rewarded. According to two main objectives of EHW, the circuit evolutionary process $\Upsilon$ can be divided into two sub-processes, that:

1. Produces the fully functional circuit, $\Upsilon_{\mathcal{F}_1}$;

2. Improves the quality of evolved fully functional circuits, $\Upsilon_{\mathcal{F}_2}$.

Defining the fully functional circuit as well as its optimisation is performed *entirely* by evolutionary algorithm. Thus, the optimisation process is based *completely* on evolution rather than on well-known logic optimisation techniques. Each of $\mathcal{F}_1$ and $\mathcal{F}_2$ can represent a single optimisation criteria or a function forming a single objective from a set of objectives. The forming of function can be performed using any of classical multi-objective optimisation methods: the MIN-MAX formulation, the method of distance functions, the method of objective weighting, etc.. The criteria used in $\mathcal{F}_1$ and $\mathcal{F}_2$ can be chosen according to the final result that is produced once these criteria is applied. During first stage of evolution $\Upsilon_{\mathcal{F}_1}$, one of $F_1$ or $F_3$ criteria or both of them are used (i.e., $\mathcal{F}_1 \in \{F_1, F_3\}$). $\mathcal{F}_1 = 100$ defines that the fully functional solution has appeared. The criteria $F_2$ and $F_4$ or both of them are utilised during second stage of evolutionary process, $\Upsilon_{\mathcal{F}_2}$ (i.e., $\mathcal{F}_2 \in \{F_2, F_4\}$). So, the fitness function $\mathcal{F}_2$ is activated if $\mathcal{F}_1 = 100\%$. The dynamic fitness function is calculated as follows:

$$\mathcal{F} = \begin{cases} \mathcal{F}_1, & \mathcal{F}_1 < 100; \\ \mathcal{F}_1 + \mathcal{F}_2, & \mathcal{F}_1 = 100. \end{cases}$$

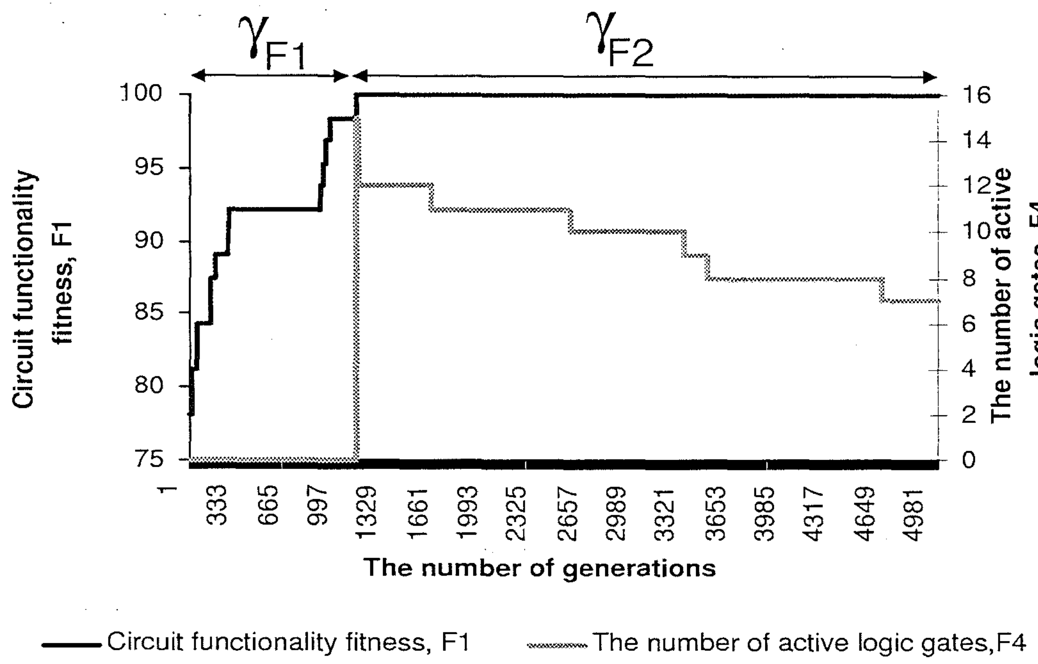


Figure 3.3: Behaviour of dynamic fitness function. The graph depicts the best fitnesses $\mathcal{F}_1$ and $\mathcal{F}_2$ of the best chromosome. The two-bit multiplier is evolved during 5000 generations using dynamic fitness function. During $\Upsilon_{\mathcal{F}_1}$ only the circuit functionality criteria $F_1$ is taken into account, hence $F_4 = 0$. During $\Upsilon_{\mathcal{F}_2}$ the circuit functionality *should* remains the same ($F_1 = 100$) and the number of active gates is targeted to minimise. The graph shows clearly that two distinctive evolutionary processes are performed to evolve a cost-optimised fully functional two-bit multiplier.

The members of the population have their fitness calculated, if their genotype has been changed during evolution. It is the fitness function the only agent responsible for the life span of the individuals.

Let us consider an example of using dynamic fitness function. Let the problem of interest is to evolve the cost-optimised fully functional two-bit multiplier. According to the problem, two optimisation criteria can be chosen to participate in evaluation process: $F_1$ and $F_4$. Criteria $F_1$ is responsible for evolving the fully functional circuit. $F_4$ is liable for evolving the cost-optimised circuit. Therefore, $\mathcal{F}_1 = F_1$ and $\mathcal{F}_2 =$

$F_4$. Fig. 3.3 demonstrates the how these criteria changes with time. The graph depicted in Fig. 3.3 shows clearly the difference between two evolutionary processes defined by dynamic fitness function. The evolutionary process $\Upsilon_{\mathcal{F}_1}$ begins at the 0-th generations and terminates at generation 1100. The second evolutionary process starts at generation 1100 and terminates at generation 5000. At the beginning of both evolutionary processes, a significant improvement can be noticed. Thus during $\Upsilon_{\mathcal{F}_2}$ the number of active logic gates is reduced significantly during the first generations. This example shows that these two processes have to be investigated separately since the nature of evaluation process has been changed.

### 3.4.4 Method of objective weighting

In this method, objective functions are combined into one overall objective function, $\mathcal{F}$, as follows [131], [133]:

$$\mathcal{F} = \sum_{i=1}^{N} w_i f_i(\mathbf{x}), \tag{3.4.12}$$

where $\mathbf{x} \in \mathbf{X}$, $\mathbf{X}$ represents the feasible region; the weights $w_i$ are fractional numbers ($0 \leq w_i \leq 1$), and all weights are summed up to 1, or $\sum_{i=1}^{N} w_i = 1$. In this method, the optimal solution is controlled by the weight vector $\mathbf{w}$. It is clear from Eq. 3.4.12 that the preference of an objective can be changed by modifying the corresponding weight. A solution obtained with equal weights to all objectives may offer least objective conflict. But as a real-world situation demands a satisfying solution, priority must be introduced into the formulation. In our case each objective is first optimised and all objective function values are computed at each individual optimum solution. Thereafter, depending on the importance of objectives a suitable weight vector is chosen and the single-objective problem given in Eq. 3.4.12 is used to find the desired solution.

In this method applied to logic design problem, the weights are assigned to any chosen optimisation criteria $F_1$, $F_2$, $F_3$ and $F_4$ representing $f_i(\mathbf{x})$. For example, criteria $F_1$ and $F_4$ be chosen to participate in evaluation process and $w_1 = 0.6$, $w_2 = 0.4$. In this case the fitness function **F** can be defined as follows:

$$\mathcal{F} = 0.6F_1 + 0.4F_4. \tag{3.4.13}$$

### 3.4.5 Method of distance functions

In this method, the scalarisation is achieved by using a demand-vector $\overline{y}$, which has to be specified by the decision maker. This method is similar to the method of objective weighting. The only difference is that in this method the goal for each objective function is required to be known whereas in the previous method the relative importance of each objective is required. The single objective function derived from multiple objectives is as follows [131], [133]:

$$\mathcal{F} = \big[\sum_{i=1}^{N} |f_i(\mathbf{x}) - \overline{y}_i|^e\big]^{1/e}, \quad 1 \leq e \leq \infty, \tag{3.4.14}$$

where $\mathbf{x} \in \mathbf{X}$ (the feasible region). Usually a Euclidean metric $e = 2$ is chosen, with $\overline{\mathbf{y}}$ as individual optima of objectives [134]. Therefore the fitness is calculated as follows:

$$\mathcal{F} = \sqrt{\sum_{i=1}^{N} |f_i(\mathbf{x}) - \overline{y}_i|^2}. \tag{3.4.15}$$

It is important to note that the solution obtained by solving Eq. 3.4.15 depends on the chosen demand-level vector. Arbitrary selection of a demand level may be highly undesirable.

For instance, let the problem of interest is to evolve a two-bit multiplier with a minimal number of primitive active logic gates. Hence, $F_1$ and $F_4$ are chosen to be the optimisation criteria. Since the goal is to evolve a fully functional solution ($F_1 = 100$), then $\overline{y_1} = 100$. The most efficient synthesized two-bit multiplier contain 7 primitive logic gates (see Appendix C). This means that $\overline{y_4} = 7$. So, for a given task, the fitness function can be defined as follows:

$$\mathcal{F} = \sqrt{|F_1 - 100|^2 + |F_4 - 7|^2}. \qquad (3.4.16)$$

### 3.4.6 MIN-MAX formulation

This method is different in principle from the above two methods. This method attempts to minimise the relative deviations of the single objective functions from the individual optimum [131], [133]. That is, it tries to minimise the objective conflict between circuit functionality and cost-optimised criteria. For a minimisation problem, the corresponding MIN-MAX problem is formulated as follows:

$$\text{minimise } \mathcal{F}(\mathbf{x}) = \max[Z_j(\mathbf{x})], \; j = 1, 2, \cdots, N, \qquad (3.4.17)$$

where $\mathbf{x} \in \mathbf{X}$ (the feasible region) and $Z_j(\mathbf{x})$ is calculated for a nonnegative target optimal value $\overline{f}_j > 0$ as follows:

$$Z_j(\mathbf{x}) = \frac{f_j - \overline{f}_j}{\overline{f}_j}, \quad j = 1, 2, \cdots, N. \qquad (3.4.18)$$

This method can yield the best possible compromise solution when the objectives with equal priority are required to be optimised. However, the priority of each objective can by varied by introducing dimensionless weights in the formulation. This can also be modified as a goal-programming technique by introducing a demand-vector in the formulation.

For example, in the case of evolving the fully functional cost-optimal two-bit multiplier implemented using FPGA technology, $F_1$ and $F_4$ can be chosen as the optimisation criteria. As it has been mentioned in previous section, the most optimal two-bit multiplier design contains 7 primitive logic gates. Then, $\overline{F_4} = 7$. The target circuit functionality is 100%, then $F_1 = 100$. Therefore, the fitness function defined by MIN-MAX formulation is calculated as follows:

$$\text{minimise } \mathcal{F}(\mathbf{x}) = \max\left[\frac{F_1 - 100}{100}, \ \frac{F_4 - 7}{7}\right]. \tag{3.4.19}$$

### 3.4.7 Experimental results

In the following sub-sections we will consider some experimental results obtained for the two-bit adder and the two-bit multiplier. We investigate 1) Fitness function strategies in the extrinsic gate-level EHW; 2) Evolved circuit structures optimised by such criteria as the number of uncommitted primitive logic gates or the number of transistors in CMOS, NMOS, PMOS, dynamic CMOS circuits. The initial data for this series of experiments are given in Table 3.5.

#### Fitness strategies

The following experiment shows us how using different fitness function strategies with different optimisation criteria affects the algorithm performance and the quality of evolved circuits. For this purpose the same experiments have been performed using the dynamic fitness function, the method of distance functions, the MIN-MAX formulation and the method of objective weighting. The percentage of correct bits, $F_1$, the number of active primitive logic gates in circuit, $F_2$ and the percentage of correct output combinations, $F_3$ have been chosen as the criteria to compare the algorithm performance. The experimental results obtained are summarised in Table 3.6 and Fig. 3.4.

Table 3.6 makes a summary of the experimental results received for algorithms with the dynamic fitness function, the MIN-MAX formulation and the method of distance functions. In further discussion the following notations have been adopted. $\overline{F_1^{bf}}$, $\overline{F_2^{bf}}$ and $\overline{F_3^{bf}}$ are the mean fitness functions $F_1$, $F_2$ and $F_3$ of the best evolved chromosomes respectively. $\mathcal{N}_f$ denotes a fully functional circuit. $\overline{F_2(\mathcal{N}_f)}$ is the mean fitness function $F_2$ of the fully function designs evolved during 100 runs. $\mathcal{R}(\mathcal{N}_f)$ is the number of evolved fully functional circuits $\mathcal{N}_f$.

Fig. 3.4 shows the algorithm performance for different combinations of chosen criteria using method of objective weighting for the two-bit multiplier (mult2.pla) and the two-bit adder (add2c.pla). Curves in Fig. 3.4 show the fitnesses $F_1$, $F_2$ and $F_3$ of the best evolved chromosomes, $\overline{F_1^{bf}}$, $\overline{F_2^{bf}}$ and $\overline{F_3^{bf}}$ and the number of fully functional circuits evolved $\mathcal{R}(\mathcal{N}_f)$ as a function of the weight, for the two bit multiplier and the two-bit adder. Each data point gives the average of 100 runs. The results obtained during evolution of the two-bit multiplier are displayed at the left side of the graph. A summary of evolving the two-bit adder is shown at the right side of the graph. The horizontal axis defines the weight of fitness function $F_1$ (Graphs A – D in Fig. 3.4) and $F_3$ (Graphs E – F in Fig. 3.4). The weight vector is defined as follows:

$$\mathbf{w} = (w_1, w_2) = (w_1, 1 - w_1); \tag{3.4.20}$$

where $w_1$ is the weight shown in the horizontal axis. For example, graph A in Fig. 3.4 shows some experimental results obtained during the evolution of the two-bit multiplier. Two criteria have been chosen to perform this experiment: $F_1$ and $F_3$. The horizontal axe defines the weight of criteria, $F_1$. Then, the weight of criteria $F_3$, $w(F_3)$ is calculated according to the Eq. 3.4.20. Thus, if the weight of $F_1$, $w(F_1)$ is 0.7, then the weight of $F_3$, $w(F_3)$ is 0.3. A similar process is applied to define the weights of other criteria for experimental results shown in Fig. 3.4

### Using $F_1$ and $F_2$ criteria

This combination of criteria has been applied for all multi-objective methods discussed above. The fitness function $\overline{F_1^{bf}}$ is higher for the algorithm with the dynamic fitness function. The same conclusion is made about the quality of evolved fully functional circuits $\overline{F_2(\mathcal{N}_f)}$ and the number of received fully functional solutions $\mathcal{R}(\mathcal{N}_f)$. This is evidence of the effectiveness of using the dynamic fitness function. The value of $\overline{F_2(\mathcal{N}_f)}$ shows fairly good performance consistently on both tested functions. Let us consider the results obtained for the two-bit multiplier evolved using an algorithm with dynamic fitness function. The following data has been obtained: $\overline{F_1^{bf}} = 99.75$, $\overline{F_2^{bf}} = 7.28$, $\overline{F_2(\mathcal{N}_f)} = 7.25$ and $\mathcal{R}(\mathcal{N}_f) = 90$. The fully functional circuits evolved using the algorithm with the MIN-MAX formulation and the method of distance functions require at least 8 primitive logic gates ($\overline{F_2^{bf}}$). This means that the fully functional circuits $\mathcal{N}_f$ evolved using these methods contain approximately one primitive logic gate more than the similar circuits evolved using the algorithm with the dynamic fitness function. In other words more compact logic circuits have been obtained when the algorithm with the dynamic fitness function has been applied.

Analysing the experimental data obtained for the algorithm with the method of objective weighting we can conclude that the fully functional designs have been received when $w_1(F_1) \geq 0.4$ for mult2.pla function and $w_1(F_1) \geq 0.6$ for add2c.pla function. This means that the functionality fitness has to dominate, if the goal of evolution is to evolve a fully functional circuit. The optimal weight $w_1(F_1)$ for add2c.pla function is 0.8 and is 0.6 for mult2.pla function. The highest number of fully functional solutions has been obtained when these parameters have been employed.

Table 3.6: Experimental results: Fitness function strategy and algorithm performance. **X** denotes that the corresponding criteria has been activated; $\overline{F_1^{bf}}$, $\overline{F_2^{bf}}$ and $\overline{F_3^{bf}}$ are the mean fitnesses $F_1$, $F_2$ and $F_3$ of the best evolved chromosomes respectively; $\overline{F_2(\mathcal{N}_f)}$ is the mean fitness function $F_2$ of fully functional designs evolved during 100 runs; $\mathcal{R}(\mathcal{N}_f)$ is the number of evolved fully functional circuits, $\mathcal{N}_f$

| **Fitness** | **Criteria** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **strategy** | $F_1$ | $F_2$ | $F_3$ | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}$ | $\overline{F_2(\mathcal{N}_f)}$ | $\overline{F_3^{bf}}$ | $\mathcal{R}(\mathcal{N}_f)$ |
| **mult2.pla** | | | | | | | | |
| One-stage | X | - | - | 99.7812 | 7.96 | 7.98 | 99.1250 | 89 |
| Dynamic | X | X | - | 99.75 | 7.28 | 7.25 | 99 | 90 |
| fitness | - | X | X | 95.4843 | 8.96 | 7.8 | 91.25 | 5 |
| MIN-MAX | X | X | - | 97.125 | 7.44 | 8.5 | 95.125 | 24 |
| formulation | X | - | X | 98.2656 | 6.49 | 0 | 93.0625 | 0 |
| | - | X | X | 96.9531 | 7.14 | 7.21 | 94.6875 | 24 |
| Method of | X | X | - | 99.6719 | 7.96 | 8.08 | 98.8125 | 82 |
| distance | X | - | X | 99.6719 | 8.02 | 7.54 | 98.1250 | 77 |
| functions | - | X | X | 97.375 | 7.95 | 7.67 | 95.5 | 34 |
| **add2c.pla** | | | | | | | | |
| One-stage | X | - | - | 96.5729 | 16.1 | 16.5 | 90.3438 | 26 |
| Dynamic fitness | X | X | - | 97.1875 | 15.29 | 13.5882 | 92 | 34 |
| MIN-MAX | X | X | - | 93.0987 | 13.93 | 16.2778 | 85.6562 | 18 |
| formulation | X | - | X | 95.4062 | 14.32 | 11.75 | 87.0938 | 4 |
| | - | X | X | 92.75 | 13.68 | 13 | 84.1875 | 7 |
| Method of | X | X | - | 95.4687 | 15.97 | 15.53 | 88.625 | 15 |
| distance | X | - | X | 96.8958 | 29.99 | 30 | 91.0625 | 29 |
| functions | - | X | X | 96.0729 | 29.6 | 29.79 | 90.875 | 29 |

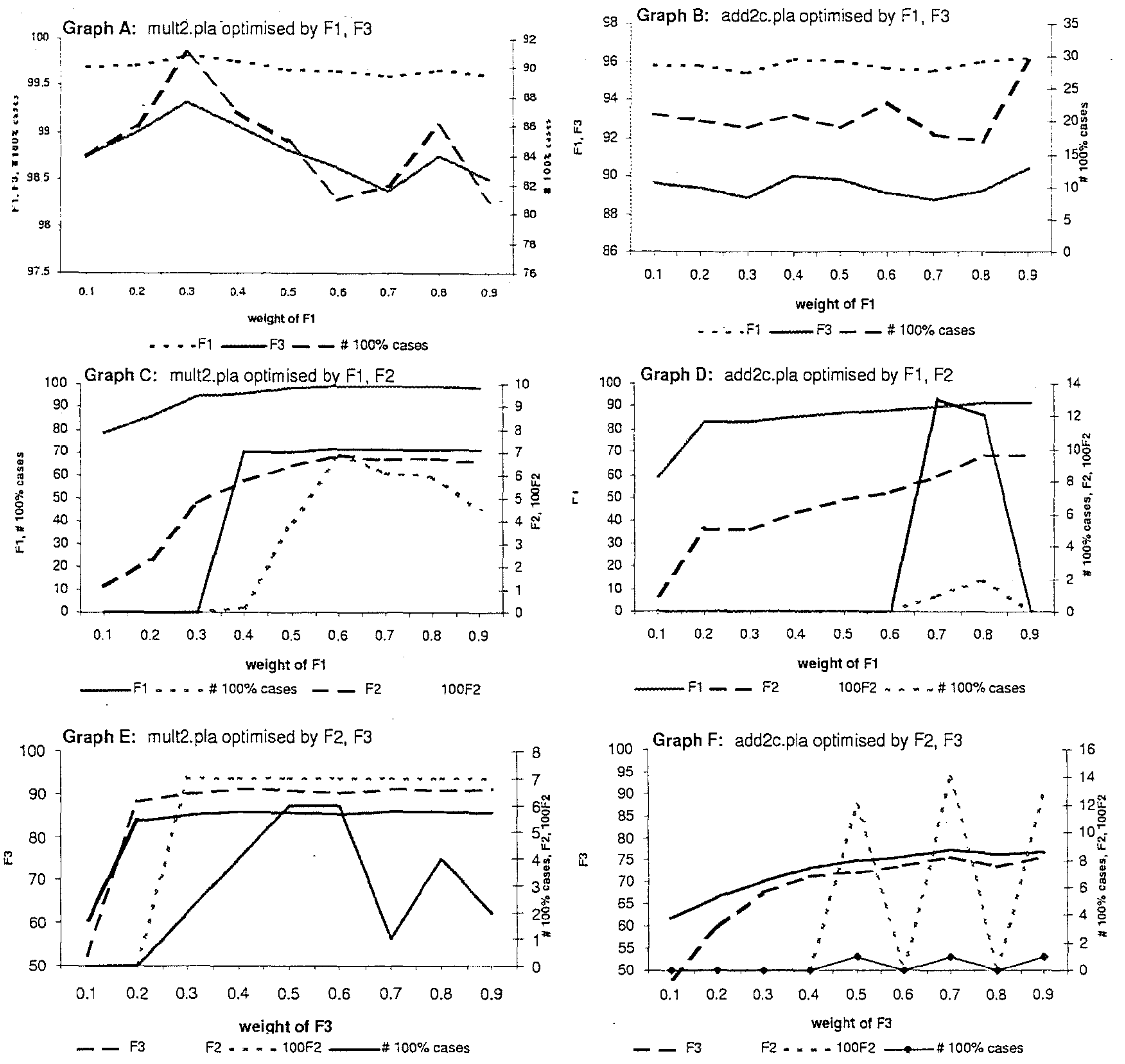


Figure 3.4: Experimental results: Method of objective weighting. $F_1$ and $F_3$ define the correctness of evolved circuits; $F_2$ determines the quality of evolved circuits; **Graphs A and B** illustrate that there is no dominated criteria among $F_1$ and $F_3$, since the evolutionary algorithm with different weights for both criteria performs similar for both logic functions: two-bit adder and two-bit multiplier. **Graphs C and D** demonstrate that $F_1$ is a dominated criterion. Thus, in the case when the weight of $F_1$ is less than the weight of $F_2$, no fully functional solutions have been evolved for both logic functions. **Graphs E and F** evidence that $F_3$ is a dominated criterion. Similarly to the previous case, the fully functional solutions have been obtained, if the weight of $F_3$ is large enough. With increasing the weight of $F_3$, the algorithm performance has been improved. **Conclusion:** In evolution process is the criterion defining the correctness of evolved circuits dominates other criteria, determined the quality of evolved circuits.

These results indicate that the priority has to be given to the functionality fitness during evolution of fully functional circuits.

So, we can conclude that the algorithm with the *introduced* dynamic fitness function performs much better than the algorithms with other multi-objective fitness functions for the task of evolving fully functional solutions optimised by given criteria.

### Using $F_1$ and $F_3$ criteria

Both $F_1$ and $F_3$ criteria define the circuit functionality, interpreted differently. $F_1$ shows the correctness fitness in terms of each output bit. $F_3$ points at the correctness of the circuit by the output combinations. In other words, these criteria participate actively during evolution of the fully functional circuit, $\Upsilon_{\mathcal{F}_1}$ and they do not improve the quality of evolved circuit. Therefore, the evolutionary process can be terminated once the fully functional solution has appeared. For this reason the dynamic fitness function strategy has not been considered in this experiment.

Considering the method of objective weighting, we can conclude that the fitness functions $\overline{F_1^{bf}}$ and $\overline{F_3^{bf}}$ show fairly good performance consistently on all combinations of weights. The fitness function $\overline{F_1^{bf}}$ achieves higher value for the algorithm with the method of distance functions than for the method of objective weighting. At the same time, the fitness function $\overline{F_2^{bf}}$ is the worst for the algorithm with the method of the distance functions. We can see that the algorithm with any combinations of weights shows nearly the same performance in terms of the number of fully functional solutions evolved. The higher number of fully functional solutions $\mathcal{R}(\mathcal{N}_f)$ has been evolved with the method of objective weighting. Thus, this number varies from 81 to 91 in the case of using the method of objective weighting for the two-bit multiplier.

In the case of employing other methods, the highest $\mathcal{R}(\mathcal{N}_f)$ has been achieved with employing the method of distance functions ($\mathcal{R}(\mathcal{N}_f) = 77.0$). We obtain from 17 to 30 fully functional two-bit multipliers when the algorithm with the method of objective weighting has been applied. 29 two-bit adders had been evolved when the method of distance functions has been applied. So, we can conclude that the method of distance functions and the method of objective weighting perform nearly the same in terms of evaluating the circuit using $\mathcal{R}(\mathcal{N}_f)$ and $\overline{F_1^{bf}}$ criteria.

$\overline{F_2(\mathcal{N}_f)}$ for add2c.pla is more than 2 times greater for the algorithm with the method of distance functions, than for the algorithms with other multi-objective methods. This means that evolved fully functional circuits contain 2 times more primitive logic gates than similar circuits obtained using other multi-objective methods.

It is obvious that for both tested logic functions the method of objective weighting performs better, as can be seen from the higher fitness functions $\overline{F_1^{bf}}$, $\overline{F_3^{bf}}$ and the number of fully functional designs evolved $\mathcal{R}(\mathcal{N}_f)$ that have been acquired.

### Using $F_2$ and $F_3$ criteria

Next, we look at the results obtained using $F_2$ and $F_3$ criteria. Let us remember that the criteria $F_2$ determines the quality of evolved circuits in terms of the number of active primitive logic gates used in a circuit and that the criteria $F_3$ points at the percentage of correct output combinations in evolved circuit. Therefore, two types of evolution are involved: evolving the circuit functionality, $\Upsilon_{\mathcal{F}_1}$ and improving the quality of evolved circuits, $\Upsilon_{\mathcal{F}_2}$. We can see that the weight of $F_3$ has to be higher than 0.3 for mult2.pla and greater than 0.3 for add2c.pla, in order to achieve some fully functional solutions. The algorithm with the method of distance functions shows better performance for evolving add2c.pla function. At the same time the algorithm with the method of objective weighting gives better performance for evolving mult2.pla function. Therefore, there is no clear evidence which of the multi-objective methods performs better.

In this subsection first, we were able to confirm that the dynamic fitness function in evaluation process has a good performance on both tested arithmetic functions if the criteria $F_1$ and $F_2$ are activated. Next, the method of objective weighting has a good performance on optimisation by $F_1$ and $F_3$ criteria. Also, it has been shown that the weight of circuit functionality has to dominate when the criteria of circuit functionality and the quality of evolved circuits have been applied. The experimental results suggest that the bit-by-bit circuit functionality criteria $F_1$ performs better than the criterion $F_3$, comparing the correctness of output combinations.

#### Evolving logic circuits using criteria $F_2$ and $F_4$

In this section we will consider some experimental results obtained for the two-bit multiplier. The main idea of this experiment is to define whether using different optimisation criteria during the second stage of dynamic fitness function affects the evolved circuit structures and algorithm performance or not. The initial data for the experiment are given in Table 3.7. The circuit structures are optimised using such criteria as the number of primitive logic active gates, the number of transistors in different circuit implementations (CMOS, NMOS, PMOS, dynamic MOS). We do not consider evolution of the PMOS circuits. The point is that the NMOS and PMOS implementations of the primitive logic gates require the same number of transistors. In order to examine how the functional set of logic gates used impacts the quality of evolved circuits, 6 functional sets have been chosen. Only primitive logic gates of one- and two-inputs are allowed to participate in evolution. Each gate is considered as a separate chromosomic element. We count each of them, including NOTs that associated with AND, OR and EXOR gates. For instance, the NAND gate contains two basic logic gates.

The obtained experimental results are summarised in Table 3.8. Let us remember that the *quality of evolved circuits* is defined by the number of primitive logic gates in circuit, $F_2$, and the number of transistors in the circuit produced using different implementation technologies, $F_{4CMOS}$, $F_{4DMOS}$, $F_{4NMOS}$. The *algorithm performance* can be measured by the number of fully functional solutions achieved, $\mathcal{R}(\mathcal{N}_f)$.

The results shown in Table 3.8 illustrate the dependence of the algorithm performance and the quality of evolved circuits on the functional set of logic gates $\mathbb{FS}$ for all optimisation criteria used. For example, $\overline{F_2(N_f)}$, achieved when $\mathcal{F}_2 = F_2$, varies from 11.0526 to 7.28846 for different functional sets of logic gates. The most efficient circuits in terms of the number of primitive logic gates used can be evolved using $\mathbb{FS}_4$. The criteria $\overline{F^{bf}_{4DMOS}}$ ($F_2 = F_{4DMOS}$) alters from 57.85 to 73.3 in the case of using the functional sets $\mathbb{FS}_1, -, \mathbb{FS}_6$. The experimental results demonstrate that the most efficient dynamic MOS circuits optimised by the number of transistors can be evolved using $\mathbb{FS}_6$. $\overline{F_{4NMOS}}$ acquired for the case of $\mathcal{F}_2 = F_{4NMOS}$ changes from 42.18 to 53.22. The results show that the most efficient NMOS circuits optimised by the number of transistors can be produced using $\mathbb{FS}_6$. The same conclusion can be made about evolving the CMOS circuits.

So, the most efficient logic circuits optimised by the number of transistors can be produced using the functional set of logic gates $\mathbb{FS}_6$. This confirms that the 'favourable' functional set is the same for the circuits implemented using different

Table 3.7: Initial data: Evolving logic functions optimised by different criteria. Functional set $\mathbb{FS}_j$ is encoded according to the Table 2.2

| **Circuit, *.pla file** | **mult2** |
|---|---|
| **EHW parameters** | |
| **Circuit layout,** $N_{cols} \times N_{rows}$ | $1 \times 10$ |
| **Connectivity parameter,** $N_{connect}$ | 10 |
| **Functional sets,** | |
| $\mathbb{FS}_1$ : | $\{2, 6, 7, 8, 9, 10, 11, 21, 22, 23, 24\}$ |
| $\mathbb{FS}_2$ : | $\{6, 10, 11, 21, 22, 23, 24\}$ |
| $\mathbb{FS}_3$ : | $\{6, 7, 8, 9, 10\}$ |
| $\mathbb{FS}_4$ : | $\{4, 6, 10\}$ |
| $\mathbb{FS}_5$ : | $\{6, 7, 10\}$ |
| $\mathbb{FS}_6$ : | $\{4, 6, 10, 12, 23, 24\}$ |
| **Function representation** | Truth table |
| **Gate distribution** | Proportional |
| **Type of layout** | Fixed |
| **Type of building blocks** | Two-Input One-Output |
| $N_{in}^{max}(\mathcal{B})$ | 2 |
| **EA parameters** | |
| **Type of EA** | a rudimentary $(1 + \lambda)$ ES |
| **Number of generations** | 5000 |
| **Population size** | 5 |
| **Number of algorithm runs** | 100 |
| **Circuit mutation rate** | 0.05 |
| **Fitness strategy** | Dynamic fitness function, $\mathcal{F}_1 + \mathcal{F}_2$ |
| **Criteria for optimisation** | Circuit functionality<br># FETs in CMOS circuit<br># FETs in NMOS circuit<br># FETs in PMOS circuit<br># FETs in dynamic CMOS circuit |

Table 3.8: Experimental results: Dynamic fitness function strategy using different optimisation parameters at the second stage of evolution. $\mathcal{F}_1$ and $\mathcal{F}_2$ are the first and second criteria of the dynamic fitness function $\mathcal{F}$, ($\mathcal{F}_1 = F_1$); $F_{4CMOS}$, $F_{4DMOS}$ and $F_{4NMOS}$ are the number of transistors in CMOS, dynamic MOS and NMOS circuits respectively; $\overline{F_k^{bf}}$ is the mean value of criteria $F_k$ for the best evolved chromosomes; $N_f$ is the fully functional circuit; $\overline{F_2(\mathcal{N}_f)}$ is the mean fitness function $F_2$ of $N_f$ evolved during 100 runs; $\mathcal{R}(\mathcal{N}_f)$ is the number of evolved fully functional circuits, $\mathcal{N}_f$.

| $\mathcal{F}_2$ | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}$ | $\overline{F_2(\mathcal{N}_f)}$ | $\overline{F_{4CMOS}^{bf}}$ | $\overline{F_{4DMOS}^{bf}}$ | $\overline{F_{4NMOS}^{bf}}$ | $\mathcal{R}(\mathcal{N}_f)$ |
|---|---|---|---|---|---|---|---|
| Functional set: $\mathbb{FS}_1 : \{2,6,7,8,9,10,11,21,22,23,24\}$ | | | | | | | |
| $F_2$ | 97.5156 | 12.04 | **11.0526** | 66.08 | 77.3 | 55.17 | 38 |
| $F_{4CMOS}$ | 97.4531 | 12.64 | 12.1111 | **63.76** | 74.48 | 53.18 | 36 |
| $F_{4DMOS}$ | 97.8438 | 12.2 | 11.6585 | 61.62 | **71.89** | 51.35 | 41 |
| $F_{4NMOS}$ | 97.2344 | 12.55 | 11.4706 | 63.86 | 74.51 | **53.22** | 34 |
| Functional set: $\mathbb{FS}_2 : \{6,10,11,21,22,23,24\}$ | | | | | | | |
| $F_2$ | 97.7969 | 10.83 | **10.1707** | 64.62 | 73.87 | 53.09 | 41 |
| $F_{4CMOS}$ | 98.0469 | 11.15 | 11.05 | **59.8** | 68.08 | 48.98 | 40 |
| $F_{4DMOS}$ | 98.2969 | 10.76 | 10.5641 | 61.54 | **70.27** | 50.52 | 39 |
| $F_{4NMOS}$ | 98 | 10.96 | 10.6047 | 60.38 | 68.81 | **49.5** | 43 |
| Functional set: $\mathbb{FS}_3 : \{6,7,8,9,10\}$ | | | | | | | |
| $F_2$ | 98.3125 | 9.44 | **8.40351** | 65.68 | 76.78 | 54.81 | 57 |
| $F_{4CMOS}$ | 98.3438 | 9.85 | 8.71429 | **63.6** | 74.78 | 53.29 | 49 |
| $F_{4DMOS}$ | 98.4844 | 9.86 | 9 | 62.2 | **73.3** | 52.2 | 56 |
| $F_{4NMOS}$ | 98.2031 | 9.69 | 9.15385 | 62.06 | 73.01 | **52.02** | 52 |
| Functional set: $\mathbb{FS}_4 : \{4,6,10\}$ | | | | | | | |
| $F_2$ | 98.7812 | 7.26 | **7.28846** | 59.4 | 68.3 | 49 | 52 |
| $F_{4CMOS}$ | 98.625 | 7.3 | 7.37255 | **57.38** | 66.15 | 47.42 | 51 |
| $F_{4DMOS}$ | 98.5781 | 7.41 | 7.46667 | 58.7 | **67.65** | 48.5 | 45 |
| $F_{4NMOS}$ | 98.3594 | 7.42 | 7.42857 | 58.4 | 67.32 | **48.26** | 42 |
| Functional set: $\mathbb{FS}_5 : \{4,6,10,12,23,24\}$ | | | | | | | |
| $F_2$ | 99.2812 | 7.87 | **7.63636** | 61.62 | 71.19 | 51 | 77 |
| $F_{4CMOS}$ | 98.9844 | 8.26 | 8.08955 | **58.18** | 67.71 | 48.4 | 67 |
| $F_{4DMOS}$ | 99.2656 | 8.38 | 8.25333 | 59.2 | **68.92** | 49.26 | 75 |
| $F_{4NMOS}$ | 99.0469 | 8.5 | 8.22857 | 60.08 | 69.94 | **49.99** | 70 |
| Functional set: $\mathbb{FS}_6 : \{4,6,10,12,23,24\}$ | | | | | | | |
| $F_2$ | 98.0569 | 9.6 | **9.55263** | 54.48 | 61.86 | 44.55 | 38 |
| $F_{4CMOS}$ | 97.9844 | 9.92 | 9.90698 | **53.2** | 60.24 | 43.42 | 43 |
| $F_{4DMOS}$ | 98 | 9.76 | 9.81818 | 51.1 | **57.85** | 41.7 | 44 |
| $F_{4NMOS}$ | 97.9531 | 9.58 | 9.86047 | 51.6 | 58.56 | **42.18** | 43 |

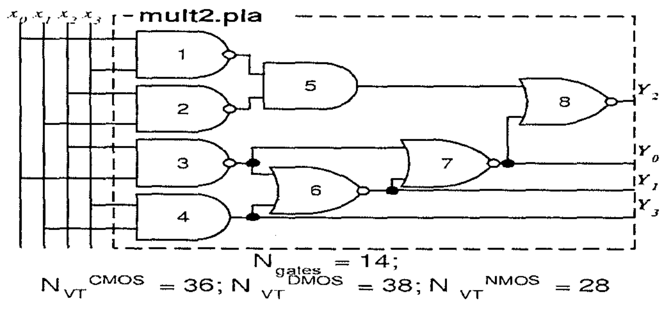


Figure 3.5: Most efficient evolved two-bit multiplier design optimised by the number of used transistors (A): Functional set: $\mathbb{FS}_1$ :{2, 6, 7, 8, 9, 10, 11, 21, 22, 23, 24}

technologies and optimised by the number of transistors. Nevertheless, the most efficient logic circuits optimised by the number of primitive active logic gates can be evolved using functional set $\mathbb{FS}_4$. It is clear that the algorithm chooses different 'favourable' functional sets for the circuits optimised by the number of transistors and the number of primitive logic gates.

It is necessary to note that the quality of evolved circuits optimised by the criteria $F_{4CMOS}$, $F_{4NMOS}$ and $F_{4DMOS}$ is nearly the same. For example, the criterion $\overline{F^{bf}_{4CMOS}}$ obtained with $\mathbb{FS}_3$ changes from 62.06 to 63.6 for optimisation criteria $F_{4DMOS}$, $F_{4CMOS}$ and $F_{4NMOS}$. That is not a big difference in comparison with $\overline{F^{bf}_{4CMOS}} = 65.68$ obtained during optimisation by $F_2$ criterion. Similar results have been obtained for other functional sets of logic gates and for optimisation criteria $F_{4NMOS}$ and $F_{4DMOS}$. This shows that there is no need to perform separate optimisation by the number of transistors for different implementation technologies. Note, that this is not extended to another circuit parameters that can be optimised (the circuit delay, the area used, etc.).

Next we will look at the structures of circuits optimised by the number of transistors and compare them with the circuits optimised by the number of primitive active logic gates.

The most efficient evolved two-bit multiplier design optimised by the number of used transistors is depicted in Fig. 3.5 (design **A**). This design represents the most efficient evolved circuit that can be implemented using NMOS, PMOS and dynamic MOS technologies. The number of transistors is different for different circuit implementations. Thus, the CMOS circuit contains only 36 transistors, the NMOS circuit consists of 28 transistors and the dynamic MOS circuit needs 38 transistors. The design in question requires 14 primitive logic gates of four types (NOT, AND, OR and EXOR). Note that EXOR gates are not used in this circuit implementation. The design contains 4 levels. The number of levels in a circuit is defined by the maximum number of primitive logic gates that are connected to each other interactively. The first level of design shown in Fig. 3.5 contains 4 logic gates indexed 1, 2, 3 and 4. The second level of the circuit in question consists of 2 logic gates labeled 5 and 6. The third level of the circuit has only one logic gate indexed 7 and the last level includes one logic gate marked 8. It is obvious that the circuit delay depends on the number of levels in the circuit. Greater number of levels in the circuits mean slower circuits.

It has been reported in literature [26], [9] that the most efficient evolved two-bit multiplier optimised by the number of primitive logic gates used in the circuit requires only 7 logic gates. Therefore, the design given in Fig. 3.5 is not efficient in terms of the number of primitive active logic gates. It is necessary to note that it is impossible to evolve the efficient design with the minimal number of transistors and the minimal number of primitive active logic gates because the following statements contradict each other:

1. The most efficient gate implementations in terms of the number of transistors

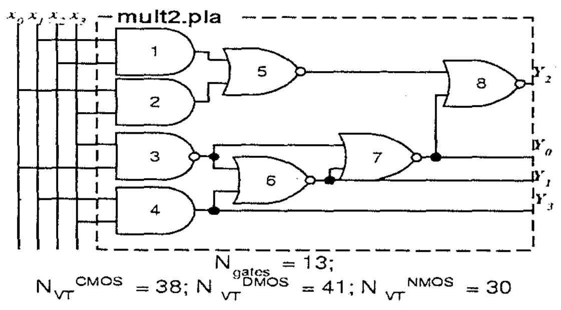


Figure 3.6: Evolved two-bit multiplier design (**B**): Functional set: $\mathbb{FS}_2$ :{6, 10, 11, 21, 22, 23, 24}

used are NAND and NOR. They require fewer transistors than AND and OR logic circuits produced using NMOS, CMOS or dynamic MOS technologies.

2. In our interpretation a primitive logic gate is a gate implementing the primitive logic operator such as NOT, OR, AND or EXOR. This means that the NAND and NOR gates contain 2 primitive logic gates: (AND, NOT) and (OR, NOT), respectively.

Therefore, reducing the number of active primitive logic gates leads to increasing the number of transistors used in circuit. This can be confirmed by two designs shown in Fig. 3.6 (design **B**) and Fig. 3.7 (design **C**).

It can be seen that the circuit given in Fig. 3.6 requires less number of primitive logic gates than the design depicted in Fig. 3.5. Also, the circuit design **B** contains more transistors than design **A**. Analysing the circuit structures we found out that the implementations of the circuit outputs $Y_0$, $Y_1$ and $Y_3$ are identical. The designs **A** and **B** are different by the implementation of the circuit output $Y_2$. The sub-circuit containing gates labeled 1, 2 and 5 in design **A** requires 5 primitive logic gates distinct from the fact that the similar sub-circuit in design **B** needs only 4 primitive

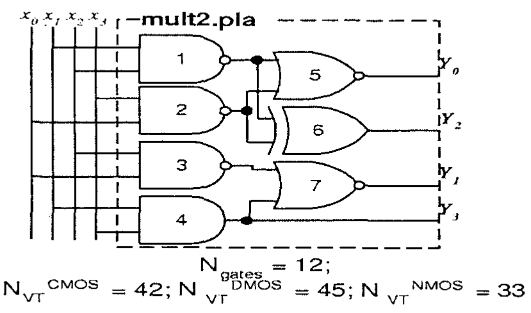


Figure 3.7: Most efficient evolved two-bit multiplier design optimised by the circuit delay(C): Functional set: $\mathbb{FS}_1$ :{2, 6, 7, 8, 9, 10, 11, 21, 22, 23, 24}

logic gates. Also, the circuit shown in Fig. 3.7 is faster than the circuits discussed above. The circuit design **C** is the most efficient evolved circuit in terms of minimal circuit delay. Nevertheless, this sub-circuit in design **A** contains less transistors than the similar sub-circuit in design **B**. This demonstrates that the size of evolved circuit can be altered by changing the sub-circuit structures.

The circuit design discussed above contains 4 levels and is distinct from design C shown in Fig. 3.7 in that it consists of only 2 levels. This design needs less primitive logic gates than designs **A** and **B** scrutinized above. At the same time, this design requires more transistors than designs **A** and **B**. The analysis of designs **A**, **B** and **C** shows that the efficient circuits have different configurations for different optimisation criteria. The designer implementing the MOS circuits would be more interested in efficient circuit implementation optimised by the circuit delay, the employed area, the compactness of connections, the number of transistors, etc. In this case designs **A** or **C** can be chosen as efficient. From another point of view, the designer of the FPGA circuits would be interested in minimisation of the number of primitive active logic gates or the number of any 'atomic' devices. Using different optimisation criteria we can satisfy the requirements of both designers. This is an evidence of universality of

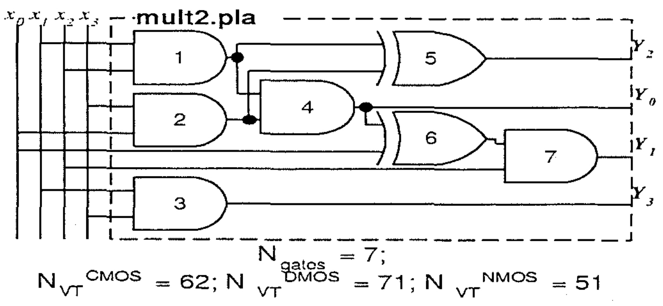


Figure 3.8: Evolved two-bit multiplier design (**D**): Functional set: $\mathbb{FS}_1$ :{2, 6, 7, 8, 9, 10, 11, 21, 22, 23, 24}

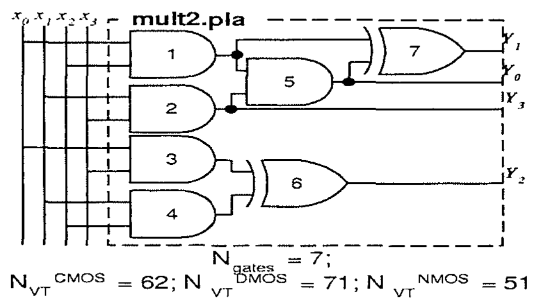


Figure 3.9: Evolved two-bit multiplier design (**E**): Functional set: $\mathbb{FS}_2$ :{6, 10, 11, 21, 22, 23, 24}

this method in terms of implementation technologies used.

Next we will compare the efficient circuits optimised by the number of transistors and the number of primitive logic gates. Fig. 3.8 (design **D**) and Fig. 3.9 (design **E**) depict the best evolved two-bit multipliers optimised by the number of primitive logic gates used. Both of these circuits require 62 transistors to be implemented using CMOS technology or 51 transistors to be produced using NMOS technology or 71 transistors to be performed using dynamic MOS technology. Obviously, these are not the most efficient designs optimised by the number of transistors used. Both of these circuits contain 5 AND and 2 EXOR gates connected differently. It is necessary to note that *all* evolved efficient two-bit multipliers optimised by the number of primitive logic gates contain the same number of transistors. This means that *all* 7-gate circuits for the two-bit multiplier have an identical 'atomic' structure. Note that, during an experiment discussed in Section 3.4.7, 1173 fully functional solutions have been analysed. Among them, 183 solutions have 7 primitive active logic gates. The number of evolved 7-gate logic circuits is relatively small because in most experiments the optimisation by the number of transistors used has been applied and therefore, the number of primitive logic gates in evolved designs has been relatively high.

The logic gates are connected differently inside the 7-gate two-bit multiplier circuits. Let us consider closely the circuit structures of the 7-gate two-bit multipliers shown in Fig. 3.8 (design **D**) and Fig. 3.9 (design **E**). Both designs have two independent sub-circuits. In circuit **D** one involves $Y_0$, $Y_1$ and $Y_2$ and the other, $Y_3$. This circuit structure is very similar to one conventional efficient two-bit multiplier reported in [9]. In conventional design, the gate labeled 7 is EXOR and the gate marked 6 is AND, whereas in design **D** the 7-th gate is AND and the 6-th one is EXOR. Therefore, we can consider design **D** as an example of a conventional circuit implementing the two-bit multiplier. In circuit **E**, one sub-circuit includes $Y_0$, $Y_1$ and $Y_3$ and the other, $Y_2$. The last circuit reveals some strangeness, because it implements differently the output $Y_2$, that will never happen in conventional design because of multiplication principles. In the conventional model of multiplication one output $Y_0$ is re-used. The circuit outputs $Y_3$ and $Y_0$ are re-used in circuit **E**. $Y_1$ is produced by four gates in the circuit **E**, whereas it needs five in the conventional circuit (design **D**). The circuit for $Y_2$ in both designs is effectively the same. In the conventional circuit $Y_3$ has nothing to do with $Y_1$; in design **E**, $Y_3$ is used to produce $Y_1$. This shows that the 7-gate two-bit multiplier evolved does not always replicate the principles of

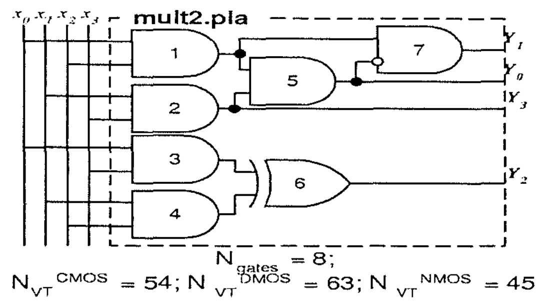


Figure 3.10: Evolved two-bit multiplier design (F): Functional set: $\mathbb{FS}_3$ :{6, 7, 8, 9, 10}

multiplication.

Since no circuit designs evolved with the minimal number of transistors have been previously reported in the literature, the comparison of evolved circuits with another similar approaches is impossible.

Next, we will look at 293 8-gate two-bit multipliers evolved during the experiments described in Section 3.4.7. Examining these circuits reveals that the number of transistors in *all* circuits evolved is *constant*. Thus, the CMOS circuits contain only 54 transistors, the NMOS circuits consist of 45 transistors and the dynamic MOS circuits need 63 transistors. The evolved logic circuits have different structures with different number of primitive logic gates involved. One of the evolved 8-gate two-bit multipliers is illustrated in Fig. 3.10 (design **F**). This circuit is more efficient in terms of the number of transistors used than the similar one with 7 primitive logic gates (Fig. 3.8, Fig. 3.9). Design **F** contains 7 logic gates and 8 primitive logic gates. The structure of the circuit is very similar to the one shown in Fig. 3.9. The difference is in gate labeled 7. This gate is EXOR gate in design **E** and AND gate with one inverted input in design **F**. Because of inversion the circuit in question requires more primitive logic gates. The analysis of this structure reveals that some of evolved circuits have

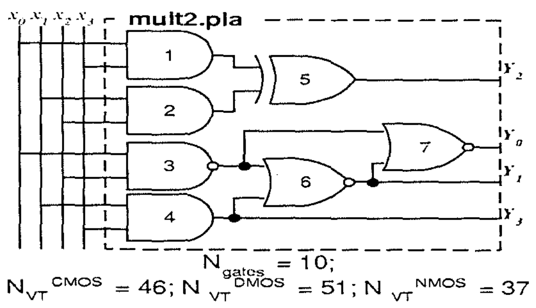


Figure 3.11: Evolved two-bit multiplier design (G): Functional set: $\mathbb{FS}_6$ :{4, 6, 10, 12, 23, 24}

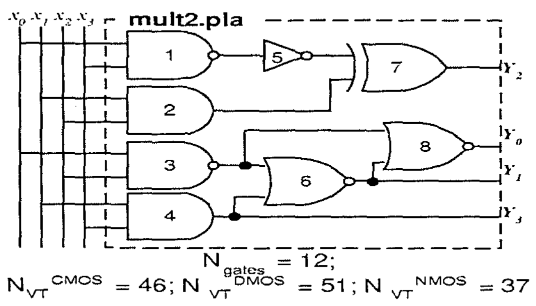


Figure 3.12: Evolved two-bit multiplier design (I): Functional set: $\mathbb{FS}_6$ :{4, 6, 10, 12, 23, 24}

the same connection between gates but involve different logic gates.

Introducing such new optimisation criteria as the number of transistors used in the fully functional circuits evolved brings a new aspect of circuit structure analysis. We consider the evolved circuit not only in terms of the number of primitive logic gates, but also compare the evolved circuits using other optimisation criteria. We discover that the circuits with the same number of transistors required in NMOS, CMOS and dynamic MOS circuits can contain different number of primitive logic gates. This can be illustrated by the circuits shown in Fig. 3.11 – Fig. 3.13. The CMOS implementation of these circuits contain only 46 transistors, the NMOS circuits consist of 37 transistors and the dynamic MOS circuits need 51 transistors. The circuit given

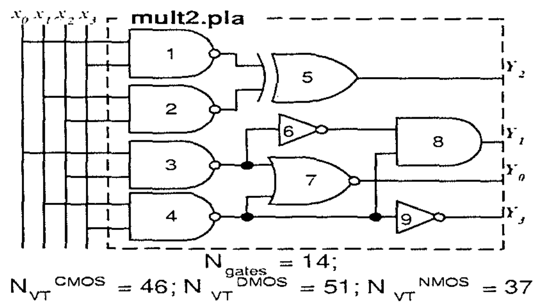


Figure 3.13: Evolved two-bit multiplier design (**J**): Functional set: $\mathbb{FS}_6$ :{4, 6, 10, 12, 23, 24}

in Fig. 3.11 (design **G**) requires 10 primitive logic gates, the circuit shown in Fig. 3.12 (design **I**) - 12 and the circuit presented in Fig. 3.13 (design **J**) - 14 primitive logic gates. The designs **G** and **I** differs by logic gate labeled 1, and therefore one more inversion is added in design **I**. The design **I** can be easily reduced to the circuit shown in Fig. 3.11, using the double inversion rule. Another two inversions are added to the circuit shown in Fig. 3.13 in comparison with the design **G**. In this case the circuits first differ by the logic gates marked 2 and 4. The size of all circuits in terms of the number of transistors is the same because adding an inversion gate can be compensated by changing the type of logic gate. Thus in the case of designs **G** and **I**, we changed the type of gate marked 1 and added one more inverter. The cost of this sub-circuit is $(cost(NAND) + cost(NOT))$, that equals to $cost(AND)$ for any implementation technologies, where $cost(NAND)$, $cost(NOT)$ and $cost(AND)$ are the cost of logic gates NAND, NOT and AND in terms of the number of required transistors, respectively. Note that according to the design rule of basic logic gates $cost(AND) = cost(NAND) + cost(NOT)$. Therefore, the cost of circuit does not alter. The same analysis can be applied to designs **G** and **J** or designs **I** and **J**.

So, we can conclude that comparison of logic circuits by the number of transistors used does not allows us to find if the compared circuits have some logical reduction or not.

## 3.5 Evolutionary processes specified by dynamic fitness function and their analysis.

The objective of the work reported in this section is to investigate how the circuit evolution is carried out. This is interesting thing to do for three main reasons. Firstly, to investigate what type of genes have most influence on the algorithm performance. Secondly, to see how effective an allocation of active logic gates might be in a digital circuit design task. And thirdly, to define the difference between two evolutionary processes specified by dynamic fitness function. In order to achieve these goal we investigate the genotypes of the best chromosomes, which bring some improvements in the evolution process.

### 3.5.1 Analysing the evolved circuits using a probabilistic approach.

In this section we will present a probabilistic approach, that define how the different types of genes located differently participate in the evolutionary process. The analysis is carried out under the collection of differential chromosomes. A *differential chromosome* shows the dissimilarity between genotypes of two chromosomes encoding the circuit structure. Phenotype of this chromosome defines the difference between phenotypes of two compared chromosomes. A number of events are generated to expound the conditions that are applied to produce the differential chromosome. The conditional probabilities of these events are calculated. The analysis of experimental results is based on the examination of the conditional probabilities calculated for each type of gene and its location. The proposed approach is scrutinized in the following sub-sections.

**Genotype of a differential chromosome.**

The genotype of *differential chromosome*, $\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}$ reflects the difference between two genotypes of chromosomes $\mathbb{C}_1$ and $\mathbb{C}_2$. Chromosome $\mathbb{C}_i$ encodes the circuit structure. The genotype of chromosome $\mathbb{C}_i$ contains the rectangular array of logic gates and the set of circuit output connections. Each logic gate is described by the set of connectivity genes and the functionality gene [1].

The genotype of differential chromosome has the equivalent structure to the genotype of chromosome $\mathbb{C}_i$ and consists of the same types of genes. The chromosome $\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}$ has identical rectangular structure to the chromosomes $\mathbb{C}_1$ and $\mathbb{C}_2$. Each gate contains functionality and connectivity genes, which define the dissimilarity between the corresponding genes in the chromosomes $\mathbb{C}_1$ and $\mathbb{C}_2$. The functionality genes in the gate genotype define the functional difference between functionality genes of the considering gate in the compared chromosomes. Let us consider the $i$-th logic gate in $\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}$. The functional gene of this gate defines the dissimilarity between the $i$-th functional gene in the gate in $\mathbb{C}_1$ and $\mathbb{C}_2$. A circuit output gene represents the number of circuit genes which are not equal in chromosomes $\mathbb{C}_1$ and $\mathbb{C}_2$. The functional set represents any primitive logic gates AND, OR, EXOR, NOT with inverted and primary inputs. In order to receive a more accurate analysis of the circuit evolution, the "two-gene" interpretation of gate functionality is introduced as follows : $< d_{gt} \ d_{it} >$, where $d_{gt}$ is the primitive gate type, $d_{gt} \in \{\text{AND, OR, NOT, EXOR}\}$ and the $d_{it}$ defines the number of different inputs (primitive or inverted) used in the logic cell. Thus, the gate $\mathrm{c}_i$ in $\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}$ can be described by three genes: $\mathrm{c}_i = \{d_{gt} \ d_{it} \ i_d\}$, where

$d_{gt}$ defines whatever the primitive type of the $i$-th gate is the same or not for the chromosomes $\mathbb{C}_1$ and $\mathbb{C}_2$; $d_{it}$ determines the number of different input types in the $i$-th cell and $i_d$ is the number of different uncommitted connections. The gate genotype contains an uncommitted connection or input, if the logic function describing the behaviour of logic cell does not depend on this input. Note, that if all genes of the differential chromosome are 0, then chromosomes $\mathbb{C}_1$ and $\mathbb{C}_2$ are equivalent and describe the same circuit.

**Phenotype of differential chromosome.**

The fitness function of the differential chromosome is defined as follows:

$$\mathcal{F}_{\mathbb{D}} = \mathcal{F}_{\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}} \begin{cases} \mathcal{F}_{\mathbb{C}_2} - \mathcal{F}_{\mathbb{C}_1}, & F_1(\mathbb{C}_1) < 100, F_1(\mathbb{C}_2) < 100; \\ 100 + \mathcal{F}_{\mathbb{C}_1} + \mathcal{F}_{\mathbb{C}_2}, & F_1(\mathbb{C}_1) = 100, F_1(\mathbb{C}_2) = 100. \end{cases}$$

where $F_1(\mathbb{C}_i)$ defines the functionality of the circuit described by the chromosome $\mathbb{C}_i$. Chromosome $\mathbb{C}_1$ is generated first during evolutionary process followed by chromosome $\mathbb{C}_2$.

$\mathcal{F}_{\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}}$ reflects the phenotypic difference between $C_1$ and $C_2$ and defines the functionality of analysed circuits. If the chromosomes $\mathbb{C}_1$ and $\mathbb{C}_2$ do not implement the desired function completely, then these chromosomes participate in the $\Upsilon_{F_1}$ evolutionary process. Otherwise, the structures of $\mathbb{C}_1$ and $\mathbb{C}_2$ are generated during the $\Upsilon_{F_2}$ evolutionary process. $\mathcal{F}_{\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}} < 100$ defines that the compared chromosomes are not fully functional. $\mathcal{F}_{\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}} = 100$ means that the differential chromosome compares two fully functional circuits.

**An example.**

Let us consider the differential chromosome $\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}$ shown in Table 3.9. This chromosome defines the dissimilarity between chromosomes $\mathbb{C}_1$ and $\mathbb{C}_2$ (also given in Table

Table 3.9: An example of generating a differential chromosome. $\mathbb{C}_1$ and $\mathbb{C}_2$ are two compared chromosomes; $\mathbb{D}$ is the differential chromosome; $\mathbf{o} = \{o_1\ \ o_2\}$ is the circuit output vector; $\mathcal{F}_{\mathbb{C}_i}$ and $\mathcal{F}_{\mathbb{D}}$ are the fitness functions of chromosomes $\mathbb{C}_i$ and $\mathbb{D}$ respectively.

| $\mathbb{C}$ | **cell 0, $\mathbf{c}_0$** | **cell 1, $\mathbf{c}_1$** | **cell 2, $\mathbf{c}_2$** | **cell 3, $\mathbf{c}_3$** | $\mathbf{o}$ | |
|---|---|---|---|---|---|---|
| | $\{c_f^0\ i_0^0\ i_1^0\ i_2^0\}$ | $\{c_f^1\ i_0^1\ i_1^1\ i_2^1\}$ | $\{c_f^2\ i_0^2\ i_1^2\ i_2^2\}$ | $\{c_f^3\ i_0^3\ i_1^3\ i_2^3\}$ | $\{o_1\ o_2\}$ | $\mathcal{F}_{\mathbb{C}_i}$ |
| $\mathbb{C}_1$ | $\{22\ \mathbf{1}\ 2\ 0\}$ | $\{\mathbf{15}\ 2\ \mathbf{1}\ \mathbf{0}\}$ | $\{\mathbf{15}\ 3\ 4\ \mathbf{0}\}$ | $\{16\ 2\ 4\ \mathbf{0}\}$ | $\{5\ 6\}$ | 88.97 |
| $\mathbb{C}_2$ | $\{22\ \mathbf{0}\ 2\ 0\}$ | $\{\mathbf{9}\ 2\ \mathbf{0}\ \mathbf{1}\}$ | $\{\mathbf{6}\ 3\ 4\ \mathbf{1}\}$ | $\{16\ 2\ 4\ \mathbf{1}\}$ | $\{5\ 6\}$ | 93.03 |
| $\mathbb{D}$ | $\mathbf{d}_0$ | $\mathbf{d}_1$ | $\mathbf{d}_2$ | $\mathbf{d}_3$ | $\mathbf{o}_d$ | |
| | $\{d_{gt}^0\ d_{it}^0\ i_d^0\}$ | $\{d_{gt}^1\ d_{it}^1\ i_d^1\}$ | $\{d_{gt}^2\ d_{it}^2\ i_d^2\}$ | $\{d_{gt}^3\ d_{it}^3\ i_d^3\}$ | $\{o_{d1}\ o_{d2}\}$ | $\mathcal{F}_{\mathbb{D}}$ |
| $\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}$ | $\{0\ 0\ 1\}$ | $\{1\ 0\ 1\}$ | $\{1\ 2\ 0\}$ | $\{0\ 0\ 1\}$ | $\{0\ 0\}$ | 4.06 |

3.9). The unequal genes in the chromosomes $\mathbb{C}_1$ and $\mathbb{C}_2$ are shown in bold. Let us consider cell $\mathbf{c}_3(\mathbb{C}_1) = \{16\ \ 2\ \ 4\ \ 0\}$ and $\mathbf{c}_3(\mathbb{C}_2) = \{16\ \ 2\ \ 4\ \ 1\}$. The gate genotypes are non-identical by gene $i_2^3$. The functional gene $c_f^3$ corresponds to a multiplexer, which involves 3 inputs. Thus, the genotype of cell 3 in differential chromosome is $\mathbf{c}_3(\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}) = d_3 = \{0\ \ 0\ \ 1\}$. Now let us consider cell 1. The genotype of cells $\mathbf{c}_1(\mathbb{C}_1)$ and $\mathbf{c}_1(\mathbb{C}_2)$ are different by genes $c_f^1$, $i_1^1$ and $i_2^1$. The cell $\mathbf{c}_1(\mathbb{C}_1)$ implements the logic function $\overline{i_0} \vee \overline{i_1}$ and the cell $\mathbf{c}_1(\mathbb{C}_2)$ represents the logic function $\overline{i_0} \wedge \overline{i_1}$ (Table 2.2). Both logic functions employ inverted inputs $\overline{i_0}, \overline{i_1}$, but the primitive logic operator is different. Therefore, we can define genes $d_{it}^1 = 0$ and $d_{gt}^1 = 1$ in $\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}$ for cell 1. Note that the input $i_2^1$ in this case is redundant. The number of different connections is defined as follows. The number of unequal connections is two, but gene $i_2$ is redundant, therefore $i_d^1 = 1$. The logic functions describing the behaviour of cell 2 is $\overline{i_0} \vee \overline{i_1}$ in $\mathbb{C}_1$ and $i_0 \wedge i_1$ in $\mathbb{C}_2$. In this case the primitive operator has been changed from AND to OR and the inputs involved change their type from $\overline{i_0^2}$ to $i_0^2$ and from $\overline{i_1^2}$ to $i_1^2$. Therefore, the genotype for this gate is $\{1\ \ 2\ \ 0\}$. A similar analysis has been carried out to define the genotypes of the rest of the cells in the differential chromosome $\mathbb{D}_{\mathbb{C}_1\mathbb{C}_2}$.

**Probabilistic analysis.**

The issue of this work is to define how genes influence the evolution process. Therefore, the differential chromosomes with $\mathcal{F}_{\mathbb{D}} \neq 0$ are considered. The probabilistic analysis is based on the analysis of differential chromosome genotypes. The differential chromosomes analysed have been calculated based on ordinary chromosomes involved in a successful evolution. Successful evolution produces fully functional design.

In order to explain how the analysis is carried out, the following notations have been adopted. Let us define the outcomes of experiment $\xi_M$ with functional set $M$ to be the genotype and phenotype of the differential chromosomes. The sample space $\Omega$ associated with an experiment $\xi_M$ is the collection of all possible phenotypes and genotypes of differential chromosome in $\xi_M$. The intersection of events $E_1$ and $E_2$, written as $E_1 \cdot E_2$, is defined as the set of outcomes which belong to both $E_1$ and $E_2$. Given two subsets of $\Omega$, say $E_1$, $E_2$, the union of $E_1$, $E_2$, written as $E_1 \cup E_2$, is defined as the set of outcomes which belong to either $E_1$ or $E_2$ or both. The circuit evolutionary process $\Upsilon$ carried out during algorithm execution contains two evolution sub-processes:

1. Evolution of the circuit functionality, $\Upsilon_{\mathcal{F}_1}$;

2. Evolution aimed at improving the quality of evolved fully functional circuit, $\Upsilon_{\mathcal{F}_2}$.

Note that these two processes can not be performed at the same time. The first process is carried out when the functionality of the best chromosomes is less then 100% and the second process is performed if the evolutionary strategy (ES) tends to improve the quality of evolved circuits. In this piece of work, the quality of evolved circuit is defined by the number of active logic gates in the circuit. According to the representation of differential chromosome, there are 4 types of genes which could influence the ES performance: (1) the cell type gene; (2) the input cell type gene; (3) the connection gene; (4) the circuit output gene. So, we can define the following events, which could be associated with an experiment $\xi_M$:

$E_0$ the fitness function of differential chromosome is greater than 0, $\mathcal{F}_{\mathbb{D}} > 0$;

$E_1$ the functionality fitness function $\mathcal{F}_{1d}$ is less than 100%, $\mathcal{F}_{1d} < 100.0$ (i.e. the circuit functionality evolution $\Upsilon_{\mathcal{F}_1}$ is considered);

$E_2^j$ the cell type gene $d_{gt}$ located in the $j$-th cell is greater than 0;

$E_3^j$ the input cell type gene $d_{it}$ located in the $j$-th cell is greater than 0;

$E_4^j$ the connection gene $i_d$ located in the $j$-th cell is greater than 0;

$E_5^k$ the circuit output gene $o_d$ located in the $k$-th circuit output is greater than 0.

The event $E_0$ defines that the differential chromosome is calculated using two chromosomes with different fitness functions. Event $E_1$ shows that the compared chromosomes are not fully functional. This means that the compared chromosomes have been involved in evolutionary process $\Upsilon_{\mathcal{F}_1}$. Note that $\overline{E_1}$ defines that the functionality fitness function $F_{1d}$ of differential chromosome is greater than or equal to 100.0. So, the event $\overline{E_1}$ agrees with the case when the differential chromosome compare the fully functional circuits. In other words, the compared chromosomes have participated in the evolutionary process $\Upsilon_{\mathcal{F}_2}$. The events $E_2^j - E_4^j$ and $E_5^k$ define how the compared chromosomes are different.

The probabilities of the events $E_1 \cdot E_0$ and $\overline{E_1} \cdot E_0$ can be defined as

$$p(E_1 \cdot E_0) = \frac{N_{E_0 E_1}}{N_{gen} * R(\mathcal{N})}; \tag{3.5.1}$$

$$p(\overline{E_1} \cdot E_0) = \frac{N_{E_0 \overline{E_1}}}{N_{gen} * R(\mathcal{N})} \tag{3.5.2}$$

where $N_{E_0 E_1}$ is the number of differential chromosomes with $\mathcal{F}_{\mathbb{D}} \neq 0$ and $\mathcal{F}_{1d} \neq 100.0$ (i.e. defines the execution of process $\Upsilon_{\mathcal{F}_1}$); $N_{E_0 \overline{E_1}}$ is the number of differential chromosomes with $\mathcal{F}_{\mathbb{D}} \neq 0$ and $\mathcal{F}_{1d} = 100.0$ (i.e. defines the execution of process $\Upsilon_{\mathcal{F}_2}$), $\mathcal{N}$ is the final evolved network or circuit; $R(\mathcal{N})$ is the number of algorithm runs; $N_{gen}$ is the number of generations. The conditional probability of $E_2^j$, given that $E_1$ has occurred, is defined as

$$p(E_2^j | (E_0 \cdot E_1)) = \frac{p(E_0 \cdot E_1 \cdot E_2^j)}{p(E_0 \cdot E_1)} = \frac{{}_jN_{E_0 E_1}^{gt}}{N_{E_0 E_1}} \tag{3.5.3}$$

where ${}_jN_{E_0 E_1}^{gt}$ is the number of times $E_0$, $E_1$ and $E_2$ occurred (i.e. the number of non-zero cell type genes $d_{gt}^j$ in differential chromosomes with $F_{1d} < 100$ and $\mathcal{F}_{\mathbb{D}} \neq 0$). The conditional probabilities of $E_3^j$, $E_4^j$ and $E_5^k$, given that $E_0 \cdot E_1$ has occurred, are calculated analogously to Eq. 3.5.3:

$$p(E_3^j | (E_0 \cdot E_1)) = \frac{{}_jN_{E_0 E_1}^{it}}{N_{E_0 E_1}}; \tag{3.5.4}$$

$$p(E_4^j | (E_0 \cdot E_1)) = \frac{{}_jN_{E_0 E_1}^{c}}{N_{E_0 E_1}}; \tag{3.5.5}$$

$$p(E_5^k | (E_0 \cdot E_1)) = \frac{{}_kN_{E_0 E_1}^{o}}{N_{E_0 E_1}}; \tag{3.5.6}$$

where ${}_jN_{E_0 E_1}^{it}$ and ${}_jN_{E_0 E_1}^{c}$ are the number of non-zero input cell and connection genes of the $j$-th cell in the differential chromosomes with functionality fitness function $F_{1d} < 100$ and fitness function $\mathcal{F}_{\mathbb{D}} \neq 0$ respectively; ${}_kN_{E_0 E_1}^{o}$ is the number of non-zero circuit output genes located in the $k$-th position in differential chromosomes with

$\mathcal{F}_{1d} < 100$ and $\mathcal{F}_{\mathbb{D}} \neq 0$. The conditional probabilities calculated in Eq. 3.5.3 – Eq. 3.5.6 correspond to the evolutionary process $\Upsilon_{\mathcal{F}_1}$, such that the functionality of the circuit is evolved.

The conditional probabilities shown below correspond to the evolutionary process $\Upsilon_{\mathcal{F}_2}$, which forces the improvement of the functional circuit in terms of the number of active gates used.

$$p(E_2^j|(E_0 \cdot \overline{E_1})) = \frac{{}_jN_{\mathcal{F}_1}^{gt}}{N_{E_0\overline{E_1}}}; \tag{3.5.7}$$

$$p(E_3^j|(E_0 \cdot \overline{E_1})) = \frac{{}_jN_{\mathcal{F}_1}^{it}}{N_{E_0\overline{E_1}}}; \tag{3.5.8}$$

$$p(E_4^j|(E_0 \cdot \overline{E_1})) = \frac{{}_jN_{\mathcal{F}_1}^{c}}{N_{E_0\overline{E_1}} * N_{in}^{max}}; \tag{3.5.9}$$

$$p(E_5^k|(E_0 \cdot \overline{E_1})) = \frac{{}_kN_{\mathcal{F}_1}^{o}}{N_{E_0\overline{E_1}} * N_{out}}. \tag{3.5.10}$$

For example, let us compute the conditional probability $p(E_4^5|(E_0 \cdot \overline{E_1}))$ with $N_{in}^{max} = 4$, $N_{gen} = 5000$, $R(\mathcal{N}) = 100$. In this case, the average number $N_{E_0\overline{E_1}}$ of differential chromosomes with $\mathcal{F}_{\mathbb{D}} \neq 0$ and $\mathcal{F}_{1d} = 100.0$ per one successful run is 30. There are 200 non-zero connection genes in the differential chromosomes in question that are located in the 5-th logic cell. Then the conditional probability can be calculated as follows: $p(E_4^5|(E_0 \cdot \overline{E_1})) = 200/(100 * 30 * 4) = 0.0167$.

The conditional probabilities calculated above define the probability with the genes influencing positively the evolution process. In other words, these genes belong to the chromosome that have just changed fitness value and the whole evolutionary process has been successful. This means that as a result of the evolution process, a fully functional solution has been evolved.

### 3.5.2 Experimental results.

In this section we will consider some experimental results obtained for two-bit multiplier (mult2.pla) and two-bit adder with carry (add2c.pla). The main idea of these experiments is to define how diverse types of genes located differently influence on successful ES performance. The analysis of obtained data has been performed in two stages. Firstly, the dependence of ES performance on the functional set of logic gate has been defined. Secondly, the influence of gene type and its location on the ES performance have been investigated.

The initial data for the experiment are given in Table 3.10. Any type of gene in the chromosome genotype was allowed to change with constant mutation probability $p_m$. The chosen functional sets of logic gates contain a specific subset of the primitive logic functions. For example, all functional sets involved in evolution of the two-bit multiplier have OR and EXOR primitive logic gates. {OR, EXOR, AND} has been used during evolution of the two-bit adder. Let $p_t(\{\textbf{and}\})$ be the percentage of the primitive logic gate AND in the functional set. $p_t(\{\textbf{or}\})$, $p_t(\{\textbf{not}\})$ and $p_t(\{\textbf{exor}\})$ are defined analogously. Let $p_i(\{\textbf{not}\})$ be the percentage of the inverted inputs in the functional set. Let us consider the computation process of the percentages mentioned above using the following example. Let us consider the functional set $M5 = \{6, 7, 10\}$ that can be interpreted as follows: $M5 = \{i_0 \wedge i_1, i_0 \wedge \overline{i_1}, i_0 \oplus i_1\}$. There are 3 types of logic gates. 6 and 7 encode the AND logic gate with and without using the inverted inputs. 10 interprets EXOR gate with primary inputs. Therefore, $p_t(\{\textbf{and}\}) = 2/3 = 0.66$ and $p_t(\{\textbf{exor}\}) = 1/3 = 0.33$. No logic gates OR and NOT are in the examined functional set. Hence, $p_t(\{\textbf{not}\}) = 0$ and $p_t(\{\textbf{or}\}) = 0$. All logic functions in $M5$ are two-input. So, 6 inputs in these functional circuits can be considered. There is only one logic function labeled 7 which contain an inverted input. Therefore, $p_i(\{\textbf{not}\}) = 1/6 = 0.16$. $p_t$ and $p_i$ describe the functional characteristics of the functional set. Table 3.11 summarises these characteristics for the functional sets used in this work. Note that all functional sets have different number of logic gates.

Two strategies have been applied in order to choose the functional set: (1) $p_t(\{\textbf{and}\})$ and $p_t(\{\textbf{exor}\})$ are not equal; (2) $p_t(\{\textbf{and}\})$, $p_t(\{\textbf{or}\})$ and $p_t(\{\textbf{exor}\})$ are equal. The two-bit multiplier has been evolved using functional sets $M1 - M5$ generated according to the first strategy. Thus, $p_t(\{\textbf{and}\}) = 0.8$ and $p_t(\{\textbf{exor}\}) = 0.2$ for the functional set $M3$ (Table 3.11)). The two-bit adders have been synthesised using the functional sets produced according to the second strategy. For instance, the functional set $A5$ used to evolve the two-bit adder has the following characteristics: $p_t(\{\textbf{and}\}) = p_t(\{\textbf{or}\}) = p_t(\{\textbf{exor}\}) = 0.33$ (Table 3.11).

Analysing the experimental results shown in Table 3.12, we can conclude that the ES performance depends on the proportion of primitive logic cells used in evolution. Thus, the functional sets $M4$ and $M5$ have the same set of primitive logic cells, but in terms of performance the better results have been achieved with functional set $M5$. Similar results have been obtained for the two-bit adder. The ES performs better with the smaller percentage of inverted inputs involved. It is interesting to note, that the ES performance deteriorated when the primitive logic operations were employed instead of using inverted inputs.

In order to define how the different types of genes and their location influence the EA performance, the differential chromosomes have been generated and the conditional probabilities mentioned in the previous section have been calculated as follows:

Table 3.10: Initial data Probabilistic analysis. $N_{in}^{max}(\mathcal{B})$ is the maximum number of inputs in the building block $\mathcal{B}$.

| **Circuit, *.pla file** | **mult2** | **add2c** |
|---|---|---|
| | **EHW parameters** | |
| **Circuit layout,** $N_{cols} \times N_{rows}$ | $1 \times 10$ | $1 \times 15$ |
| **Connectivity parameter,** $N_{connect}$ | 10 | 15 |
| **Functional set,** $\mathbb{FS}$ | $M1$, $M2$, $M3$, $M4$, $M5$ | $A1$, $A2$, $A3$, $A4$ |
| **Function representation** | Truth table (Boolean logic) | |
| **Gate distribution** | Proportional | |
| **Type of layout** | Fixed | |
| **Type of building blocks** | Two-Input One-Output | |
| $N_{in}^{max}(\mathcal{B})$ | 4 | |
| | **EA parameters** | |
| **Type of EA** | a rudimentary $(1+\lambda)$ ES | |
| **Number of generations,** $N_{gen}$ | 5000 | 15000 |
| **Population size,** $\lambda$ | 5 | |
| **Number of EA runs** | 100 | |
| **Circuit mutation rate,** $p_{cm}$ | 0.05 | |
| **Fitness strategy** | Dynamic fitness function, $\mathcal{F}_1 + \mathcal{F}_2$ | |

1. Consider the history of the best chromosomes, if the final functional solution has been evolved during ES performance;

2. Select $\mathbb{C}_{t_1}$ created at generation $t_1$ such that the fitness of the best chromosome in question has been changed in comparison with the previous one;

3. Choose $\mathbb{C}_{t_2}$ produced at generation $t_2$ such that the fitness of the best chromosome in question has been increased in comparison with chromosome $\mathbb{C}_{t_1}$, $t_2 > t_1$, $\mathcal{F}_{t_1} < \mathcal{F}_{t_2}$ and there is no improvement in terms of fitness function between generations $t_1$ and $t_2$;

4. Generate the differential chromosome $\mathbb{D}_{\mathbb{C}_{t_1}\mathbb{C}_{t_2}}$.

Table 3.11: The functional characteristics of chosen functional sets.

| Functional set | $\mathrm{p_t}(\{\mathrm{or}\})$ | $\mathrm{p_t}(\{\mathrm{and}\})$ | $\mathrm{p_t}(\{\mathrm{exor}\})$ | $\mathrm{p_t}(\{\mathrm{not}\})$ | $\mathrm{p_i}(\{\mathrm{not}\})$ |
|---|---|---|---|---|---|
| $M1 = \{6,7,8,9,10,11,21,22\}$ | 0 | 0.5 | 0.5 | 0 | 0.5 |
| $M2 = \{6,10,11,21,22\}$ | 0 | 0.2 | 0.8 | 0 | 0.4 |
| $M3 = \{6,7,8,9,10\}$ | 0 | 0.8 | 0.2 | 0 | 0.4 |
| $M4 = \{4,6,10\}$ | 0 | 0.33 | 0.33 | 0.33 | 0 |
| $M5 = \{6,7,10\}$ | 0 | 0.66 | 0.33 | 0 | 0.16 |
| $A1 = \{4,6,10,12\}$ | 0.25 | 0.25 | 0.25 | 0.25 | 0 |
| $A2 = \{6,7,10,11,12,13\}$ | 0.33 | 0.33 | 0.33 | 0 | 0.25 |
| $A3 = \{6,7,8,10,11,12,13,14,21\}$ | 0.33 | 0.33 | 0.33 | 0 | 0.33 |
| $A4 = \{6,7,8,9,10,11,12,13,14,15,21,22\}$ | 0.33 | 0.33 | 0.33 | 0 | 0.5 |

The selection procedure mentioned above ensures that the differential chromosomes have been generated from the chromosomes with improved fitness.

The conditional probabilities given in Eq. 3.5.1 - Eq. 3.5.10 have been calculated for all differential chromosomes separately for experiments $\xi_{Mi}, i = 1, \cdots, 5$ and $\xi_{Ai}, i = 1, \cdots, 4$ and reported in Fig. 3.14 - Fig. 3.17. The location of logic cells inside circuit can be represented as a string of logic cells, because the number of rows in circuit layout is 1. If we labeled the logic cells located from left to right as $0, \cdots, (N_{cols} - 1)$, then we can represent them in figures on horizontal axes (Fig. 3.15 - Fig. 3.17). Thus, the horizontal axe defines the location of the logic gates in the rectangular array. Because the number of columns in the rectangular array is 1, location of each logic gate can be defined by the index of the column. For example, 5 at horizontal axe defines the logic cell located in the 5-th column. We apply similar approach to show the location of the circuit outputs (Fig. 3.14). In further discussion we will refer to the cell located at less significant position as to the position is located at the left side of the rectangular array and to the most significant position if it is

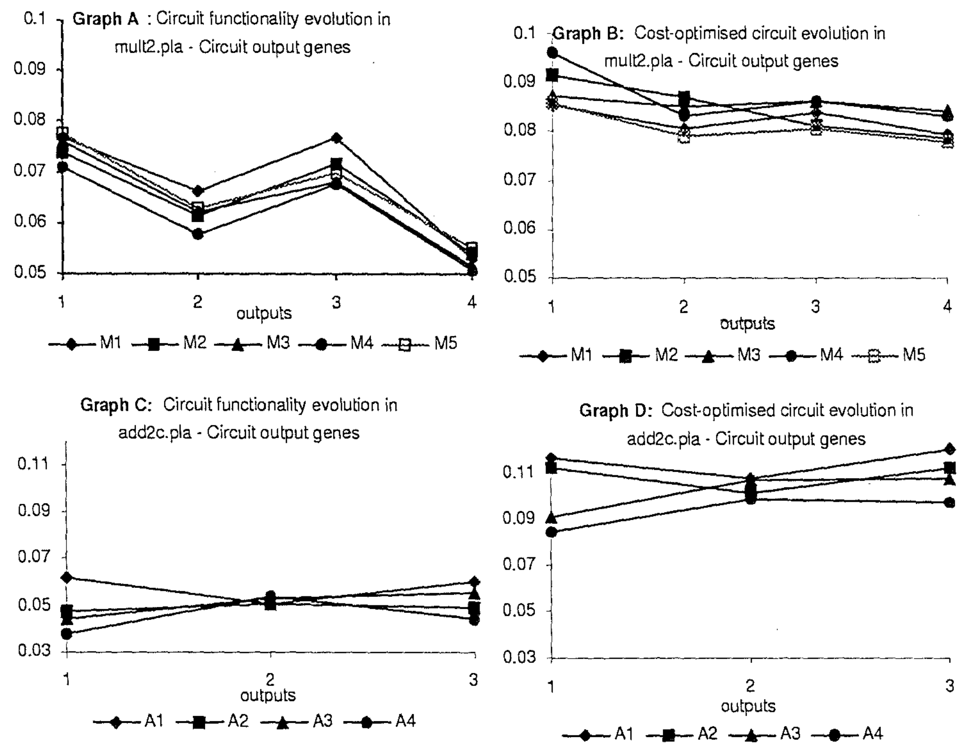


Figure 3.14: Circuit Output Genes. The horizontal axis defines the outputs in the evolved circuits. The vertical axes in Graphs A and C correspond to the conditional probabilities calculated according to Eq. 3.5.6 for the two-bit multiplier and the two-bit adder respectively. These graphs describe the evolutionary process aimed to evolve a fully functional circuit, $\Upsilon_{\mathcal{F}_1}$. Conditional probabilities calculated for both tested functions using Eq. 3.5.10 are mapped to the vertical axes of Graphs B and D. These graphs illustrate the evolutionary process that produces cost-optimised circuit, $\Upsilon_{\mathcal{F}_2}$. A comparison of these graphs shows that the conditional probabilities obtained for evolutionary process aimed to evolve fully functional circuits $\Upsilon_{\mathcal{F}_1}$ 2 times lower than the similar probabilities obtained for evolutionary process $\Upsilon_{\mathcal{F}_2}$. **Conclusion**: The circuit output genes are more essential during evolution $\Upsilon_{\mathcal{F}_2}$.

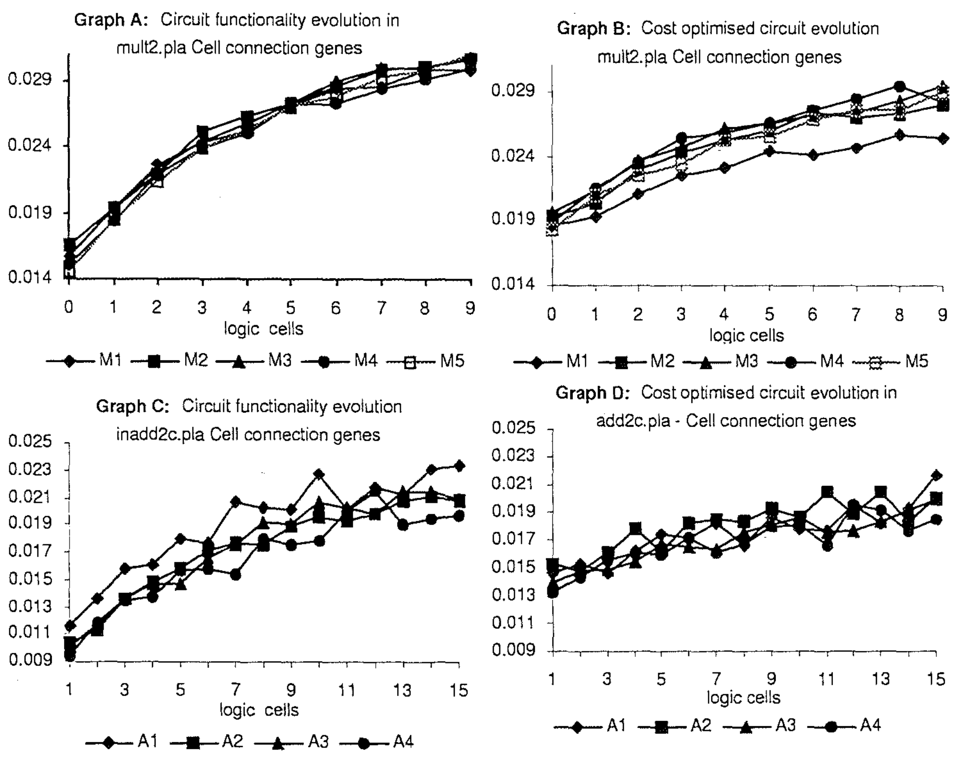


Figure 3.15: Connection Genes. The horizontal axis defines the positions of logic gates in the circuit layout. The vertical axes in Graphs A and C correspond to the conditional probabilities calculated according to Eq. 3.5.6 for the two-bit multiplier and the two-bit adder respectively. These graphs describe the evolutionary process aimed to evolve a fully functional circuit, $\Upsilon_{\mathcal{F}_1}$. Conditional probabilities calculated for both tested functions using Eq. 3.5.10 are mapped to the vertical axes of Graphs B and D. These graphs illustrate the evolutionary process that produces cost-optimised circuit, $\Upsilon_{\mathcal{F}_2}$. Comparison of these graphs shows that the conditional probabilities in all evolution processes are higher for genes located in more essential positions. The conditional probabilities for evolutionary process $\Upsilon_{\mathcal{F}_2}$ in logic gates located in less essential positions are higher than the similar probabilities for evolutionary process $\Upsilon_{\mathcal{F}_1}$. **Conclusion:** The genes located in more essential positions are very important in both evolutionary processes. The genes located in less essential positions become more essential in evolutionary process $\Upsilon_{\mathcal{F}_2}$.

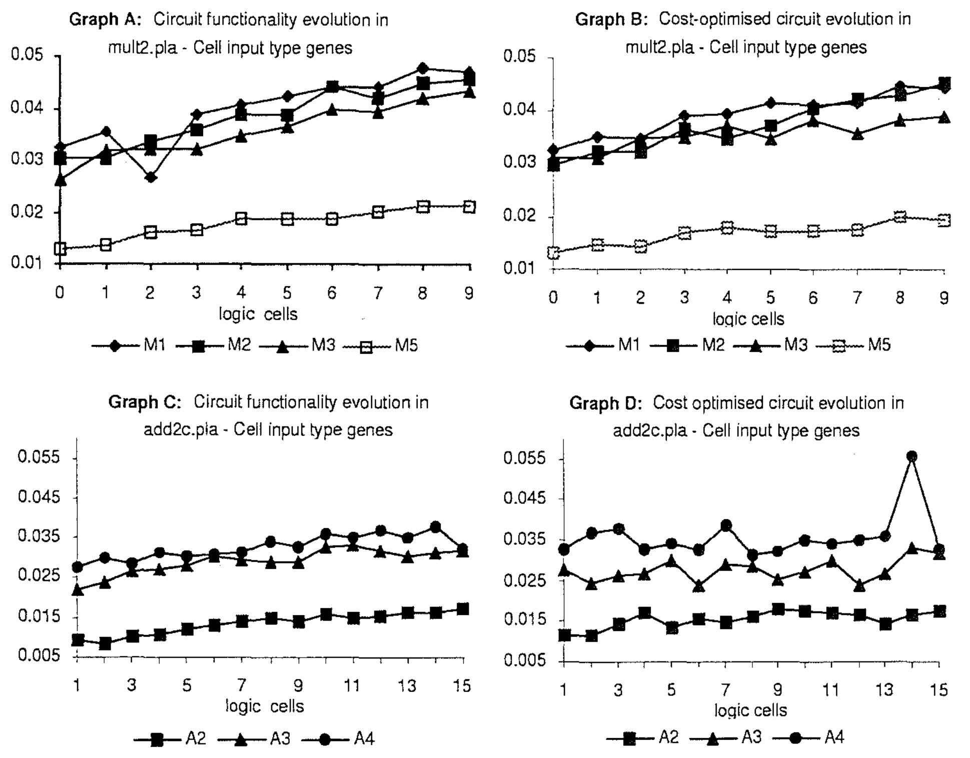


Figure 3.16: Input Type Genes. Horizontal axe defines the positions of logic gates in the circuit layout. Vertical axes in Graphs A and C correspond to the conditional probabilities calculated according to Eq. 3.5.5 for the two-bit multiplier and the two-bit adder respectively. These graphs describe the evolutionary process aimed to evolve a fully functional circuit, $\Upsilon_{\mathcal{F}_1}$. Conditional probabilities calculated for both tested functions using Eq. 3.5.9 are mapped to the vertical axes of Graphs B and D. These graphs illustrate the evolutionary process that produces cost-optimised circuit, $\Upsilon_{\mathcal{F}_2}$. Comparison of these graphs shows that the conditional probabilities in all evolution processes are slightly higher for genes located in more essential positions. No differences between two evolutionary processes have been noticed. **Conclusion:** There is no difference between two evolutionary processes. The genes in logic gates located in more essential positions are a bit more essential than in logic gates located in less essential positions.

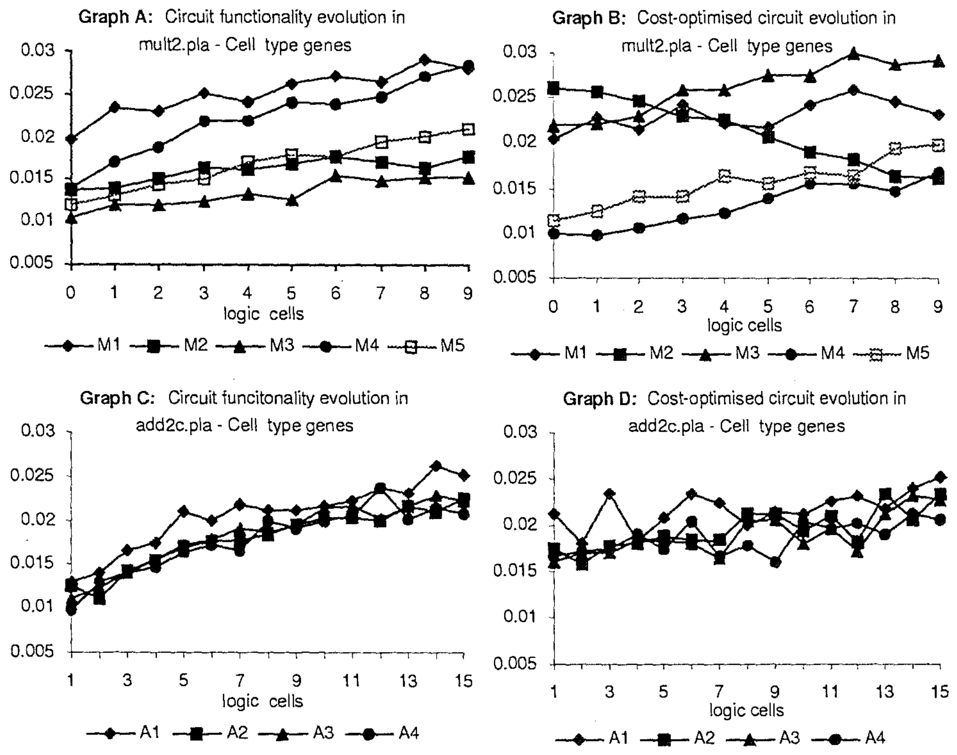


Figure 3.17: Cell Type Genes. The horizontal axe defines the positions of logic gates in the circuit layout. The vertical axes in Graphs A and C correspond to the conditional probabilities calculated according to Eq. 3.5.3 for the two-bit multiplier and the two-bit adder respectively. These graphs describe the evolutionary process aimed to evolve a fully functional circuit, $\Upsilon_{\mathcal{F}_1}$. Conditional probabilities calculated for both tested functions using Eq. 3.5.8 are mapped to the vertical axes of Graphs B and D. These graphs illustrate the evolutionary process that produces cost-optimised circuit, $\Upsilon_{\mathcal{F}_2}$. Comparison of these graphs shows that the conditional probabilities in all evolution processes are slightly higher for genes located in more essential positions. No differences between two evolutionary processes have been noticed for the two-bit multiplier. In case of evolving the two-bit adder, during circuit functionality evolution ($\Upsilon_{\mathcal{F}_1}$) the conditional probabilities obtained for logic gates located in less essential positions are slightly higher than in cost-optimised circuit evolution $\Upsilon_{\mathcal{F}_2}$. **Conclusion:** There is no clear difference between the two evolutionary processes for both logic functions tested.

Table 3.12: Experimental Results: $\overline{F_1^{bf}}$ and $\overline{F_2^{bf}}$ are the mean fitnesses $F_1$ and $F_2$ of the best evolved chromosomes respectively; $\overline{F_2(\mathcal{N}_f)}$ denotes the average fitness $F_2$ for fully functional circuits evolved; $N_f$ is the fully functional circuit; $R(\mathcal{N}_f)$ defines the number of fully functional circuits evolved.

| **Circuit** | $n$ | $m$ | **Functional set, $\mathbb{FS}$** | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}$ | $\overline{F_2(\mathcal{N}_f)}$ | $R(\mathcal{N}_f)$ |
|---|---|---|---|---|---|---|---|
| mult2.pla | 4 | 4 | $M1$ | 97.3484 | 7.665 | 7.41 | 332 |
| | | | $M2$ | 98.0812 | 7.138 | 7.14 | 390 |
| | | | $M3$ | 98.2109 | 7.499 | 7.32 | 517 |
| | | | $M4$ | 98.6609 | 7.254 | 7.24 | 534 |
| | | | $M5$ | 99.0844 | 7.222 | 7.09 | 707 |
| add2c.pla | 5 | 3 | $A1$ | 94.1437 | 10.184 | 10.9529 | 85 |
| | | | $A2$ | 94.1833 | 10.038 | 10.4434 | 106 |
| | | | $A3$ | 93.7333 | 9.956 | 10.4902 | 102 |
| | | | $A4$ | 93.7041 | 10.104 | 10.5290 | 85 |

located at the right side of the rectangular array. Analysing Fig. 3.14 - Fig. 3.17, we can conclude:

1. The conditional probabilities behave similarly for both tested logic functions: two bit multiplier and two-bit adder. So, the conclusions drawn below can be extended to the class of arithmetic logic functions.

2. The conditional probabilities for the circuit output and connection genes are higher for the evolution process $\Upsilon_{F_2}$ than for $\Upsilon_{F_1}$. This means that the circuit output and connection genes are more actively participating in the evolution of fully functional circuits ($\Upsilon_{F_2}$);

3. The conditional probabilities for connection genes in logic gates located in less significant positions (i.e. in the low-level columns) are lower then in the same genes located in the high-level columns. This means that these genes are more active participant during evolution $\Upsilon_{F_2}$ than in $\Upsilon_{F_1}$. In other words ES considers

the less significant logic gates as more essential if the evolution $\Upsilon_{F_2}$ is carried out.

4. The conditional probabilities for the circuit output genes are approximately 5 times higher than for other types of genes. In other words, the circuit output gene is the most significant gene in the given chromosome representation.

5. The level of conditional probabilities for input cell type is 2 times higher than conditional probability for cell type gene. This means, that during evolution changes of input cell genes have more influence ES performance, than the cell type gene.

One should remember remind that the experimental results discussed above have been obtained using a rudimentary $(1+\lambda)$ ES. This algorithm involves only mutation that applied to the population generated from the best chromosome. Therefore, the examined experimental results reflect the behaviour of the circuit mutation operator. Hence, the performance of ES can be improved if these suggestions are taken into account in the mutation operator. In other words, during mutation, different type of genes located differently have to participate differently during the evolutionary processes $\Upsilon_{F_1}$ and $\Upsilon_{F_2}$. This is an issue of further work. Also, we can conclude that the functionality of logic gates has to be described by two genes. The point is that the experimental results prove that input and gate functionalities influence differently on ES performance.

## 3.6 Summary

In this chapter some issues concerning analysis and verification of evolved circuits in the gate-level EHW have been considered. Firstly, some function representations and their suitability for the extrinsic EHW have been examined. Secondly, the dynamic fitness strategy has been introduced to improve the quality of evolved circuits. Thirdly, new optimisation criterion to evolve different types of circuit has been suggested. And finally, two evolutionary processes defined by proposed dynamic fitness function are investigated using a probabilistic analysis approach, that shows the difference of these evolutionary processes.

Analysis of the function representations, that can be suitable for using in the extrinsic EHW illustrates that

1. The truth table can be used in any occasion because it defines the logic function for *all ON and OFF conditions.*

2. The minterms table is *not* suitable to use in the extrinsic EHW because in this case the input combinations where function is "0" are considered as *not DON'T CARE.*

3. The cube representation is appropriate to evolving logic circuits if the logic operators are redefined according to the cube representation.

In this chapter we have introduced the dynamic fitness function and showed that the extrinsic EHW can be applied to design the FPGA-based as well as MOS-based circuits.

The idea of the dynamic fitness function is employed firstly to evolve the fully functional circuit and secondly to improve the quality of evolved circuits. At each stage of dynamic fitness function the multi-criteria optimisation can be performed. The performance of dynamic fitness function has been compared with performance of classical multi-objective methods, such as the MIN-MAX formulation, the method of distance functions and the method of objective weighting. The experimental results show that the dynamic fitness is better or comparable with the methods mentioned above, if criteria $F_1$ and $F_3$ are activated. Using the dynamic fitness function allows us to decrease the computational time of program, because the quality of evolved circuits is estimated at the second stage of evolution.

FPGA technology considers each basic logic gate as a building block. Therefore, the number of primitive active logic gates can be considered as an optimisation criteria during evolution of FPGA-based circuit. Defining the level of FPGA used, the number of primitive active logic gates can be substituted by the number of building blocks in FPGA-based circuit. For example, at the gate-level, the quality of FPGA-based circuit can be defined by the number of primitive logic gates such as NOT, AND, OR or EXOR. At the function-level, the quality of the FPGA-based circuit can be estimated by the number of such components as multiplexer, half adder, multiplier, full adder, etc.. These components are considered as a basic building blocks. In this case the quality of the evolved circuit can be defined by the number of active building blocks used in the circuit, regardless of the cost of these blocks.

CMOS, NMOS and dynamic MOS circuits consist of transistors, diodes, etc.. Hence, the quality of these circuits can be defined by the number of transistors used in circuit. The number of transistors required to implement each of these circuits is different, therefore the optimisation process for each of these technologies has to be considered separately.

The FPGA- and MOS-based circuit structures evolved have been compared. It has been shown that the most efficient designs strongly depend on the implementation technology. In other words, the most efficient FPGA-based circuit does not coincide with the most efficient MOS-based circuit. An analysis of the most efficient 7-gate FPGA-based circuits of two-bit multiplier shows that all evolved circuits contain the same number of transistors in the CMOS, NMOS and dynamic MOS circuits. This means that all existing efficient FPGA-based circuits have the same structural features, i.e. they contain 2 EXOR and 5 AND logic gates. At the same time, an analysis of the circuits with the same number of transistors show that these circuits can contain different number of primitive logic gates.

So, we can conclude that the proposed dynamic fitness function strategy can be used to design the circuit of any technology. This is an evidence of universality of this method. Also, this fitness function can be used in both gate- and function-level extrinsic EHW, because the FPGA-based design assumes using the high-level elements.

The probabilistic analysis has been performed in order to define how different types of genes and their location influence the algorithm performance. The experimental results show that the algorithm performance depends on the proportion of logic gates used in functional sets. It has been shown that different types of genes influence differently the algorithm performance. Thus, the conditional probabilities for circuit output genes are 5 times higher than for connections genes. It means that the circuit output gene has been changed 5 times more often in chromosomes with improved fitness function. The connection genes influence differently in different locations of logic gate. Thus the conditional probability for connection genes located to the left side of circuit is lower then for genes located in the right side of circuit. It has been found that there are two type of functionalities in logic cell: gate and cell functionalities. It has been defined that they have different influence the algorithm performance. The evolutions involved activating while the first and second fitness functions behave differently.

Two criteria to define the functionality of the logic function are introduced. These are:

1. the type of primitive logic function describing the behaviour of the building block (AND, OR, NOT, EXOR);
2. the type of inputs used in the building block (primary or inverted).

The experimental results show that these two functionality types have different influence the algorithm performance. Therefore, we can conclude that in the chromosome representation, the functional behaviour of the logic gate has to incorporate two different gene types defining the gate functionality: (1) input gate functionality gene (referred also as input type gene) and (2) primitive gate functionality gene (referred also as cell type gene).

# Chapter 4

# Circuit layout evolution

## 4.1 Introduction

The EHW approach has been introduced in order to develop self-reconfigurable and self-adaptive hardware. The main application area of such hardware is to perform efficiently different tasks in dynamically changing environment without human intervention. In this case the hardware should be able to define the initial parameters by itself. This is a very complicated task. At the current stage of research, the initial parameters of EHW are defined by the designer, since they are ultimately linked with the complexity of considered problem. In order to overcome this difficulty, we propose to use evolved EHW parameters together with circuit functionality. This will provide the EHW with self-adaptation to required parameters. EHW parameters include the circuit layout, connectivity restrictions, complexity of building blocks, functional set of logic functions, etc.. The conducted EHW analysis confirms that these parameters influence the algorithm performance. The circuit layout and the connectivity parameter are very important, since they define the configuration of chromosomes and restrictions of hardware. For this reason, these parameters are chosen to participate in the circuit evolution. The technique that combines the circuit layout and circuit functionality evolution at the same time is introduced in this Chapter. This is the first attempt to create hardware that adjusts the size of rectangular array during evolution. The basic idea of evolving the circuit layout together with the circuit functionality and some experimental results has been first suggested and previously reported in [1], [2].

In this Chapter we will discuss one of the possible ways to design EHW with self-adaptation to circuit configuration. The choice of suitable circuit geometry is a very complicated task and is intimately linked with the complexity of the function implemented. So, in order to avoid this we investigate the possibility of evolving the circuit geometry at the same time as trying to evolve 100% functional circuits. The circuit layout, also referred to as circuit geometry, defines the length of the chromosome, thus we work with chromosomes of variable length. In this scheme, mutation is carried out in two ways. First, we can mutate genes associated with a circuit in a homogeneous geometry, and secondly, we can by mutation choose the circuit geometry. The main purpose of circuit layout evolution was to evolve the circuit layout together with evolving circuit functionality.

In our further research we define several strategies for the gate-level extrinsic EHW with heterogeneous circuit layout. Two types of evolutionary algorithms (EA) are implemented: a standard elite genetic algorithm (GA) and a rudimentary $(1+\lambda)$ evolutionary strategy (ES). The standard elite genetic algorithm contains initialisation, selection, crossover and mutation operators. These operators have been designed for both circuit layout and gate levels in the chromosome representation. Elitism is used in order to provide the survival features of the fittest individual. The evolutionary strategy involves initialisation and mutation operators. The new population is filled with the fittest individual. The operators are similar to the ones applied to the standard elite GA.

In order to evolve better circuits in terms of the number of active gates we use the dynamic fitness function ($\mathcal{F}_1 + \mathcal{F}_2$) that contains two main stages. At the first stage the objective in digital evolution behaviour is to merely produce a 100% functionally correct circuit (functionality criteria, $F_1$, $\mathcal{F}_1 = F_1$). So, the evolutionary process is terminated at this point. Here we continue to evolve the circuit beyond the point of 100% correctness by modifying the fitness function to include a measure of circuit's efficiency (criteria defined by the number of active logic gates in the circuits, $F_2$, $\mathcal{F}_2 = F_2$). We will investigate cases where we use homogeneous, heterogeneous or partially heterogeneous (heterogeneous only at the initialisation stage of evolutionary algorithm) circuit layouts during algorithm execution and determine the algorithm performance as a function of both fitness measures.

## 4.2 Relationships between the circuit layout and algorithm performance

Justification of evolving circuit layout together with circuit functionality can be derived empirically from the following series of experiments. The main purpose of this series of experiments was to investigate how the connectivity parameter and circuit layout affect the algorithm performance. The circuit layout can be characterised by the following parameters:

1. connectivity parameter;
2. the number of rows in rectangular array;

3. the number of columns in rectangular array.

The algorithm performance can be estimated by:

1. the mean functionality fitness of the best chromosome over 100 runs, $\overline{F_1^{bf}}$ (av.F1);

2. the mean number of active gates in fully functional designs evolved over 100 runs, $\overline{F_2^{bf}(\mathcal{N}_f)}$ (100av.F2);

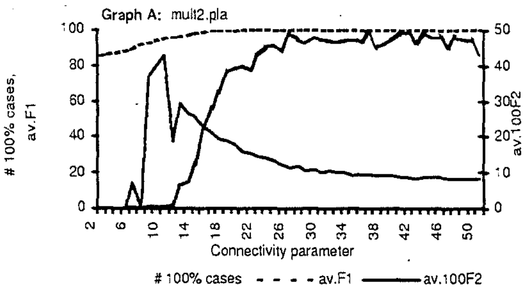
3. the number of fully functional circuits $\mathcal{N}_f$ evolved, $\mathcal{R}(\mathcal{N}_f)$ (# 100% cases).

In order to get the first positive results in a reasonable period of time we arrived at the parameters shown in Table 4.1. We perform three different experiments. Each of the experiments has different circuit layout parameters. Therefore, the circuit parameters are listed separately for each experiment performed (Table 4.1). This series of experiments has been performed using a rudimentary $(1 + \lambda)$ evolutionary strategy with uniform mutation.

These experiments have yielded some very interesting results, e.g. that performance may be dependent on the number of columns and the connectivity parameter, rather than EA parameters only. In this case the algorithm performance is defined by the number of fully functional circuits evolved during 100 runs and by the quality of evolved circuits. The quality of evolved circuits is determined by the number of active logic gates in the circuit. Therefore, the target implementation technology for evolved circuits is FPGA. Such arithmetic logic functions as a half adder (add1c.pla) and a two-bit multiplier (mult2.pla) are chosen to verify the hypothesis.

Table 4.1: Initial data: Circuit layout and algorithm performance. The EA parameters used in this experiment and further in experimental results have been chosen according to empirical study of EA behaviour.

| Circuit | add1c.pla | mult2.pla |
|---|---|---|
| Functional set | | {1, 2, 9} |
| Type of algorithm | $(1+\lambda)$ ES | |
| Population size | 5 | |
| Number of generations | 100 | 5000 |
| Number of algorithm runs | 100 | |
| Mutation rate, $p_m$ | 0.05 | |
| **Investigation of connectivity parameter** | | |
| Circuit layout, $N_{cols} \times N_{rows}$ | 50x1 | 50x1 |
| Connectivity parameter, $(N^{min}_{connect}, N^{max}_{connect}, \Delta N_{connect})$ | (1, 50, 1) | (1, 50, 1) |
| **Investigation of the number of columns** | | |
| Circuit layout, $(N^{min}_{cols}, N^{max}_{cols}, \Delta N_{cols})$ $N_{rows}$ | (2, 50, 1) 1 | (2, 50, 1) 1 |
| Connectivity parameter, $N_{connect}$ | 2,5,10,$N_{cols}$ | 2,5,10,$N_{cols}$ |
| **Investigation of the number of rows** | | |
| Circuit layout, $(N^{min}_{rows}, N^{max}_{rows}, \Delta N_{rows})$ $N_{cols}$ | (2, 50, 1) 4 | (2, 50, 1) 4 |
| Connectivity parameter, $N_{connect}$ | 4 | 4 |



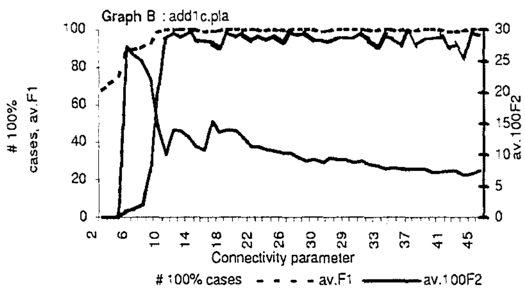


Figure 4.1: Dependence the algorithm performance on the connectivity parameter. Graphs illustrate that the algorithm performance depends on the connectivity parameter. There are four specific ranges of areas for connectivity parameter showing that the algorithm performs differently.

**Connectivity parameter and algorithm performance.**

In this section we will examine how the performance of the evolutionary algorithm is influence by increasing the flexibility of internal connections of the circuit. Fig. 4.1 shows the summarised experimental results. For each set of runs the number of columns and rows has been homogeneous and the connectivity parameter allowed to vary.

The obtained results show that with increasing the internal connectivity in the circuit, the number of fully functional solutions acquired increases. Four distinctive ranges of connectivity parameter can be identified as follows:

1. No fully functional solutions have been evolved;

2. The percentage of fully functional circuits evolved is 5% or less;

3. The percentage of fully functional circuits evolved is more than 5% and less than 90 %;

4. The percentage of fully functional circuits evolved is 90% or more.

No functional solutions have been evolved for both functions if the connectivity parameter is less then 5.

During the second phase, very few functional solutions are evolved. The size of the synthesised circuit is very large in comparison with the optimally evolved solution. For instance, the optimal evolved two-bit multiplier contains 7 primitive logic gates. The two-bit multipliers evolved during second phase consist of at least 35 logic gates. The same effect can be noticed when the half adder is evolved.

During the third phase, a sufficient number of fully functional circuits is evolved. The number of primitive logic gates in the circuit evolved during this stage decreases with increasing the connectivity parameter in the circuit layout.

During the fourth phase, no improvements in algorithm performance can be noticed with increasing the connectivity parameter. The average number of active logic gates in the circuit is stabilized and no improvements in term of the quality of evolved circuits can be found with increasing the connectivity parameter during this phase.

So, we can conclude that the connectivity parameter influences the algorithm performance. The higher the connectivity parameter, i.e. less restrictions inside circuit results in better algorithm performance. Note that there is undesirable range of connectivity parameter, within which the algorithm performs poorly.

The algorithm performance illustrated above can be explained as follows. Let "self-reproductive" logic circuit be the circuit that acts as a wire. For example, circuit described logically as NOT(NOT(a)) is "self-reproductive because it implements itself and acts as a wire. The depth of logic circuit is the minimal number of columns required to implement the circuit in question. Let $N_{cols}$ be the number of columns in the rectangular array and $N_{connect}$ be the connectivity parameter. The minimal depth

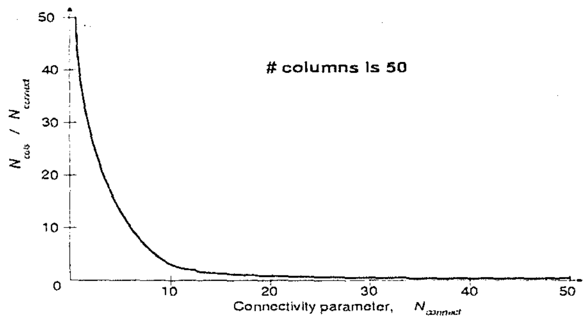


Figure 4.2: Relationship between the connectivity parameter and the minimal depth of logic circuit that can be implemented using given circuit layout. Graph shows that there is a range where the minimal depth of the logic circuit is relatively high for a specific range of connectivity parameter. Comparing this range with one obtained empirically, we can notice that they are identical.

of logic circuit that can be implemented on the rectangular array can be defined as $d = \frac{N_{cols}}{N_{connect}}$, if "self-reproductive" circuit is not allowed to be used.

Fig. 4.2 depicts the relationship between the connectivity parameter and the minimal depth of logic circuit that can be implemented using the given circuit layout. In our experiment the number of columns in the rectangular array is set to 50. The minimal circuit depth is calculated for each value of connectivity parameter starting with 1. The graph illustrated in Fig. 4.2 shows that only the circuits with a relatively high number of logic gates can be implemented on rectangular array with circuit layout 1x50. For example, the depth of the logic circuit has to be 17 or higher, if the number of columns is 50 and the connectivity parameter is 3.

So, the theoretical foundations summarised in Fig. 4.2 are proved empirically (see Fig. 4.1). Thus, as it has been mentioned above, no fully functional circuits have been evolved with relatively low connectivity parameter, since the depth of examined circuit is relatively low and with these restrictions multi-output functions can not be mapped into rectangular array.

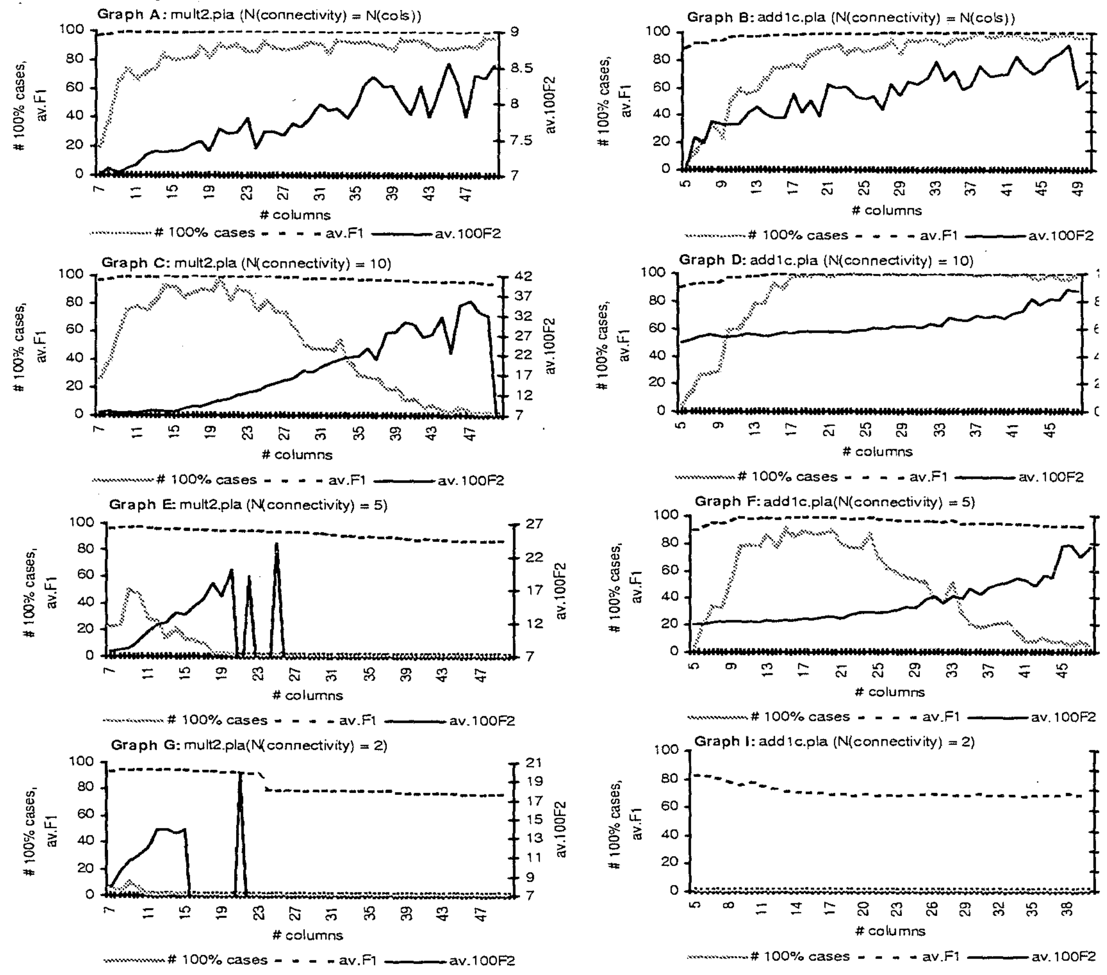


Figure 4.3: Dependence of the algorithm performance on the number of columns. Graphs A and B demonstrate that with increasing the number of columns in the circuit without connectivity restrictions ($N_{connect} = N_{cols}$), the number of primitive active logic gates in the fully functional circuits increases. Graphs C, E and G shows that with higher connectivity restrictions the number of columns required to evolve fully functional two-bit multipliers decreases. The smaller the connectivity parameter, the narrower the range of the number of columns, that can produce fully functional solution. Similar conclusion can be made considering Graphs D, F and I that correspond to the full adder.

### The number of columns and algorithm performance.

In this series of experiments we investigate how the number of columns in the rectangular array influences the algorithm performance. The experimental data are summarised in Fig. 4.3. The same experiment has been repeated for circuit layouts with connectivity parameter equal to 2, 5, 10 and $N_{cols}$. Analysing the shape of the obtained curves we can conclude that the algorithm performs differently for different values of connectivity parameter. Let us consider the case when $N_{connect} = N_{cols}$. In this case, the number of fully functional solutions evolved is stabilized relatively quickly. At the same time, the number of primitive logic gates involved in the circuit increases as the number of columns in rectangular array is increased. Therefore, in order to evolve the logic function with minimal number of primitive logic gates, a suitable number of columns has to be chosen. With decreasing the connectivity parameter in this experiment, the behaviour of the algorithm performance changes. For example, only few fully functional solutions have been evolved with $N_{connect} = 2$. This can be explained by the circuit depth limitation specified by chromosome representation. Based on data, given in Fig. 4.3, we can conclude that the connectivity parameter is ultimately linked with the number of columns in the rectangular array. This has drastic influence on the number of fully functional circuits evolved and its quality.

### The number of rows and algorithm performance

In this series of experiments we investigate how the number of rows in the rectangular array influences the algorithm performance. For this purpose the number of columns and the connectivity parameter have been chosen to be constant (Table 4.1).

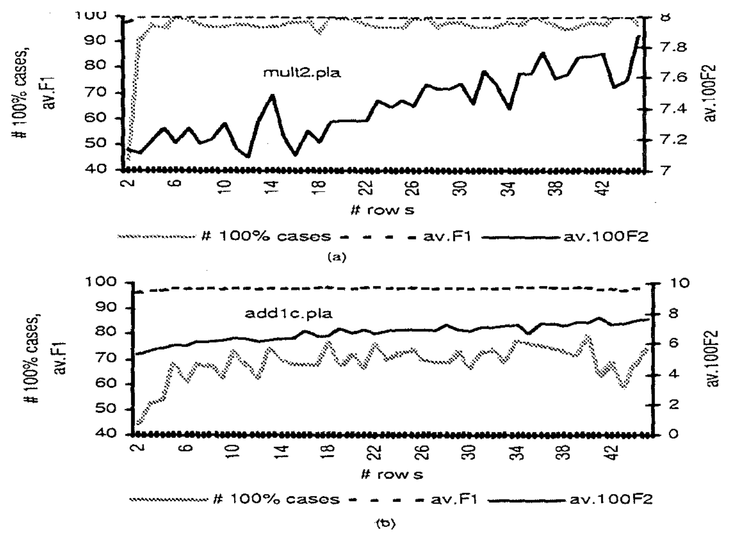


Figure 4.4: Dependence the algorithm performance on the number of rows. Graphs illustrate that the algorithm performance does not depend on the number of rows in the rectangular array.

Analysing the experimental data shown in Fig. 4.4 we can conclude that the algorithm performance does not depend on the number of rows in rectangular array. No improvements have been found in terms of algorithm performance with increasing the number of rows in rectangular array.

So, finally we can conclude that the algorithm performance depends on the connectivity parameter and the number of columns in the rectangular array and does not depend on the number of rows. The dependence of algorithm performance on the circuit layout has been investigated on relatively small fully functional circuits. Thus, it is likely that the number of rows in the rectangular array influences the algorithm performance, if a higher complexity circuit is evolved. The number of active primitive logic gates in the evolved structures strongly depends on the connectivity parameter and the number of columns in the rectangular array.

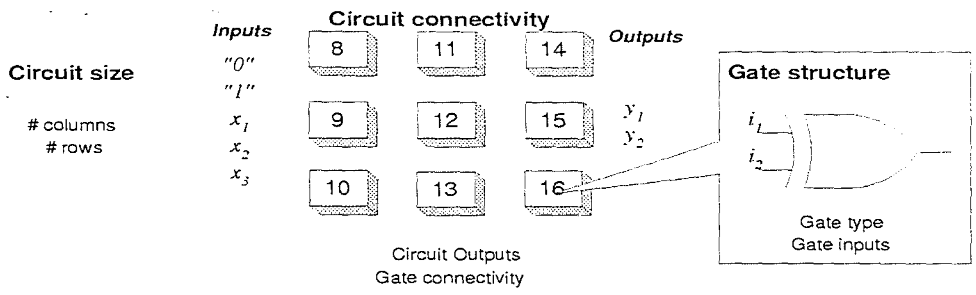


Figure 4.5: Schematic of chromosome structure implementing a 3-input 2-output logic function

## 4.3 An extrinsic EHW with heterogeneous circuit layout

In this section, an extrinsic EHW that evolves a circuit layout together with circuit functionality is introduced.

### 4.3.1 Encoding

Two aspects are required to define any combinational logic network. The first is the cell-level functionality and the second is the inter-connectivity of the cells between the circuit inputs and outputs. An encoding of chromosome was adopted that satisfies these two aspects.

A combinational logic circuit is represented as a rectangular array of logic gates (Fig. 4.5). Each logic cell in this array is uncommitted and can be removed from the network if it proves to be redundant. The inputs to any cell in the combinational network may be logical constants, primary and inverted inputs, as well as the outputs of logic cells which are in columns to the left of the cell in question. In the work reported in this Chapter we define each logic function to be chosen from the set of functions AND, OR, NOT, EXOR with primary and inverted inputs or a multiplexer.

The chromosome is represented by a 3-level structure:

1. Layout structure;

2. Circuit structure;

3. Gate (cell) structure.

At the first level, the global characteristics of the circuit are defined: These are the connectivity parameter and the number of rows and columns. The circuit geometry can be changed at this level. At the second level, the array of cells are created and the circuit outputs are determined. Finally, the third level represents the structure of each cell in the circuit. This data consists of the number of inputs, the input connections and the functional gene. The number of inputs in the cell depends on the type of cell and is defined when the value of functional gene is known (i.e. a 2-1 multiplexer has two inputs and one control input while all others have only two inputs). Note that the number of inputs as well as the number of outputs are allowed to be variable, but in this Chapter we consider only 2– or 3–input 1–output gates.

An example of the chromosome representation with the actual circuit structure is given in Fig. 4.6. Let us examine a possible circuit representing a full adder. This function has 3-inputs and 2-outputs and is implemented here on a combinational network with 3x3 circuit geometry ($N_{cols} \times N_{rows}$). The circuit inputs are labeled as follows: 0 and 1, which represent the logical constants 0 and 1 respectively, labels 2, 3 and 4 correspond to the input variables $x_0$, $x_1$ and $x_2$ respectively. The inverted inputs $\overline{x_0}$, $\overline{x_1}$, $\overline{x_2}$ are represented as 5, 6 and 7. In this example the functional gene (shown in bold) represents one of the 13 possible gates (AND, OR, EXOR with primary and inverted inputs or multiplexer). The functional gene may be a positive or negative

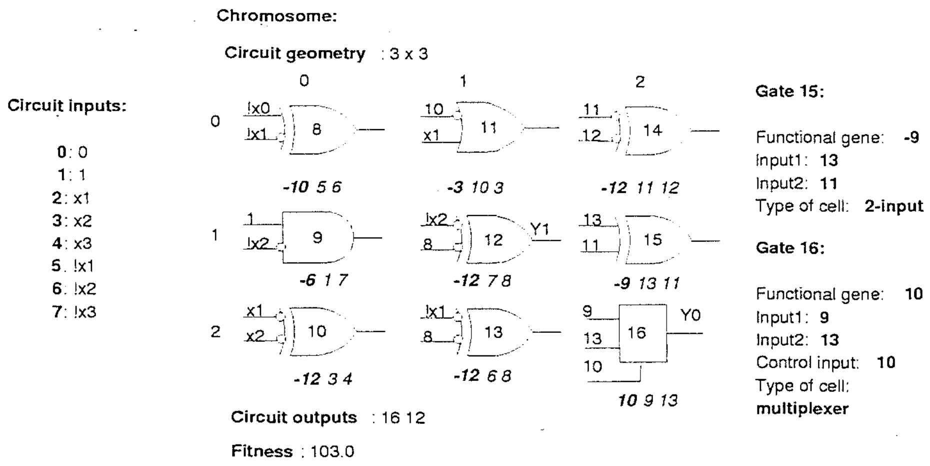


Figure 4.6: An example of the phenotype and corresponding genotype of a chromosome with 3x3 circuit layout

integer. If positive then the function is a multiplexer and the integer represents the control connection. If functional gene is negative, we use an encoding table to define the type of gate (Table 4.2).

The output of each cell is assigned an individual address. Thus the output of the cell located in the $0^{th}$ column and in the $0^{th}$ row is labeled as 8. The output of logic cell in the $2^{nd}$ column and $2^{nd}$ row is labeled as 16. The number of circuit outputs is defined by the number of outputs in the logic function implemented. The logic cell label determines each of these outputs. Let us examine the encoding of the $12^{th}$ logic cell in genotype $< -12\ \ 7\ \ 8 >$. We refer to this representation of gate as "*gate genotype*". The functional gene defining the type of this gate is -12. This value corresponds to the EXOR gate with inverted inputs in the gate-encoding table (Table 4.2).

The examined cell has two inputs. The first input is connected to the input $\overline{x_2}$ and second to the output of the $8^{th}$ cell. Cell 8 depends on two variables: $\overline{x_0}$ and

Table 4.2: Gate functionality according to the $b_0(z)$ gene in chromosome

| Gene functionality, $b_0(z)$ | Gate function |
|---|---|
| -1 | $x_1 \wedge x_2$ |
| -2 | $x_1 \wedge \overline{x_2}$ |
| -3 | $\overline{x_1} \wedge x_2$ |
| -4 | $\overline{x_1} \wedge \overline{x_2}$ |
| -5 | $x_1 \vee x_2$ |
| -6 | $x_1 \vee \overline{x_2}$ |
| -7 | $\overline{x_1} \vee x_2$ |
| -8 | $\overline{x_1} \vee \overline{x_2}$ |
| -9 | $x_1 \oplus x_2$ |
| -10 | $x_1 \oplus \overline{x_2}$ |
| -11 | $\overline{x_1} \oplus x_2$ |
| -12 | $\overline{x_1} \oplus \overline{x_2}$ |
| -13 | $\overline{x_1}$ |
| -14 | $x_1$ |

$\overline{x_1}$. So, in this case the logic function that describes the 12th cell depends on three variables: $\overline{x_0}$, $\overline{x_1}$ and $\overline{x_2}$. Let us consider the 16th gate, with genotype $< 10 \;\; 9 \;\; 13 >$. Since the functional gene is positive, the gate is a multiplexer and the functional gene with value 10 corresponds to the control input. The inputs to this multiplexer are connected to the outputs of gates 9 and 13. The outputs of the circuit are connected to the outputs of the 16th and 12th logic gates. The fitness $\mathcal{F}$ of a chromosome is defined as follows:

$$\mathcal{F} = \begin{cases} \mathcal{F}_1 = F_1, & F_1 < 100; \\ \mathcal{F}_2 = F_1 + \overline{F_2}, & F_1 = 100. \end{cases}$$

where $F_1$ is the percentage of the circuit output bits that are correct, $\overline{F_2}$ is the number of gates that are not involved in the circuit.

The maximum $\mathcal{F}_2$ is equal to $(100.0 + N_{rows}^{max} \times N_{cols}^{max})$. In this case no gates are used. The fitness function for a full adder is 103.0 for the circuit shown in Fig. 4.6. This means that this circuit represents a 100% functional full adder and there are 3 logic gates that are not involved in the combinational implementation of this circuit. In other words, there are 6 gates, which are actually used to synthesise the full adder, because $\mathcal{F}_2^{max} = 100.0 + 3 * 3 = 109.0$.

### 4.3.2 Objective Function and Fitness

One of the objectives of combinational circuit design is to construct a circuit utilising the minimum number of gates from the behavioural specification of the circuit given by the truth table. The evaluation process consists of the two main steps. First we are trying to find the circuits with 100% functionality ($F_1$ criteria) and ~~second~~ we are trying to minimise the number of active gates in 100% functional circuits ($F_4$ criteria). An active gate is a gate, which is proved to be not redundant. We use two strategies in our EA:

1. $F_1$ strategy;
2. $\mathcal{F}_1 + \mathcal{F}_2$ strategy.

In the first strategy, the chromosome is evaluated using $F_1$ criteria only and once the 100% functional circuit evolved, the evolution process is terminated. In the case of $\mathcal{F}_1 + \mathcal{F}_2$ strategy, $F_2$ criteria is activated as soon as $F_1 = 100.0$ and the number of inactive gates in circuit is estimated. When heterogeneous circuit geometry is employed, $F_2$ is calculated based on the maximum available circuit layout. For example, let the maximum circuit layout is $10 \times 10$. If the fully functional circuit has $3 \times 8$ circuit layout and contains 18 primitive active logic gates ($F_2 = 10$), then $\mathcal{F}_2 = \overline{F_2} = 10 \times 10 - 18 = 82$.

## 4.4 Evolutionary Algorithm

In order to evolve combinational logic circuits, two evolutionary algorithms have been implemented: (1) an evolutionary algorithm using tournament selection with elitism and uniform crossover, and (2) a rudimentary $(1+\lambda)$ evolution strategy. These details are given in the following subsections. During the evolution process we only allow the circuit layout to be changed by mutation or crossover by altering the number of rows or columns. In this case we will refer to this as *heterogeneous circuit layout* during evolution. When the circuit layout is not changed during the evolution process, we refer to it as the *homogeneous circuit layout.*

### 4.4.1 Initialisation

The initialisation procedure contains several steps:

1. Define circuit geometry of chromosomes in population;

2. Initialise the genotype of cells;

3. Generate the circuit outputs for each chromosome.

The first step defines the circuit geometry for the chromosomes. In heterogeneous circuit layout, any circuit geometry may be used up to the maximum number of rows and columns. In homogeneous circuit layout, all chromosomes have the same circuit geometry. We say that we have *homogeneous circuit layout* during the *initialisation* process when the circuit layout for all chromosomes is the same. The *heterogeneous circuit layout* occurs when the chromosomes are initialised with different circuit layouts. During the second and third step the initialisation of cell inputs and circuit outputs is performed in accordance with the connectivity parameter constraint and the type of variables which are able to be present throughout the circuit. Thus if the logic constants are allowed as input connections throughout the circuit, then during the initialisation procedure the inputs of gates can be chosen from the set of inputs constrained by connectivity parameter or from the set of logical constants. The same procedure is true for the primary and inverted primary inputs.

### 4.4.2 Mutation

We use two types of mutation:

1. Circuit mutation;
2. Layout mutation.

The *circuit mutation* allows us to change the type of genes in a chromosome but excludes the number of columns and rows. The *geometry mutation* changes the numbers of rows or columns in the rectangular array. The maximum numbers of rows and columns are predefined. In both cases the mutation rate has to be chosen carefully, since it can dramatically affect the EA performance.

**Circuit Mutation:**

The circuit mutation allows us to change the following three features of the circuit:

1. Cell input;
2. Cell type;
3. Circuit output.

Each of these parameters is considered as an elementary unit of the genotype. The circuit mutation rate defines how many genes in the population are involved in mutation. The chromosome contains 3 different types of genes, whose number is :

$$N_{genes} = \sum_{i=1}^{\lambda}(3 \cdot N^{i}_{gates} + N_{out}) \tag{4.4.1}$$

where $N_{outputs}$ is the number of outputs in the circuit, $N^{i}_{gates}$ is the number of gates in the $i$-th chromosome, $\lambda$ is the population size.

**Geometry Mutation:**

Geometry mutation allows us to change the number of rows and columns in a chromosome. Geometry mutation is applied to each chromosome with a given probability. In this case the numbers of rows and columns are treated as an elementary unit of the genotype. Either the number of rows or the number of columns is changed with equal probability. The geometry mutation consists of the two main steps: (1) Gene mutation; (2) Repair algorithm. In the first step the new number of columns or rows of the chromosome is randomly defined. At the second step the repair algorithm is applied to ensure that a chromosome with a new geometry represents a valid genotype.

Let us consider geometry mutation process for chromosome with 3x3 circuit geometry. Let $N_{cols}$ and $N_{rows}$ be the number of columns and rows of chromosome assigned to be mutated and *new_value* is the new value of mutated genes chosen randomly. The gene mutation procedure is as follows:

1. Define the circuit mutation rate, $p_{mg}$.

2. Generate random number for each chromosome, $rand1 \in [0, 1]$.

3. If $(rand1 < p_{mg})$ the geometry mutation is applied to the current chromosome.

4. Generate random number $rand2 \in [0, 1]$.

5. **If** ($rand2 < 0.5$) the number of columns in chromosome is chosen to be mutated and the new number of columns ($new\_value$) is generated from the range $[1, N^{max}_{cols}]$. **Else** the number of rows is considered as mutated gene and the new number of rows ($new\_value$) is generated from the range $[1, N^{max}_{rows}]$.

Let $N^{current}$ be the number of rows or columns in chromosome in which the gene is assigned to be mutated and $N^{max}$ is the maximum number of rows or columns which is allowed to be in circuit structure. Then the new value of the mutated genes can be defined using one of the following three strategies:

1. Global geometry mutation (GGM), $new\_value \in [1, N^{max}]$;

2. Bounded geometry mutation (BGM), $new\_value \in [1, N^{current}]$;

3. Local geometry mutation (LGM), $new\_value = N^{current} \pm 1$.

The first strategy allows us to generate a new circuit geometry. The number of columns and rows is randomly defined in the ranges $[1, N^{max}_{cols}]$ and $[1, N^{max}_{rows}]$ respectively. The new number of columns and rows is not related to the current circuit geometry. The second strategy is used to reduce the circuit geometry used in chromosome. The idea of this strategy came from observing that using the global geometry mutation tended to produce circuits with larger circuit geometry. The third strategy is based on the idea of a local search of the circuit geometry. This strategy guarantees to produce comparatively small numbers of new cells in the chromosome in comparison with the first one.

After $new\_value$ is defined, the geometry mutation is performed in the following manner. First, consider the case when the mutated gene is the number of columns.

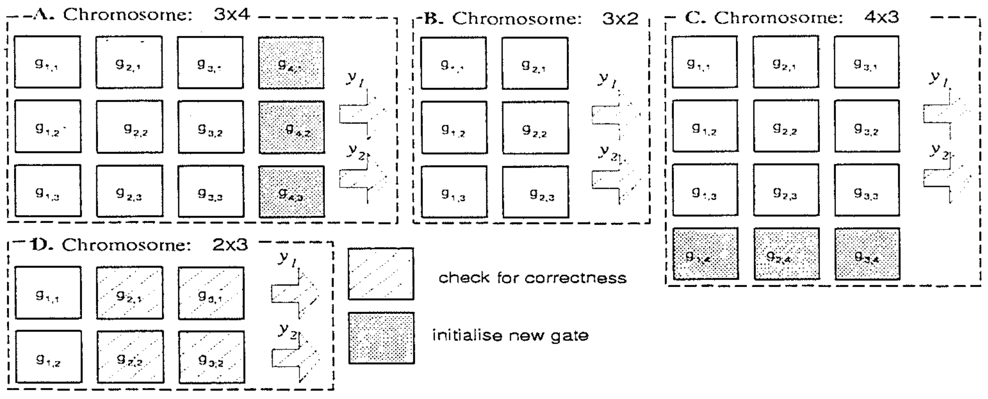


Figure 4.7: The geometry mutation process for a chromosome with geometry 3x3

In this case the new circuit structures, shown in Fig. 4.7 (structures A and B), can be synthesised. If ($new\_value > N_{cols}$), we have to add new columns in the chromosome representation (Fig. 4.7 (structure A)). The gates in the new columns are initialised using the initialisation procedure. It is possible, however, that the circuit output disobeys the levels-back constraint. Thus, the chromosome may need to be repaired. The repair algorithm checks whether the circuit outputs obey the levels-back constraint, and whether all the cell inputs are valid. If the circuit output does not satisfy this condition a new circuit output is initialised. If ($new\_value < N_{cols}$) we have to remove some columns in the circuit structure (Fig. 4.7 (structure B)). After the new structure is obtained, a repair algorithm is applied to the circuit output, because the circuit output can refer to a gate, which no longer exists in the circuit. In the case when the mutated gene is the number of rows, the structures C and D given in Fig. 4.7 can be synthesised. If ($new\_value > N_{rows}$) the new rows of gates are added to the circuit structure (Fig. 4.7 (structure C)). Again, these gates are initialised. There is no need to apply a repair algorithm to the circuit outputs in this case because the connections are not changed and the circuit outputs will still refer to the correct logic cells in the circuit structure. If ($new\_value < N_{rows}$) the last ($N_{rows} - new\_value$) rows are removed from the circuit structure (Fig. 4.7 (structure D)). In this case the inputs of the remaining gates as well as circuit outputs can refer to gates which are no longer present. Therefore each gate genotype and the circuit outputs have to be repaired.

### 4.4.3 Recombination

Recombination is implemented with uniform crossover. For two chromosomes, the uniform crossover generates two new chromosomes by swapping two genes in the chromosomes. Because our chromosome structure contains three levels, on each level the components of the chromosome can be examined like a "gene" or "swapping block". Thus we have three different crossover operators:

1. gene-uniform crossover;
2. cell-uniform crossover;
3. geometry-uniform crossover.

The number of chromosomes selected for breeding is defined by the crossover rate, which is carried out on a cellular level. In order to preserve the interconnection conditions, the repair algorithm checks the inputs of the logic gates for correctness. When two chromosomes with different geometries undergo crossover it is very likely that merely swapping genes to produce the offspring, will generate invalid genomes. These would have to be repaired (randomly initialised), and this would introduce a considerable amount of randomness into the recombination process. Therefore, the selection of the correct crossover rate and its type is very important.

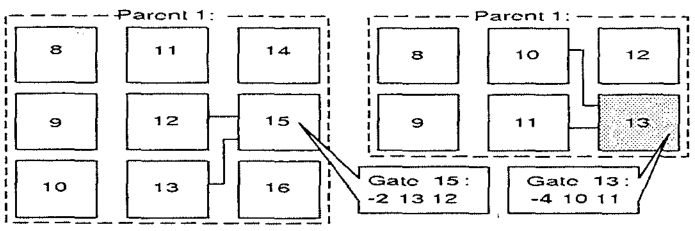


Figure 4.8: Parents for cell-uniform crossover

When we refer to the *gene-uniform crossover*, we mean that any gene of logic cell as well as the circuit outputs can be exchanged. In the case of *cell-uniform crossover*, the data describing the behaviour of a logic gate such as functional gene, inputs, control input, are swapped. In *geometry-uniform crossover*, the columns or rows of logic cells in addition to circuit outputs are involved in the crossover process. In this case a whole column or row of logic gates is swapped and connections are restored if necessary.

Let us consider the "restoring process" in the case when the cells to be swapped belong to chromosomes with different circuit layouts. In this case the cell will refer to different cells in the circuit because of the specific features of encoding. In order to avoid it we correct the cell data in such a way that they refer to the cells positionally located in the same place as with the parents' chromosome. In the case when the cell contains a connection to a non-existent cell, a new connection is randomly generated such that it is valid. Let us consider the case mentioned above with an example of cell-uniform crossover with parent chromosomes with 3x2 and 3x3 circuit geometry and assume the cell to be swapped is located in $2^{nd}$ row and $3^{nd}$ column (Fig. 4.8). Let us also consider the case where the cell from parent 2 is exchanged with cell in parent 1. This cell has connection to the $10^{th}$ and $11^{th}$ cells in the circuit. Positionally it corresponds to the cells located in $2^{nd}$ column and 1st and $2^{nd}$ (10 and 11) rows.

When we exchange this cell in the chromosome with 3x3 circuit geometry, this cell now represents the connections with cells located in $1^{st}$ column and $3^{rd}$ row and in $2^{nd}$ column and $1^{st}$ row. Thus the positional connection is broken. In order to restore it we have to reassign the inputs for this cell according to the labeling process in chromosome. Thus this cell now will be described to $< -4\ 11\ 12 >$. The same process is applied to the cell in parent 1. But in this case the input refers to the cell located in $2^{nd}$ column and $3^{rd}$ row. Because the chromosome where this cell is going to be allocated has only 2 rows (parent 2 has only 2 rows), this input has to be initialised. Thus the "restoring process" allows us to preserve the positional connections of cells and provides a less destructive process.

## 4.5 Experimental results

In this section we will consider some experimental results obtained for the full adder, the two-bit adder and the two-bit multiplier. The main idea of these experiments is to find out which of the EA strategies allows us to determine whether circuit geometry evolution brings some advantages or not.

### 4.5.1 Crossover and mutation strategies

In this series of experiments we have tried to define the best mutation and crossover strategy as well as the best overall GA and ES strategies. We investigate the GA and ES performances for three types of crossover (gene, cell and geometry-uniform crossovers), two types of mutation (circuit and geometry mutations) and three types of geometry mutation (global, boundary and local). We define two main strategies for the evolution and initialisation processes. Each of these strategies is defined by the homogeneous or heterogeneous circuit layout. The initial data are given in Table 4.3. The results obtained for both GA and ES are shown in Table 4.4 - Table 4.7. The tables are organised according to the EA strategies used. The value $\overline{F_2^{bf}(\mathcal{N}_f)}$ defines the mean number of active gates in the fully functional circuits evolved.

The first results were obtained for the case when during initialisation and evolution processes the heterogeneous circuit layout is produced (Table 4.4). The chromosomes with different circuit layouts are produced at each stage of the EA. The circuit and geometry mutations have been used together. Observing the experimental data we found that for both cases ES performs much better then the elite GA. For example, ES with local geometry mutation finds the most compact fully functional one-bit full adders ($\overline{F_2^{bf}(\mathcal{N}_f)} = 5.04$). In this case the largest number of fully functional solutions has been obtained: 45. Similar conclusion can be made about evolving the two-bit multiplier. In this case the ES with global geometry mutation produces 84 fully functional solutions. The ES with boundary geometry mutation evolves the most compact two-bit multipliers ($\overline{F_2^{bf}(\mathcal{N}_f)} = 7.6842$). It is difficult to define which of geometry mutation strategies performs better since for each function different behaviour has been noticed.

A similar conclusion can be made for the behaviour of the elite GA. For instance, the elite GA with cell-uniform crossover produced higher number of fully functional solutions for both functions. Thus, the elite GA with cell-uniform crossover and global geometry mutation engenders 40 fully functional full adders and the same algorithm but with boundary geometry mutation 28 fully functional two-bit multipliers. Note that in this case the evolved logic functions are not the most efficient, that have been evolved using elite GA. The elite GA produces a relatively compact two-bit multipliers

Table 4.3: Initial data: Circuit layout evolution using elite genetic algorithm. $\mathcal{F}_1 + \mathcal{F}_2$ is the dynamic fitness with estimation of the number of active gates in circuit; GGM, BGM and LGM are the global, boundary, local geometry mutations.

| Circuit | add1c.pla | mult2.pla | add1c.pla | mult2.pla |
|---|---|---|---|---|
| **EHW parameters** | | | | |
| Maximum circuit layout, $N_{cols}^{max} \times N_{rows}^{max}$ | 5x5 | | 5x5 | 10x10 |
| Connectivity parameter, $N_{connect}$ | 2 | | 5 | 10 |
| Functional set | $\{-1, -5, -9, -13\}$ | | $\{-1, -5, -9, -13\}$ | |
| Gate distribution | Proportional | | Proportional | |
| Type of layout | Heterogeneous<br>Homogeneous | | Heterogeneous<br>Homogeneous | |
| **EA parameters** | | | | |
| Type of EA | elite GA | | $(1 + \lambda)$ EA | |
| Population size | 15 | | 5 | |
| Number of generations | 1000 | 4000 | 100 | 5000 |
| Number of algorithm runs | 100 | | 100 | |
| Crossover type | Gene-Uniform<br>Cell-Uniform<br>Geometry-Uniform | | -<br>-<br>- | |
| Crossover rate | 0.6 | | - | |
| Selection type | Tournament | | - | |
| Selection pressure | 1.0 | | - | |
| Mutation type | Cell<br>Geometry | | Cell<br>Geometry | |
| Circuit mutation rate | 0.012 | | 0.05 | |
| Geometry mutation rate | 0.05 | | 0.1 | |
| Type of geometry mutation strategy | GGM<br>BGM<br>LGM | | GGM<br>BGM<br>LGM | |
| Fitness type | $\mathcal{F}_1 + \mathcal{F}_2$ | | $\mathcal{F}_1 + \mathcal{F}_2$ | |

Table 4.4: Experimental results: Using circuit mutation and geometry mutation.

| **Crossover Type** | **Geometry Mutation** | **Add1c.pla** $F_1^{bf}$ | **Add1c.pla** $F_2^{bf}(\mathcal{N}_f)$ | **Add1c.pla** $\mathcal{R}(\mathcal{N}_f)$ | **Mult2.pla** $F_1^{bf}$ | **Mult2.pla** $F_2^{bf}(\mathcal{N}_f)$ | **Mult2.pla** $\mathcal{R}(\mathcal{N}_f)$ |
|---|---|---|---|---|---|---|---|
| **Initialisation** | | **Heterogeneous circuit layout** | | | | | |
| **Evolution** | | **Heterogeneous circuit layout** | | | | | |
| **Elite genetic algorithm** | | | | | | | |
| Gene Uniform | Global | 94.8750 | 7.737 | 38 | 91.9687 | 9 | 7 |
| | Boundary | 90.8750 | 8.458 | 24 | 95.28125 | 8.5238 | 21 |
| | Local | 93.6875 | 7.036 | 28 | 95.5 | 8.82 | 23 |
| Cell Uniform | Global | 95.0625 | 8.05 | 40 | 91.9843 | 9.2857 | 7 |
| | Boundary | 92.5625 | 7.5 | 28 | 95.8125 | 8.5 | 28 |
| | Local | 93.6875 | 7.267 | 30 | 94.8906 | 9.0416 | 24 |
| Geometry Uniform | Global | 94.8125 | 8.056 | 36 | 92.2968 | 9.09 | 11 |
| | Boundary | 91.4375 | 7.5 | 20 | 96.0156 | 8.33 | 24 |
| | Local | 94.9375 | 7.697 | 33 | 95.1852 | 9.1852 | 27 |
| $(1+\lambda)$ **Evolutionary Strategy** | | | | | | | |
| - | Global | 94.4375 | 5.5625 | 32 | 99.6875 | 8.2261 | 84 |
| - | Boundary | 75.8750 | 5.875 | 8 | 93.4375 | 7.6842 | 19 |
| - | Local | 96.0000 | 5.04 | 45 | 99.5469 | 7.8375 | 80 |

Table 4.5: Experimental results: Using geometry mutation only, heterogeneous geometry.

| Crossover Type | Geometry Mutation | Add1c.pla $F_1^{bf}$ | Add1c.pla $F_2^{bf}(\mathcal{N}_f)$ | Add1c.pla $\mathcal{R}(\mathcal{N}_f)$ | Mult2.pla $F_1^{bf}$ | Mult2.pla $F_2^{bf}(\mathcal{N}_f)$ | Mult2.pla $\mathcal{R}(\mathcal{N}_f)$ |
|---|---|---|---|---|---|---|---|
| **Initialisation** | | **Heterogeneous circuit layout** | | | | | |
| **Evolution** | | **Heterogeneous circuit layout** | | | | | |
| **Elite genetic algorithm** | | | | | | | |
| Gene Uniform | Global | 79.0000 | 0 | 0 | 82.0625 | 0 | 0 |
| | Boundary | 76.1250 | 10.5 | 2 | 79.6406 | 0 | 0 |
| | Local | 81.0000 | 6 | 2 | 81.4531 | 0 | 0 |
| Cell Uniform | Global | 80.7500 | 0 | 0 | 82.6562 | 0 | 0 |
| | Boundary | 73.3125 | 0 | 0 | 80.1406 | 0 | 0 |
| | Local | 76.2500 | 0 | 0 | 82.0156 | 0 | 0 |
| Geometry Uniform | Global | 83.0000 | 7 | 4 | 84.4688 | 0 | 0 |
| | Boundary | 76.9375 | 10 | 1 | 81.8750 | 0 | 0 |
| | Local | 81.7500 | 9 | 4 | 83.3594 | 0 | 0 |
| **$(1+\lambda)$ Evolutionary Strategy** | | | | | | | |
| - | Global | 66.0625 | 0 | 0 | 78.375 | 0 | 0 |
| - | Boundary | 64.1875 | 0 | 0 | 70.1562 | 0 | 0 |
| - | Local | 69.125 | 0 | 0 | 79.1406 | 0 | 0 |

$(\overline{F_2^{bf}(\mathcal{N}_f)} = 8.33)$ with geometry crossover and boundary geometry mutation and a relatively compact one-bit full adders $(\overline{F_2^{bf}(\mathcal{N}_f)} = 7.036)$ with gene crossover and local geometry mutation. So, we can draw the same conclusion as it has been made above: during evolution with heterogeneous circuit layout at both the initialisation and evolution stages, it is difficult to decide with which of the genetic operators EA produces the best results. In other words, each combination of EA, crossover and geometry mutation behaves differently for both functions. Note that in the experiment described above, both circuit and geometry mutations have been used in evolutionary algorithms.

Now, let us consider the case when only geometry mutation and crossover are

Table 4.6: Experimental results: Using circuit mutation only, homogeneous geometry 4x4(add1c.pla) and 5x5(mult2.pla).

| **Crossover** | **Add1c.pla** | | | **Mult2.pla** | | |
|---|---|---|---|---|---|---|
| **Type** | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}(\mathcal{N}_f)$ | $\mathcal{R}(\mathcal{N}_f)$ | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}(\mathcal{N}_f)$ | $\mathcal{R}(\mathcal{N}_f)$ |
| **Initialisation** | **Homogeneous circuit layout** | | | | | |
| **Evolution** | **Homogeneous circuit layout** | | | | | |
| **Elite genetic algorithm** | | | | | | |
| Gene-Uniform | 95.3125 | 11.024 | 42 | 98.9219 | 7.88 | 51 |
| Cell-Uniform | 93.0625 | 9.364 | 22 | 99.0781 | 7.9824 | 57 |
| Geometry-Uniform | 94.8750 | 11 | 36 | 99.2187 | 7.79 | 58 |
| $(1+\lambda)$ **Evolutionary Strategy** | | | | | | |
| - | 97.875 | 5.9393 | 66 | 99.9062 | 8.84 | 94 |

allowed to be involved in the evolution process. In this case again the heterogeneous circuit layout is produced during both initialisation and evolution stages of evolutionary algorithm. The experimental data obtained for this experiment are given in Table 4.5. The difference between this experiment and the previous one is that the circuit mutation is not used in the evolution process. Firstly, it is interesting to note that we obtain very poor results in terms of the number of 100% cases evolved and no fully functional two-bit multiplier was obtained. Furthermore, we only evolved a few adders. We see that in the case of evolving full adder the worst crossover operator is the cell-uniform crossover. The highest value of $\overline{F_1^{bf}}$ was obtained for geometry-uniform crossover. According to experimental results obtained for this experiment we can conclude that the circuit mutation is the most productive operator that effects on evolutionary algorithm performance. Thus, excluding this operator from the evolutionary process leads to the failure of this approach.

The homogeneous circuit layout is retained for both stages of initialisation and evolution in the next experiment. In this case we are using an entirely homogeneous geometry. The experimental data shown in Table 4.6 are obtained for different types of crossover. Thus, they illustrate the EA performance with circuit mutation only. We found that geometry uniform crossover works best for mult2.pla and gene-uniform crossover for add1c.pla. The best performance, in terms of the minimal number of active gates in the evolved circuit, is obtained with cell-uniform crossover for add1c.pla and with geometry uniform crossover for mult2.pla. This could be explained because of the different complexity associated with the landscapes of these two functions: mult2.pla is considerably more difficult to evolve than add1c.pla. It is interesting to note that using gene uniform crossover produces the better results in terms of the circuit functionality criteria $F_1$ and the number of 100% cases, $\mathcal{R}(\mathcal{N}_f$. Comparing the obtained results with the best results discussed above we can conclude that the pure homogeneous geometry works perfectly well in comparison with heterogeneous one when we need to obtain the maximum number of 100% functional circuits. But in terms of the best average fitness $\overline{F_2^{bf}(\mathcal{N}_f)}$ we found that we evolve poorer circuits. Also, comparing the performances of the elite GA and the ES one can notice that ES produces essentially better results in comparison with the elite GA. Thus we find one of the disadvantages of using the homogeneous geometry: evolving a large number of 100% cases does not provide us with the best solution in terms of the number of active gates used in circuit.

In previous experiments we considered how the algorithm behaves when the same type of circuit layout is applied for both the initialisation and the evolution processes. In the following experiments we will discuss how the evolutionary algorithm behaves if different types of circuit layout are applied during the initialisation and the evolution processes. Table 4.7 illustrates the evolutionary algorithm performance with

Table 4.7: Experimental results: Using circuit mutation only, heterogeneous geometry at the initialisation stage.

| **Crossover Type** | **Add1c.pla** | | | **Mult2.pla** | | |
|---|---|---|---|---|---|---|
| | $F_1^{bf}$ | $F_2^{bf}(\mathcal{N}_f)$ | $\mathcal{R}(\mathcal{N}_f)$ | $F_1^{bf}$ | $F_2^{bf}(\mathcal{N}_f)$ | $\mathcal{R}(\mathcal{N}_f)$ |
| **Initialisation** | **Heterogeneous circuit layout** | | | | | |
| **Evolution** | **Homogeneous circuit layout** | | | | | |
| **Elite genetic algorithm** | | | | | | |
| Gene-Uniform | 92.3125 | 8.292 | 24 | 95.5312 | 9.16 | 25 |
| Cell-Uniform | 94.3750 | 8.634 | 41 | 96.2812 | 8.4615 | 26 |
| Geometry-Uniform | 91.5625 | 8.214 | 28 | 95.6406 | 8.6666 | 27 |
| **$(1+\lambda)$ Evolutionary Strategy** | | | | | | |
| - | 82.1875 | 5.7142 | 14 | 98.03125 | 8.145 | 55 |

heterogeneous circuit layout at the initialisation stage and homogeneous circuit layout at the evolution stage. This means that in this experiment the geometry mutation is not used in the evolutionary algorithm as a genetic operator. Also, the different circuit layout can be defined only at the initialisation. It is interesting to note that in this case, cell-uniform crossover delivers the best performance for the full adder and the geometry uniform crossover dispenses the best performance for the two-bit multiplier. This better performance is obtained in terms of the number of fully functional solutions evolved. Note that the cell uniform crossover brings out the best performance in terms of the average level of circuit functionality evolved for both logic functions. The number of fully functional two-bit multipliers evolved using the elite GA with cell-uniform crossover does not differ a lot from the parameter obtained using the elite GA with geometry-uniform. Therefore we can conclude that in this scheme the cell and the geometry-uniform crossovers behave similar to each other in task of evolving the two-bit multiplier. It is interesting to note that in this case, the elite GA produces a higher number of fully functional solutions evolved rather then ES, as it has been stated in previous experiments. But in the case of evolving two-bit

Table 4.8: Experimental results: Using circuit and geometry mutation, heterogeneous geometry.

| Crossover Type | Geometry Mutation | Add1c.pla $F_1^{bf}$ | Add1c.pla $F_2^{bf}(\mathcal{N}_f)$ | Add1c.pla $\mathcal{R}(\mathcal{N}_f)$ | Mult2.pla $F_1^{bf}$ | Mult2.pla $F_2^{bf}(\mathcal{N}_f)$ | Mult2.pla $\mathcal{R}(\mathcal{N}_f)$ |
|---|---|---|---|---|---|---|---|
| Initialisation | | Homogeneous circuit layout | | | | | |
| Evolution | | Heterogeneous circuit layout | | | | | |
| Elite genetic algorithm | | | | | | | |
| Gene Uniform | Global | 77.625 | 6.6667 | 3 | 96.0312 | 9.25 | 28 |
| | Boundary | 97.5 | 5.9672 | 61 | 99.6406 | 10.0241 | 83 |
| | Local | 96.8125 | 6.22 | 50 | 99.6875 | 9.8765 | 81 |
| Cell Uniform | Global | 75.375 | 7 | 1 | 95.9062 | 10.5 | 38 |
| | Boundary | 97.625 | 6.2096 | 62 | 99.6562 | 9.7073 | 82 |
| | Local | 97.3125 | 6.0345 | 58 | 99.5312 | 10.4615 | 78 |
| Geometry Uniform | Global | 96 | 5.5681 | 44 | 96.0487 | 10.00 | 22 |
| | Boundary | 89.375 | 5.6285 | 35 | 99.5156 | 9.66 | 77 |
| | Local | 97.375 | 5.8 | 61 | 99.5625 | 10.475 | 80 |
| $(1+\lambda)$ Evolutionary Strategy | | | | | | | |
| - | Global | 97.0625 | 5.81481 | 54 | 99.6562 | 8.7024 | 84 |
| - | Boundary | 91.8125 | 5.67742 | 31 | 98.9687 | 8.6986 | 73 |
| - | Local | 97.8125 | 5.73 | 65 | 99.6718 | 9.1765 | 85 |

multiplier, the highest number of fully functional solutions is produced using the ES.

The last experiment was performed using a homogeneous circuit layout at the initialisation stage and a heterogeneous circuit layout at the evolution stage. Note that during the evolution process, only geometry circuit mutation can produce a heterogeneous circuit layout. The experimental results for this experiment are summarised in Table 4.5. These results show that in this scheme the elite GA with the boundary geometry mutation produces higher number of fully functional solutions than others. But at the same time the elite GA with global geometry mutation generates more efficient circuits in terms of the number of active logic gates used. These two statements are valid for both tested logic functions. Similar regular results are obtained for ES performance. Thus, ES with local geometry mutation evolves higher number of fully functional solutions. At the same time ES with boundary geometry mutation induces more efficient logic circuits in terms of the number of logic gates used in the circuit. So, this analysis shows that depending on the type of evolutionary algorithm used, different types of geometry mutation affect differently on algorithm performance. It is also interesting to notice that in the scheme of evolutionary algorithms, clearly some regularities can be noticed, that allows us to draw conclusions about which geometry mutation operator behaves better and under which circumstances. These results are extremely interesting, since they show that the algorithm with this set up of circuit layout performs better than the evolutionary algorithm with a homogeneous circuit layout at all stages. This means that using this scheme of EA construction allows us to produce good results in terms of the number of fully functional solutions obtained and the quality of evolved circuits as well as to evolve the suitable circuit layout for the given problem. This example shows that it is possible to evolve EHW parameters together with the circuit functionality. This is the first attempts to evolve self-adaptive system in terms of adaptation to parameters of EHW.

Analysing the whole experimental data discussed above we can conclude the following.

- The most efficient circuits in terms of the number of primitive active logic gates are generated when the scheme with heterogeneous circuit layout applied to both initialisation and evolution stages of evolutionary algorithm. In this case ES with local geometry mutation generates the most efficient full adder and ES with boundary geometry mutation procreates the most efficient two-bit multiplier.

- The highest number of fully functional solutions has been evolved with homogeneous circuit layouts at both initialisation and evolution stages of evolutionary algorithm. Thus, 94 and 66 successful solutions for two-bit multiplier and one-bit full adder, respectively, have been evolved with ES (see Table 4.6).

- The combination of homogeneous circuit layout at the initialisation stage and heterogeneous circuit layout at evolution stage produces satisfactory results in terms of both criteria: the number of evolved fully functional solutions and quality of evolved circuits.

### 4.5.2 Dynamic fitness strategy in heterogeneous circuit layout evolution

The following experiment shows us how using different fitness evaluation strategies affects the algorithm performance and the quality of evolved circuits. For this purpose, the same experiments were performed for both homogeneous and heterogeneous circuit layouts with and without the dynamic fitness function in the evaluation process. The initial data for this experiment are given in Table 4.9.

The experimental results obtained are summarised in the Table 4.10. An analysis of obtained data shows that the algorithm with the global geometry mutation produces the best results for both functions in terms of the best average functionality criteria $\overline{F_1^{bf}(\mathcal{N}_f)}$ and the number of obtained fully functional solutions $\mathcal{R}(\mathcal{N}_f)$. At the same time the elite GA with boundary geometry mutation generates the most efficient full adders and the elite and the elite GA with circuit mutation only produces the most efficient two-bit multipliers. So, there is no specific combination of evolutionary algorithm and genetic operator that produces the best behaviour in terms of the number of primitive active logic gates. It is also interesting to note that when

Table 4.9: Initial data: Dynamic fitness function and heterogeneous circuit geometry; $\mathcal{F}_1+\mathcal{F}_2$ is the dynamic fitness with estimation of the number of active gates in circuit; GGM, BGM and LGM are the global, boundary, local geometry mutations.

| Circuit | add1c.pla | mult2.pla |
|---|---|---|
| **EHW parameters** | | |
| Maximum circuit layout, $N_{cols}^{max} \times N_{rows}^{max}$ | 5x5 | 1x10 |
| Connectivity parameter, $N_{connect}$ | 2 | 10 |
| Functional set | $\{-1, \cdots, -12\}$ | |
| Gate distribution | Proportional | |
| Type of layout | Heterogeneous<br>Homogeneous | |
| **EA parameters** | | |
| Type of algorithm | GA | ES |
| Population size | 15 | 5 |
| Number of generations, $N_{gen}^{min}, N_{gen}^{max}$ | 500, 5000 | 4000, 25000 |
| Number of GA runs | 100 | |
| Crossover type | Gene-Uniform | - |
| Crossover rate | 0.65 | - |
| Selection type | Tournament | - |
| Selection pressure | 1.0 | - |
| Mutation type | Cell<br>Layout | |
| Circuit mutation rate | 0.012 | 0.05 |
| Layout mutation rate | 0.05 | |
| Fitness type | $\mathcal{F}_1 + \mathcal{F}_2$ | |
| Type of layout mutation strategy | GGM<br>BGM<br>LGM | |

Table 4.10: Experimental results: Strategies of fitness function; $F_1$ and $\mathcal{F}_1 + \mathcal{F}_2$ are the fitness without and with estimation of the number of active gates in circuit respectively; GM is the geometry mutation.

| Mutation Type | $\mathrm{N_{gen}}$ | Fitness Type | **Add1c.pla** | | | **Mult2.pla** | | |
|---|---|---|---|---|---|---|---|---|
| | | | $F_1^{bf}$ | $F_2^{bf}(\mathcal{N}_f)$ | $\mathcal{R}(\mathcal{N}_f)$ | $F_1^{bf}$ | $F_2^{bf}(\mathcal{N}_f)$ | $\mathcal{R}(\mathcal{N}_f)$ |
| **Circuit layout:** | | | **Homogeneous** | | | | | |
| Circuit Mutation | $N_{gen}^{min}$ | $F_1$ | 98.6875 | 10.304 | 79 | 95.734375 | 10.3636 | 11 |
| | $N_{gen}^{min}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | 97.6875 | 10.288 | 66 | 95.875 | 7.8 | 20 |
| | $N_{gen}^{max}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | 99.8750 | 7.582 | 98 | 98.3594 | 7.102 | 97 |
| **Circuit layout:** | | | **Heterogeneous** | | | | | |
| Global GM | $N_{gen}^{min}$ | $F_1$ | 94.7500 | 7.794 | 34 | 95.6719 | 13.6 | 15 |
| | $N_{gen}^{min}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | 94.0625 | 7.517 | 29 | 95.0300 | 11 | 25 |
| | $N_{gen}^{max}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | 99.4375 | 6.077 | 91 | 99.0781 | 9.885 | 82 |
| Boundary GM | $N_{gen}^{min}$ | $F_1$ | 92.1875 | 8.091 | 22 | 92.0312 | 10 | 1 |
| | $N_{gen}^{min}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | 91.0000 | 8.207 | 29 | 92.5000 | 10 | 2 |
| | $N_{gen}^{max}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | 96.9375 | 5.631 | 65 | 96.6562 | 9.727 | 42 |
| Boundary GM | $N_{gen}^{min}$ | $F_1$ | 94.5625 | 7.184 | 38 | 93.4219 | 21 | 3 |
| | $N_{gen}^{min}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | 93.4375 | 7.655 | 29 | 90.5300 | 9 | 4 |
| | $N_{gen}^{max}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | 98.8125 | 5.756 | 82 | 97.5312 | 9.773 | 43 |

we evolve functions during $1,000$ (addlc.pla) or $3,000$ (mult2.pla) generations, we do not achieve significant improvements in terms of the number of active gates in circuit. When we increase the number of generations to $50,000$ it is clear that the average best $F_2$ criteria is improved. Thus, in the case of addlc.pla function we can notice that the size of circuit is slightly reduced, but in case of mult2.pla it is improved only slightly. One of the reasons why we can see only small improvements for the mult2.pla function is that the first GA with $F_1$ only achieves a sufficient number of 100% functional circuits when the number of generations is this large. Therefore the optimising dynamic fitness function $\mathcal{F}_1 + \mathcal{F}_2$ does not have long enough to make a significant difference. It is interesting to note that when we use a homogeneous circuit layout, the average best circuit functionality, $\overline{F_1^{bf}}$ is higher in comparison with the same experiments for a heterogeneous circuit layout. However the average best $F_2$ criteria, $\overline{F_2^{bf}(\mathcal{N}_f)}$ for this case is the lowest one and this does not provide good solutions in terms of the number of active gates. If we consider the algorithm performance in terms of the number of fully functional circuits evolved, it is best to use the homogeneous circuit geometry, but if we use the heterogeneous one we should employ global geometry mutation. Thus, we can conclude that homogeneous circuit layout is useful only in terms of the number of fully functional circuits evolved. At the same time the heterogeneous circuit layout provides better quality circuits in terms of the number of primitive logic gates used. This experiment confirms that using dynamic fitness strategy, $(\mathcal{F}_1 + \mathcal{F}_2)$ allows us to improve the quality of circuits evolved as well.

### 4.5.3 Distributions of circuit layout and circuit functionality

There is one very important aspect that has to be investigated once the circuit layout evolution is introduced. This is how the values of circuit layout change in the final evolved fully functional circuit. Also, the following questions have to be answered, "Has the algorithm a favourite circuit layout for each problem or not?" and "How the circuit functionality parameter, $F_1$ changes with varying the circuit layout?" In order to reply to these questions the following experiment has been set up. The two-bit adder and the two-bit multiplier are chosen as tested logic functions. In order to define the most accurate the distributions of the circuit layout and the circuit functionality, the large number of circuits have to be analysed. For this reason, the two-bit multiplier has been evolved 1000 times and the two-bit adder - 800 times. The initial data are given in Table 4.11. The ES with homogeneous circuit layout at the initialisation stage and heterogeneous circuit layout at the evolution stage (performed using global geometry mutation) evolves successfully 797 fully functional two-bit multipliers and 359 fully functional two-bit adders (see Table 4.12). Some of them are duplicate. The evolved circuit structure of the two-bit multiplier and the two-bit adder are similar to ones discussed in Chapter 3 and Appendix C. For this reason, the evolved circuit structures are not discussed in this Chapter. As it can be seen the average circuit functionality parameter achieved is relatively high.

The collected data have been summarised by defining the following behaviours:

1. The distribution of the circuit layout in fully functional evolved circuits;
2. The distribution of the number of primitive logic gates actually used in the fully functional evolved circuits;
3. The distribution of the circuit functionality in all evolved circuits.

The distribution of the circuit layout in the fully functional evolved circuits is defined as follows. Each tested logic function is evolved using 10x10 circuit layout.

Table 4.11: Initial data: Circuit layout distribution.

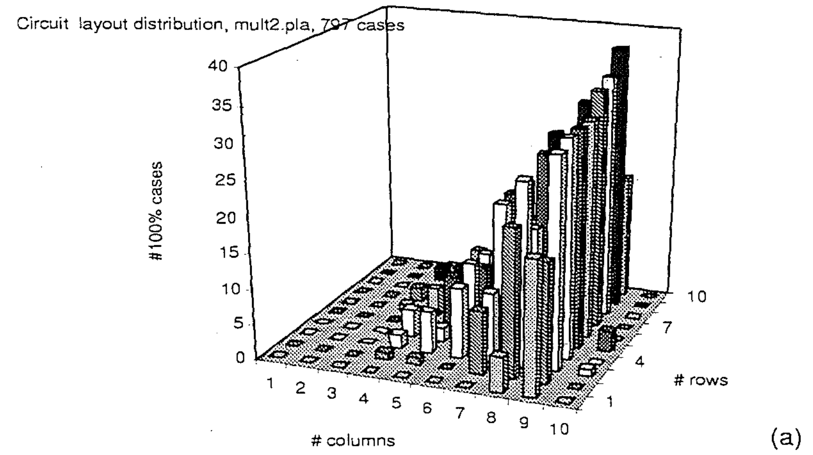
| Circuit | add2c.pla | mult2.pla |
|---|---|---|
| **EHW parameters** | | |
| Maximum circuit layout, $N_{cols}^{max} \times N_{rows}^{max}$ | 15x15 | 10x10 |
| Connectivity parameter, $N_{connect}$ | 15 | 10 |
| Functional set | $\{-1,-5,-9,-13\}$ | |
| Gate distribution | Proportional | |
| Type of layout | Heterogeneous Homogeneous | |
| **EA parameters** | | |
| Type of algorithm | $(1+\lambda)$ ES | |
| Population size, $\lambda$ | 5 | |
| Number of generations, $N_{gen}^{min}, N_{gen}^{max}$ | 15000 | 5000 |
| Number of algorithm runs | 1000 | 800 |
| Mutation type | Cell Layout | |
| Circuit mutation rate | 0.05 | |
| Layout mutation rate | 0.1 | |
| Fitness type | $\mathcal{F}_1 + \mathcal{F}_2$ | |
| Type of layout mutation strategy | GGM | |

Table 4.12: Experimental results: Algorithm performance during investigation of the circuit layout and circuit functionality distributions.

| Logic function | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}$ | $\overline{F_2^{bf}(\mathcal{N}_f)}$ | $\mathcal{R}(\mathcal{N}_f)$ |
|---|---|---|---|---|
| mult2.pla | 99.5563 | 8.364 | 8.0276 | 797 |
| add2c.pla | 97.6663 | 15.3736 | 8.0276 | 359 |

Since the heterogeneous circuit layout has been used in this experiment, each evolved fully functional circuit has its specific circuit layout that has been evolved during evolution. In order to define how many fully functional solutions have been evolved with each specific circuit layout, a 3-D graph has been chosen to represent the data. The $x$ and $y$ axes correspond to the circuit layout and the $z$ axis defines the number of fully functional solutions evolved with given circuit layout. The distributions of the circuit layout for both tested logic functions are depicted in Fig. 4.9. Analysing this data one can easily notice that there is a specific area of circuit layout in which the most of fully functional circuits are evolved. These areas differ for both functions. In the case of evolving two-bit multiplier, no fully functional solutions are generated when the number of columns in the rectangular array is less then 4. The algorithm performance is improved with increasing the number of columns. It is interesting to note that very few logic circuits have been evolved with number of columns equalled 10. Also, it can be noticed that the EA tends to increase the number of rows in the rectangular array. The most fully functional two-bit multipliers are evolved when the circuit layout changed from 7x2 to 9x7. Similar conclusion can be made about the evolution of the two-bit adder. In this case the "productive" circuit layout area changes from 9x4 to 9x13. Comparing this data with the algorithm performance, one can notice the presence of a high level of redundancy in the evolved circuits. This means that redundancy is very important in EHW, since EA tends to use it in the evolutionary process

Another aspect that has to be investigated is how the functionality criteria, $F_1$ of the final circuit at each run is distributed. These data are summarised in Table 4.13. The following conclusion can be made: the lower the circuit functionality the



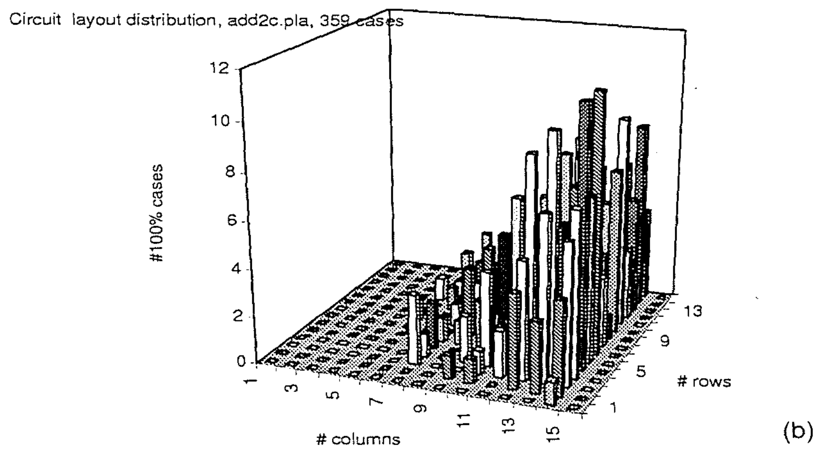


Figure 4.9: The circuit layout distribution The graphs illustrate how the evolutionary algorithm defines the circuit layout automatically and evolves the fully functional solutions. Graphs illustrate that there is a favourite area of circuit layout, where the evolutionary algorithm produces some fully functional solutions. For example, no circuit layout with 3 columns has been chosen during evolution of two-bit multiplier. In the case of the two-bit adder, no circuit layout with less than 6 columns has been chosen by evolutionary algorithm.

Table 4.13: Experimental results: The circuit functionality distribution. Results show that the higher number of circuits has been evolved with higher circuit functionality.

| **Fitness** | $\mathcal{R}(\mathcal{N}_f)$ | |
|---|---|---|
| **function,** $F_1$ | **mult2.pla** | **add2c.pla** |
| 85.4167 | 0 | 2 |
| 86.4583 | 0 | 1 |
| 87.5000 | 0 | 18 |
| 88.5417 | 0 | 2 |
| 89.5833 | 0 | 6 |
| 90.6250 | 0 | 6 |
| 91.6667 | 0 | 16 |
| 92.7083 | 0 | 11 |
| 93.7500 | 7 | 57 |
| 94.7917 | 0 | 9 |
| 95.3125 | 18 | 0 |
| 95.8333 | 0 | 118 |
| 96.8750 | 26 | 51 |
| 97.9167 | 0 | 113 |
| 98.4375 | 153 | 0 |
| 98.9583 | 0 | 32 |
| 100.0000 | 797 | 359 |

lower the number of circuits that evolved with it. For example, 7 two-bit multipliers have been evolved with a circuit functionality of 93.75. At the same time, 18 two-bit multipliers have 95.3125% of correct bits. Similar conclusion can be made about the two-bit adder. But in this case, one can notice the presence of some "favourite" circuit functionality values. Thus, 118 two-bit full adders have been evolved with circuit functionality equal to 95.833 and 113 circuits with $F_1 = 97.9167$. Note that the number of fully functional circuits evolved increases with the compounding number of generations in EA. Therefore, by slightly increasing the number of generations, the circuit functionality in evolved circuit will be increased as well.

The final aspect that has to be studied, once the circuit layout evolution is introduced, is the distribution of the number of active logic gates in evolved circuit. Since the maximum circuit layout used to evolve the two-bit multiplier circuit is 10x10, the maximum number of logic gates that can be actually used in the circuit is 100. In the case of evolving a two-bit adder, a maximum 225 logic gates can be involved in the circuit. Therefore, theoretically the number of logic gates actually used in a circuit can vary from 1 to 100 in case of evolution of the two-bit multiplier and from 1 to 225 in the case of evolution of the two-bit adder. The distribution of the number of active gates in fully functional circuits is depicted in Fig. 4.10. This distribution shows that the circuits are evolved with relatively small number of active logic gates in it. The maximum number of fully functional circuits is obtained for the smallest possible number of active logic gates. Thus, the most efficient two-bit multiplier contains at least 7 primitive logic gates. The number of active logic gates in the two-bit multiplier circuits varies from 7 to 14. No fully functional circuits are evolved with number of active logic gates higher than 15. Similar conclusion can be made for two-bit adder. This means that the proposed methodology of evolving circuit layout together with circuit functionality tends to decrease the number of primitive active logic gates in the circuit even if the average successful automatically generated circuit layout contains relatively high number of logic gates (see Fig. 4.10).

## 4.6 Summary

In this chapter the self-adaptive extrinsic EHW approach is proposed. A number of researches showed in previous work that the EHW performance strongly depends on the initial EHW parameters, such as a functional set of logic gates and circuit

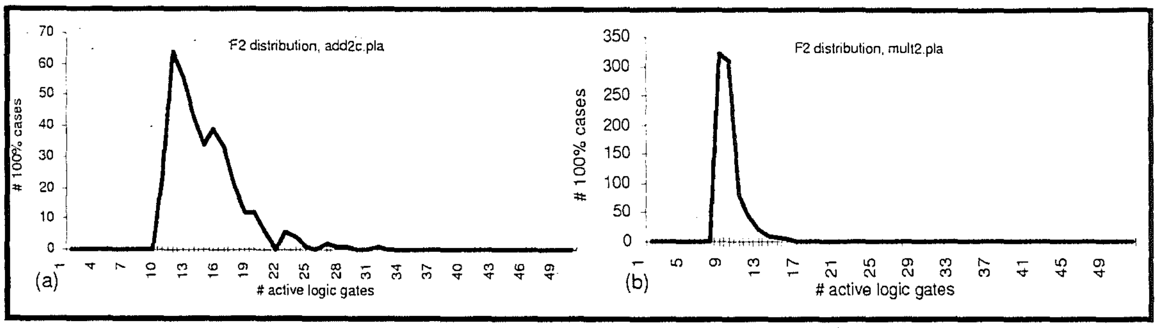


Figure 4.10: The distribution of the number of active logic gates used in circuits evolved using circuit layout heterogeneous approach. The graphs show that there is a specific range of the number of primitive active logic gates, that contain evolved circuits.

layout. In order to overcome this problem, an extrinsic EHW approach that defines the suitable circuit layout during evolution as well as solves the assigned task is proposed. The circuit layout is chosen as one of EHW parameters that can be evolved during the search. This is the first step to design a fully self-adaptive extrinsic EHW that would define all EHW parameters during evolution as well as solve the specified task. This hardware would be able to evolve circuits of any complexity without the intervention of the designer. All initial essential EHW data would be defined during evolution.

The performance of the evolved circuit layout together with circuit functionality and the quality of evolved circuits has been analysed. The following main conclusions can be made as a result of this analysis:

- The most efficient circuits in terms of the number of primitive active logic gates are generated when the scheme with a heterogeneous circuit layout is applied to both the initialisation and evolution stages of the evolutionary algorithm.

- The highest number of fully functional solutions has been evolved with homogeneous circuit layouts at both initialisation and evolution stages of evolutionary algorithm.
- The combination of homogeneous circuit layout at the initialisation stage and heterogeneous circuit layout at the evolution stage produces satisfactory results in terms of both criteria: the number of evolved fully functional solutions and the quality of evolved circuits.

Also it has been empirically shown that applying dynamic fitness function strategies with a large number of generations allows us to obtain better results in terms of the number of active gates used in circuit. This can be explained as follows. Dynamic fitness function forces EA to perform two types of evolutionary processes. The first search is invoked to find a fully functional solution. The second search is invoked to minimise the size of evolved fully functional solutions. Therefore, in this experiment, the number of generations that have been set up was only enough to perform the first part of search. Using a large number of generations allows us to activate the second part of the evolution process and finally minimise the size of evolved circuits. For this reason, EA performed during larger number of generations produces more efficient fully functional circuits in terms of the number of primitive logic gates used.

Another aspect that has to be studied once the circuit layout evolution is introduced is the distribution of the circuit layout in the fully functional circuits evolved. Analysing this data one can easily notice the presence of a high percentage of gate redundancy in the evolved circuits. Since the circuit layout is defined automatically, we can conclude that gate redundancy is very important in the circuit evolution.

The distribution of the number of primitive active logic gates shows that although the EA tends to choose automatically a relatively large circuit layout. There are not a lot of changes in the number of active logic gates in the fully functional circuits evolved. This proves that the EA tends to evolve the circuits with a relatively high percentage of gate redundancy. Therefore, the circuit layouts that are automatically chosen are relatively large in comparison with the number of active logic gates actually used in the circuit.

In this chapter it has been empirically proved that the evolutionary algorithm can automatically generate the most suitable circuit layout for a given problem as well as to solve this problem. This approach is the first attempt to evolve self-adaptive systems in terms of adaptation to EHW parameters.

# Chapter 5

# Function level extrinsic EHW

## 5.1 Introduction

The gate-level EHW has been intensively studied in Chapters 3–4. In this chapter complex functions are constructed using smaller complex building blocks, that have been previously evolved.

From this point of view, the function-level EHW approach can be introduced. In this case, instead of decomposition methods, building blocks described by more complex multi-input multi-output logic functions are involved in the evolution. In function-level EHW, the functional set of logic gates can contain both primitive logic operators and complex logic functions, such as adders, multipliers, etc. These complex logic functions are multi-input multi-output. The chromosome representation adapted in gate-level extrinsic EHW can use only multi-input one-output logic functions as building blocks. This means, that a new chromosome representation, adapted to these features, has to be developed. In this chapter we introduce the new chromosome representation adapted to the function-level extrinsic EHW. The evaluation process performs a similar task to the one represented and discussed in Chapter 3. The new chromosome representation requires the use of the connection repair algorithm at the recombination stage of the evolutionary process. This issue is also discussed in this chapter. Finally, the features of circuit structures evolved at function-level are discussed and as a result of this discussion, the issue of how to define the quality of evolved circuits has emerged. Some suggestions are made in this area as well. Thus, in this chapter a function-level extrinsic EHW is proposed for the first time and some issues associated with this method are discussed.

## 5.2 Chromosome representation

In order to synthesize logic circuits an elite evolutionary algorithm with tournament selection has been implemented. In order to investigate the behaviour of function-level EHW in detail during evolution the circuit layout as well as the input functionality gene are not allowed to changed. Thus, only logic gates with primary inputs were in use. During initialisation, the genotype of chromosomes has been generated randomly.

### 5.2.1 Encoding

In order to define the chromosome representation, the following notations have been adopted:

| | |
|---|---|
| $N_{cols}^{max}, N_{rows}^{max}$ | the maximum number of columns and rows in the rectangular array respectively; |
| $N_{cols}, N_{rows}$ | the number of columns and rows in the rectangular array respectively, $N_{cols} \in \{1, \cdots, N_{cols}^{max}\}$ and $N_{rows} \in \{1, \cdots, N_{rows}^{max}\}$; |
| $N_{connect}$ | the *connectivity parameter* representing the number of columns on the left to which a cell in a particular column $c_{col}$ or an output may be connected and $N_{connect} \in \{1, \cdots, N_{cols}\}$; |

$N_{in}^{max}, N_{out}^{max}$ the maximum number of inputs and outputs in any building block respectively;

$N_{in}(\mathcal{B}), N_{out}(\mathcal{B})$ the number of inputs and outputs in building block $\mathcal{B}$ respectively;

Table 5.1: Gate functionality according to the $b_0(z)$ gene in chromosome

| Gene functionality, $b_0(z)$ | Gate function |
|---|---|
| 0 | Logic constant |
| 2 | $NOT : \overline{x_0}$ |
| 6 | Wire: $x_0$ |
| 7 | AND: $x_0 \cdot x_1$ |
| 8 | OR: $x_0 \vee x_1$ |
| 9 | EXOR: $x_0 \oplus x_1$ |
| 15 | Multiplexer |
| 17 | 1-digit full adder |
| 18 | 2-digit multiplier |
| 19 | 2-digit adder |
| 20 | 3-digit multiplier |
| 21 | 3-digit adder |
| 22 | Half adder |

**Functional set.**

The functional set of all possible logic functions that can be used in evolution is shown in Table 5.1. Let $T_f^{all}$ be the set of integers defining the codes of logic functions reported in Table 5.1. $|T_f^{all}|$ defines the maximum number of logic functions allowed to be used in evolution. We can assign any logic function to describe the behaviour of building block. In a given case we chose arithmetic functions such as half adders, full adders or multipliers. Any of the logic functions mentioned in Table 5.1 can be involved in the evolution. The set of logic functions actually used in evolution and encoded as shown in Table 5.1 is defined as follows: $T_f = \{t_f : t_f \in T_f^{all},\ \text{some}\ t_f\}$.

$|T_f|$ determines the number of primitive and standard logic functions involved in evolution that are used to define the behaviour of a building block.

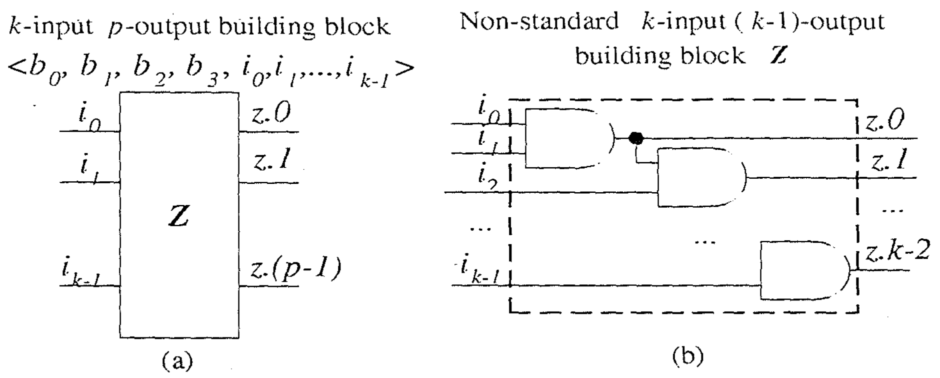
We specify subsets of logic functions that influence the number of inputs and outputs in building blocks by different way as follows:

1. the subset of 1-input, 1-output functions (unary operators), $T_f^1$;
2. the subset of 2-input, 1-output functions (primary operators), $T_f^2$;
3. the subset of multi-input multi-output *standard* functions, such as half adder, full adders, multipliers, $T_f^3$.

Based on the notations given above, we can summarise:

$$T_f = T_f^1 \cup T_f^2 \cup T_f^3, \quad T_f^1 \cap T_f^2 \cap T_f^3 = \emptyset. \tag{5.2.1}$$

The behaviour of a building block can be represented by any of the logic functions mentioned above or by the set of 2-input 1-output functions connected interactively (Fig. 5.1 (b)). Let us consider a building block $\mathcal{B}$ that can implement any logic function from subsets $T_f^1, T_f^2$ or $T_f^3$. Thus if $\mathcal{B}$ implements the logic function from subset $T_f^1$, then $N_{in}(\mathcal{B}) = 1$, $N_{out}(\mathcal{B}) = 1$. In the case of using 2-input, 1-output logic functions ($T_f^2$), the number of inputs can be variable and can be defined as follows: $N_{in}(\mathcal{B}) \in T_{in} = \{2, \cdots, N_{in}^{max}\}$, $N_{out}(\mathcal{B}) \in T_{out} = \{1, \cdots, N_{in}(\mathcal{B}) - 1\}$. For example, if $\diamondsuit$ defines the 2-input 1-output logic primitive function and $N_{in} = 4$, then $\{i_0, i_1, i_2, i_3\}$ and $\{o_0, o_1, o_2\}$ are the set of inputs and outputs in building block belonging to $T_f^2$ respectively. The number of outputs in a building block is $N_{out} = N_{in} - 1 = 3$. The building block has more than one output since each logic gate in the building block can be connected to the output. These outputs can be analytically represented as



$k$-input $p$-output building block
$<b_0, b_1, b_2, b_3, i_0, i_1, \ldots, i_{k-1}>$
$i_0$
$i_1$
$i_{k-1}$
Z
z.0
z.1
z.(p-1)
(a)
Non-standard $k$-input ($k$-1)-output building block Z
$i_0$
$i_1$
$i_2$
...
$i_{k-1}$
z.0
z.1
...
z.k-2
(b)

Figure 5.1: Building block level representation

follows:

$$
\begin{aligned}
o_0 &= i_0 \Diamond i_1; \\
o_1 &= (i_0 \Diamond i_1) \Diamond i_2; \\
o_2 &= ((i_0 \Diamond i_1) \Diamond i_2) \Diamond i_3.
\end{aligned}
$$

When we use standard logic functions defined in $T_f^3$, the number of inputs and outputs are fixed and can not be changed.

So, we can define the relation between type of logic function chosen to describe the behaviour of building block **b** and the number of inputs and outputs in $\mathcal{B}$ as follows.

$$
\begin{aligned}
\{(t_f, t_{in}, t_{out}) : \quad & (t_f \in T_f^1 \wedge t_{in} \in \{1\} \wedge t_{out} \in \{1\}) \vee \\
& (t_f \in T_f^2 \wedge t_{in} \in T_{in} \wedge t_{out} \in T_{out}, \text{ some } t_{in}, \text{ some } t_{out}) \vee \\
& (t_f \in T_f^3 \wedge T_f^3 \to T_{in} \wedge T_f^3 \to T_{out})\}. \qquad (5.2.2)
\end{aligned}
$$

### Building block level representation.

Let us consider a building block $\mathcal{B}_z$ labeled as $z$. Let $T_i$ be the set of integers $\{0, 1, \cdots, 2^{N_{in}^{max}}\}$. Let $N_{out}(\mathcal{B}_z)$ be the number of outputs of building block and $N_{out}^{max}$

is defined to be no more than 10. Define $V(c_{col})$ as the set of real numbers $v$ such that

$$\begin{aligned} c_{col} > N_{connect}: \quad & a_{min} = n + (c_{col} - N_{connect}) * N_{rows}; \\ & a_{max} = n + c_{col} * N_{rows} - 1 \\ & a_{min} \le v \le a_{max} + 0.1 * N_{out}(\mathcal{B}_{a_{max}}) \\ c_{col} \le N_{connect}: \quad & a_{min} = 0; \;\; a_{max} = n + c_{col} * N_{rows} - 1; \\ & a_{min} \le v \le a_{max} + 0.1 * N_{out}(\mathcal{B}_{a_{max}}) \end{aligned}$$

Any building block $\mathcal{B}_z$ in $c_{col}$ column and $c_{row}$ can be represented as follows:

$$\mathcal{B}_z = < b_0 \;\; b_1 \;\; b_2 \;\; b_3 \;\; \mathbb{I} >,$$

where $b_0 \in T_f$ is the *building block functionality gene*, which defines the type of building block $\mathcal{B}_z$; $b_1 \in T_i$ is the *input functionality gene* that determines the type of inputs in building block $\mathcal{B}_z$; $b_2$ and $b_3$ are the number of outputs and inputs in building block; the set $\mathbb{I} = \{i_0, i_1, \cdots, i_{N_{in}^{max}-1}\}$ defines the connections between building blocks, $\mathbb{I} = \{i \in V(c_{col}), \text{ some } i\}$. The genes $b_0, b_2$ and $b_3$ are calculated according to Eq. 5.2.2. Thus relation $(b_o \;\; b_3 \;\; b_2)$ is determined in Eq. 5.2.2. The gene $b_1$, that is an integer, defines the type of inputs used in building block. The lowest bit corresponds to the input $i_0$ and highest bit corresponds to the input $i_{N_{in}^{max}}$. If corresponding bit is zero the input is primary, otherwise the input is inverted. For example, if $b_1 = 11$, that corresponds to the binary sequence $< 1011 >$. This sequence can be encoded as $< i_3 \;\; i_2 \;\; i_1 \;\; i_0 >$. Then the inputs $i_0, i_1$ and $i_3$ will be considered as primary and the input $i_2$ will be inverted.

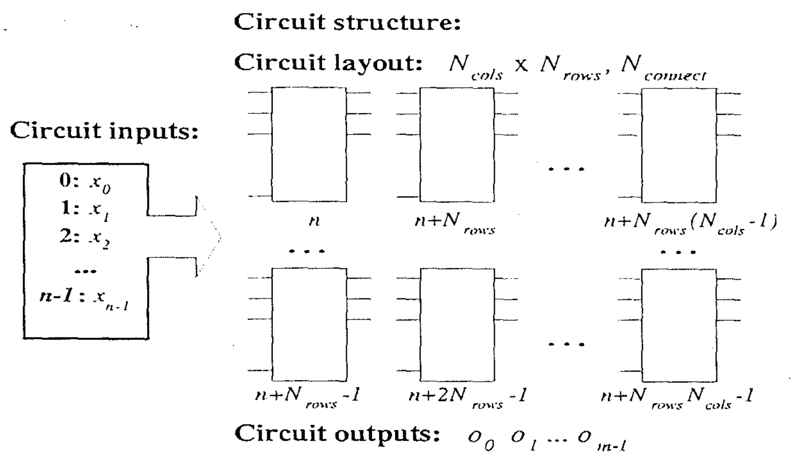


Figure 5.2: Circuit level representation

## Circuit level representation.

Let us consider the rectangular array $\mathbb{B}$ of the logic building blocks $\{\mathcal{B}_{c_{col}c_{row}} : \{\mathcal{B}_{c_{col}c_{row}} \in \mathbb{B},\ c_{col} = \{0, \cdots, N_{cols} - 1\},\ c_{row} = \{0, \cdots, N_{rows} - 1\}\}\}$. The building block $\mathcal{B}_{c_{col}c_{row}}$, located in column $c_{col}$ and row $c_{row}$, is labeled by integer $(n + N_{rows} * c_{cols} + c_{rows})$. For example, if a 4-input logic function is evolved using 3x4 circuit layout, the building block $\mathcal{B}_{00}$ located in 0-th column and 0-th row is labeled as 4, $\mathcal{B}(4)$. The building block $\mathcal{B}_{23}$ located in 2-nd column and 3-rd row is labeled as 15, $\mathcal{B}(15)$.

Let us consider how the connectivity parameter influences the circuit structure. For the first column of building blocks in the chosen geometry, the inputs to the actual building blocks may only be connected to the actual circuit inputs, i.e. the inputs of logic function implemented. However provided that the connectivity parameter $N_{connect}$ is greater than 1 the building blocks from the second column can be connected to the outputs of building block from the first column as well as to the circuit inputs. If $N_{connect}$ had been chosen to be 1, the building blocks from the second column can be only connected to the outputs of building blocks from the first column. In case when $N_{connect} = N_{cols}$ the building blocks can be connected to any outputs of building blocks located to the left or to the circuit inputs. Decreasing $N_{connect}$ has the effect of reducing the number of possible circuit solutions that may be found.

Let $\mathbb{O}$ be the set of integers such that $\{o : o \in V(N_{cols})\}$, $|\mathbb{O}| = m$. The set $\mathbb{O}$ defines the circuit outputs. Therefore the circuit genotype can be represented as follows:

$$C =< N_{cols}\ N_{rows}\ N_{connect}\ \mathbb{B}\ \mathbb{O} > . \tag{5.2.3}$$

**Genotype.**

The value $g(x)$ at position $x$ (measured from the left and starting at 0) is chosen as follows:

**Circuit layout**

$x = 0 \quad g(0) = N_{cols} \in \{1, \cdots, N_{cols}^{max}\}$

$x = 1 \quad g(1) = N_{rows} \in \{1, \cdots, N_{rows}^{max}\}$

$x = 2 \quad g(2) = N_{connect} \in \{1, \cdots, N_{cols}\}$

**Building block**

$0 \leq x < |\mathbb{B}| * (4 + |\mathbb{I}|) \quad (g(x)\ g(x+2)\ g(x+3))$, if $(x-3) \bmod (4+|\mathbb{I}|) = 0$;

$g(x) \in T_i$, if $(x-3) \bmod (4+|\mathbb{I}|) = 1$;

$g(x) \in V(c_{col})$, if $(x-3) \bmod (4+|\mathbb{I}|) = 4, \cdots, 3+|\mathbb{I}|$;

**Circuit outputs**

$|\mathbb{B}| * (4 + |\mathbb{I}|) \leq x \quad g(x) \in V(N_{cols})$

g240

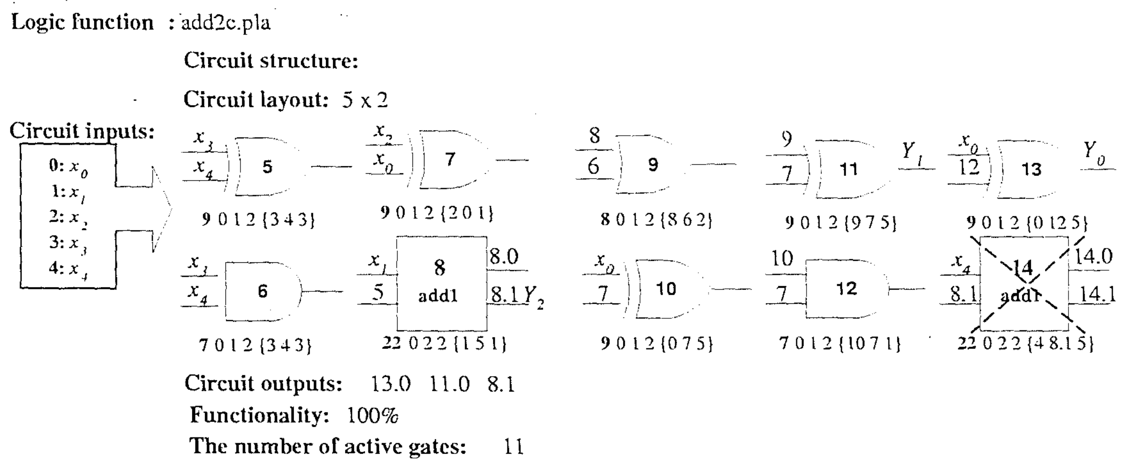


Figure 5.3: An example of the phenotype and corresponding genotype of a chromosome with 5x2 circuit layout The number of building blocks employed is 9. The logic gate labeled 8 implements a 1-digit full adder, that requires 3 primitive logic gates to be implemented. Therefore, the number of primitive logic gates is 11.

### An example.

An example of chromosome representation with the actual circuit structure is given in Fig. 5.3. This circuit represents a 2-bit adder evolved using AND, OR, EXOR and half adder binary logic functions. This function has 5 inputs and 3 outputs and is implemented here on a combinational network with 5x2 circuit layout ($N_{cols} \times N_{rows}$). The labels of circuit inputs 0, 1, 2, 3, 4 correspond to the input variables $x_0$, $x_1$, $x_2$, $x_3$ and $x_4$ respectively. We use an encoding table to define the type of building block (i.e. functional gene that is shown in bold) (see Table 5.1).

Each cell is assigned an individual address. Thus the building block located in 0th column and 0th row is labeled as 5. The building block located in 4th column 1st row is labeled as 14. Each output of building block is labeled with a real number. The integer part of this number defines the code of building block and the fractional part determines the position of output in building block. For example, the 8th building block $\mathcal{B}_8$ located in 1st row and 1st column has 2 outputs. The first output is numbered as 8.0 and the second one as 8.1. The number of circuit outputs is defined by the number of outputs in the logic function implemented. Let us examine the encoding of the 14th building block represented as $<$ 22 0 2 2 {4 8.1 5} $>$. We refer to this representation of the building block as *building block genotype*. The functional gene defining the type of this gate is 22. This corresponds to the half adder in encoding table (Table 5.1). The examined cell has two inputs and two outputs. The first input is connected to the input $x_4$ and second to the second output of building block 8, labeled as 8.1. The inputs of building block 8 are connected to input variable $x_1$ and output of building block 5. The logic function of building block 5 depends on the input variables $x_3$ and $x_4$. Therefore the logic function implemented in building block 14 depends on three input variables: $x_1$, $x_3$, and $x_4$. The outputs of the circuit are connected to the outputs of building blocks 13, 11 and 8.1.

### 5.2.2 Fitness Function

We use a dynamic fitness strategy. First we are trying to find the circuit with 100% functionality ($F_1$) and second we are trying to minimise the number of active gates in a functionally complete circuit ($F_2$). Thus, $F_1$ rewards the circuits which have the correct digits in the correct positions for the circuit outputs. $F_2$ adds a reward for the 100% functional circuits that have a minimal number of building blocks.

To present these fitness functions more formally, we need some definitions. Let function $F(X)$ be represented as a Boolean matrix mapping denoted as $X \longrightarrow Y$, where $X$ is a $(2^n \times n)$ matrix of all the given $n$-variable inputs; $2^n$ is the number of input combinations, and $Y$ is a $(2^n \times m)$ matrix of the corresponding $m$ outputs.

Then the synthesis of binary functions can be stated as follows. Design a sequence of operations that accomplish the mapping $X \longrightarrow Y$. This mapping is achieved by applying a sequence of primitive operations. In our case the sequence of primitive operations is defined in the rectangular array of building blocks. Let $\mathcal{N}(x)$ be the output of a network $\mathcal{N}$ on the input combination $\mathbf{x}$, where $\mathbf{x}$ is the $n$-digit binary vector whose individual digits are the inputs to $\mathcal{N}$. Suppose that $C$ is a correct circuit, so that $C(\mathbf{x}) = m * 2^n$. Since $m * 2^n$ is the number needed to have the digits in $\mathbf{x}$ correctly sorted for output combination with $m$ digits.

We use the dynamic fitness function strategy $\mathcal{F}_1 + \mathcal{F}_2$ in order to evaluate the evolved circuits.

Our first component of dynamic fitness function $\mathcal{F}_1$ returns the number of correctly sorted digits over all inputs in $X$. Let $\Delta(\mathbf{x_n}, \mathbf{y_n})$ be the number of digits in $\mathbf{x_n}$ and $\mathbf{y_n}$ which agree with each other (the opposite of Hamming distance between $\mathbf{x_n}$ and $\mathbf{y_n}$), where $x_n$ and $y_n$ are binary vectors of $n$ elements. For example, $\Delta((110100),(101010)) = 2$. Then we can formally define our first fitness function as

$$\mathcal{F}_1(\mathcal{N}) = \frac{\sum_{\mathbf{y_l} \in Y} \sum_{\mathbf{x_n} \in X} \Delta(C(\mathbf{x_n}), N(\mathbf{x_n}))}{m * 2^n} * 100. \tag{5.2.4}$$

This definition implies that if $\mathcal{F}_1(\mathcal{N}) = 100\%$, the circuit evolved is correct, i.e. it is a fully functional circuit $\mathcal{N}_f$.

The second component of dynamic fitness function $\mathcal{F}_2$ defines the number of primitive logic cells unused in the circuit. The *cost* or size of the fully functional circuit $\mathcal{N}$ is defined as

$$cost(\mathcal{N}) = \sum_{j=0}^{j<N_c*N_r-1} cost(\mathcal{B}_j) \tag{5.2.5}$$

and the cost of the building block $\mathcal{B}_j$ is calculated as

$$cost(\mathcal{B}_j) = \begin{cases} N_j^p, & \mathcal{B}_j \text{ is a committed building block} \\ 0, & \mathcal{B}_j \text{ is an uncommitted building block.} \end{cases} \tag{5.2.6}$$

where $N_j^p$ is the minimum number of primitive logic cells required to implement the logic function describing the behaviour of building block $\mathcal{B}_j$.

The maximum cost of network can be calculated as follows:

$$cost(\mathcal{N}^{max}) = \sum_{j=0}^{j<N_c*N_r-1} N_j(\mathcal{B}_{complex}) \tag{5.2.7}$$

where $N_j(\mathcal{B}_{complex})$ is the minimum number of primitive logic cells required to implement the most complex logic function from the functional set of logic gates used.

The number of primitive logic cells unused in circuit can be calculated as follows:

$$F_2 = cost(\mathcal{N}^{max}) - cost(\mathcal{N}). \tag{5.2.8}$$

We consider the building block $\mathcal{B}_j$ as a sub-circuit with the structure that is not allowed to be changed. So, the cost of building block does not take into account whether all outputs of building block have been involved or not. For example, let the two-bit multiplier be represented as building block $\mathcal{B}_j$ and the first digit of this two-bit multiplier be *only* involved in the circuit $\mathcal{N}$. The cost of the two-bit multiplier is 7 [25]. Although the first digit of two-bit multiplier is implemented using only one primitive logic gate, the cost of building block $\mathcal{B}_j$ is 7. Members of the population with a changed genotype have their fitness calculated.

Note, that fitness $\mathcal{F}_2$ is activated, when $\mathcal{F}_1 = 100\%$. The dynamic fitness function is calculated as follows:

$$\mathcal{F} = \begin{cases} \mathcal{F}_1, & \mathcal{F}_1 < 100; \\ \mathcal{F}_1 + \mathcal{F}_2, & \mathcal{F}_1 = 100. \end{cases}$$

For example, the fitness function of the circuit shown in Fig. 5.3, can be defined as follows. The circuit implements a full adder completely, thus $F_1 = 100\%$. Therefore we have to compute the number of primitive logic cell actually used in the circuit. Analysing the connectivity of basic blocks, we define that building block $\mathcal{B}_{14}$ is not in use. Thus, there are 9 building blocks actually involved in the circuit implementation. Note that $\mathcal{B}_8$ describes the half adder, that can be implemented using 3 logic cells. Therefore, the number of active primitive logic cells required to implement this circuit is 11. The maximum number of logic cells required to implement the most complex building block is 3, thus $F_2 = 3 * 5 * 2 - 11 = 19$ and $\mathcal{F} = \mathcal{F}_1 + \mathcal{F}_2 = 119$.

## 5.3 Connection repair algorithm

A two-level representation for connection gene has been introduced in a previous section. As a result of the mutation operator, some repair algorithms have to be designed in order to restore the connections. The genotype can be incorrect if the functionality of any logic gate has been changed. This happens because the introduced functional set of logic gates contains a different type of logic gates with different number of outputs. Changing the number of outputs in the logic cell is essential, because the connection genes associated with this particular cell depend on the number of outputs in this cell. Therefore, once the number of outputs in the logic cell has been changed, the gate input connections and circuit output connections have to be checked for correctness. In the evolutionary algorithm applied to the function-level extrinsic EHW there is only one "destructive" operator - mutation. As it has been mentioned above, changing only one type of gene - the functionality of the logic gate - has a destructive effect on the correctness of the genotype. Therefore, because of the mutation of the functionality gene, it is necessary to introduce repair algorithms for:

1. circuit output connection;

2. gate input connection;

3. gate output connection.

It should be remembered that the primary gate connection refers to the label of the logic gate and the secondary gate connection refers to the label of the output in the logic gate under consideration. Only the value of the secondary gate connection can cause the destruction of genotype correctness. Therefore, it is sensible to check only the secondary connection of the gate in the genes mentioned above.

The connection repair algorithm is activated once the number of outputs in logic cell has been changed. In the case when the functionality has been changed, but the number of outputs in the cell remains the same, the connection repair algorithm is not activated. Once the necessity of using the connection repair algorithm is validated, the circuit output connection and the gate input connection are checked for correctness. There is no need to check all connection genes in the chromosome, since the connection parameter $N_{connect}$ defines the range of logic gates to which the outputs of the logic gate at hand can be connected. Therefore, it is sensible to check whether the connection genes are correct or not only in the logic gates located in columns labeled from $N_{current} + 1$ to $N_{current} + N_{connect}$. If the number of columns located to the right from the current column is less than $N_{connect}$, the circuit output connections have to be checked for correctness as well. If the connection gene is not correct, a new value is assigned to it. In this case, the reference to the logic gate remains the same and a new value of gate output is assigned.

More formally, this procedure can be described as follows: Let $z.j$ be the value of connection gene, that has failed and $\mathcal{B}_z(N_{out})$ be the number of outputs in the logic gate $z$. If $j \geq \mathcal{B}_z(N_{out})$, a new value for this gene is randomly defined from the range $j \in \{0, ..., \mathcal{B}_z(N_{out}) - 1\}$.

For instance, let us consider the circuit depicted in Fig. 5.3. Assume that the functionality of the 8-th logic cell has been changed from half adder to AND gate. In this case, the number of outputs in the logic gates is changed from 2 to 1. Let us remember, that $N_{connect} = N_{cols} = 4$ for this circuit. This means, that we have to check all connection genes located in columns 3-5, i.e. all logic gates labelled from 9 to 14 and all circuit output genes. Let us consider the first connection gene in logic cell 9 that is equal to 8.0. This means that this input is connected to the 0th output of the 8th logic cell. Since $0 < 1$, (1 is the new number of outputs in the 8th logic cell), this value is correct and the connection repair algorithm is not activated. Now let us consider the 2nd input of the logic cell labelled 14. This input is equal to 8.1. Obviously, there is no output labelled 1 in the logic gate 8. So, this gene is incorrect and has to be changed to value 8.0 since this gate has currently only one output. The same changes have to be made to the 3rd circuit output gene which is equal 8.1.

In this section we discussed the necessity to introduce the connection repair algorithms and explained them in detail. The chromosome genotype introduced in this chapter can be incorrect, if the connection repair algorithm is not activated.

## 5.4 Experimental results

The main point of these experimental results is to define how the gate-level extrinsic EHW differs from the function-level EHW.

### 5.4.1 Algorithm performance

The main idea of this experiment is to determine whether function-level chromosome representation brings some advantages or not to circuit evolution. The initial data for the experiments is given in Table 5.2.

Each experiment has been performed 100 times. The functional set of logic gates contains both 2-input and complex logic functions. The choice of complex logic functions used in evolution is based on a priori knowledge of combinational circuit design and on the knowledge about the functionality of building blocks (in the case of FPGA-based circuit design). For example, adders and multipliers can be used as basic blocks in the FPGA design. In this series of experiments we attempt to evolve such arithmetic logic functions as a two-bit adder (add2c.pla), a two-bit multiplier (mult2.pla) and a three-bit multiplier (mult3.pla). Based on a priori knowledge of circuit design we can conclude that these arithmetic functions can be easily synthesised using such basic building blocks as half and full adders. Consequently, the results obtained using the half adder and full adder as a basic block are not so surprising. At the same time, no knowledge about using a two-bit multiplier in the synthesis of a three-bit multiplier has been found. Intuitively, one can assume that because multipliers are built using the same construction rules, high-order multipliers can be synthesised using low-order multipliers. Therefore, some of our functional sets of logic gates contain the two-bit multiplier when the evolution of the three-bit multiplier is performed.

In this section we will discuss some experimental results obtained during evolution of logic circuits at gate- and function-level. The EA performs a fixed number of generations for both approaches. The functional set of logic gates for gate-level EHW is a subset of {AND, OR, EXOR, NOT}. Note that functional set of NOT, AND, OR,

Table 5.2: Initial data

| Circuit | mult2.pla | mult3.pla | add2c.pla |
|---|---|---|---|
| **EHW parameters** | | | |
| Circuit layout | 10x1 | 30x1 | 15x1 |
| Connectivity parameter | 10 | 30 | 15 |
| **EA parameters** | | | |
| Type of algorithm | $(1+\lambda)$ ES | | |
| Population size | 5 | | |
| Number of generations | 5 000 | 100 000 | 15 000 |
| Number of algorithm runs | 100 | | |
| Mutation type | Circuit Mutation | | |
| Mutation rate | 5% | | |

EXOR(i.e. 2-7-8-9 according to encoding table (Table 5.1)) corresponds to execution of gate-level EHW. In this case the number of inputs in a building block can not be more then 2. In the case of function-level EHW, a half bit adder, one-bit multiplier and two-bit multiplier have been added to the main functional set. The experimental results obtained are shown in Table 5.3. The functional set of logic functions contained the complex functional blocks are shown in bold.

Let us consider the experimental results obtained for the two-bit multiplier. Analysing the obtained data we can conclude that in terms of the number of active primitive logic gates used in circuit, the gate-level EHW performs better. But in terms of the number of fully functional circuits evolved during 100 EA runs, both methods perform nearly the same. In case of evolving a three-bit multiplier and a two-bit adder it is clear that the function-level EHW performs much better. Thus, no fully functional three-bit multipliers have been evolved using gate-level EHW. But using function-level EHW, we were able to produce some functional solutions. It has been reported that the fully functional circuit of the three-bit multiplier can be evolved at gate-level after only from $3,000,000$ up to $10,000,000$ generations with the initial

Table 5.3: Experimental Results. Functional sets shown in bold correspond to the function-level EHW approach, otherwise - to gate-level EHW.

| **Circuit** | $n$ | $m$ | **Functional set** | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}$ | $\overline{F_1^{bf}(\mathcal{N}_f)}$ | $\mathcal{R}(\mathcal{N}_f)$ |
|---|---|---|---|---|---|---|---|
| mult2.pla | 4 | 4 | 2-7-8-9 | 98.2968 | 7.22 | 7.14 | 36 |
| | | | **2-7-8-9-22** | 98.2031 | 11.56 | 8 | **37** |
| | | | **2-7-8-9-17** | 97.7344 | 15.51 | 8.75 | **16** |
| | | | **2-7-8-9-17-22** | 98.3438 | 15.83 | 10.9355 | **31** |
| mult3.pla | 6 | 6 | 2-7-8-9 | 95.5686 | 32.2667 | 0 | 0 |
| | | | **2-7-8-9-18** | 97.1807 | 59.6087 | 33.3 | **3** |
| | | | **2-7-8-9-18-22** | 98.0547 | 58.8 | 42 | **5** |
| | | | **2-7-8-9-17-18-19-22** | 99.5734 | 95.1525 | 65.33 | **6** |
| add2c.pla | 5 | 3 | 2-7-8-9 | 93.75 | 10.25 | 11.1429 | 14 |
| | | | **2-7-8-9-22** | 93.9167 | 12.62 | 12.5833 | **24** |
| | | | **2-7-8-9-17-22** | 99.6042 | 10.69 | 10.4375 | **96** |
| | | | **2-7-8-9-17-18-22** | 99.7708 | 11.74 | 11.2128 | **94** |

parameters used in this series of experiments [9], [116]. In our case the fully functional solutions have been evolved after 100,000 generations. This means that the function-level EHW requires in 30-100 times less number of generations in order to obtain the fully functional solution. The same effect was obtained for the two-bit adder. Using a function-level EHW approach allows us to evolve fully functional solutions easier than using gate-level EHW approach.

Let us compare the average best functionality fitness functions $(\overline{F_1^{bf}})$ for the logic functions under consideration. It is clear that the best functionality fitness function is lower when a gate-level EHW has been applied. Analysing the average number of active primitive logic gates in the best fully functional chromosome $(\overline{F_2^{bf}(\mathcal{N}_f)})$, we find that there is no significant difference between function- and gate-level EHW. Consequently, we can conclude that function-level EHW performs better than gate-level EHW in terms of the number of fully functional binary circuits evolved.

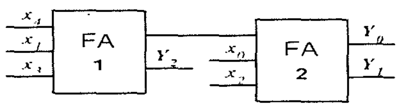


Figure 5.4: Evolved two-bit adder design (**A**); # logic gates = 10; # building blocks = 2; HA is half adder.

## 5.4.2 Evolved circuits

In this section we will discuss the specific features of circuit structures evolved using function-level extrinsic EHW with initial data given in Table 5.2.

### A two-bit adder

The conventional structure of the two-bit adder synthesised using one-bit full adders is depicted in Fig. 5.4 (design A). This structure requires 10 primitive logic gates. This is a well-known circuit that can be found in any handbook on circuit design foundations [129]. In order to investigate whether EA will learn well-known human design techniques or not, the two-bit full adder has been evolved using one-bit full adder as one of the elements in the functional set of logic components. The analysis of circuit structures evolved shows that 90% of fully functional two-bit adders had the circuit structure shown in Fig. 5.4. This fact proves that the EA can find the optimal circuit structure designed by a human. The circuit structures synthesised without an one-bit full adder appeared in the remaining 10% of all fully functional circuits evolved. This shows that the EA exploring the search space does not converge to the only optimal solution. This experimental result illustrates that in order to evolve the most optimal circuit design, EA can be be executed several times.

Another aspect of our curiosity in the circuit structures evolved lies in the area of using non-standard logic functions as building blocks in order to evolve circuits.

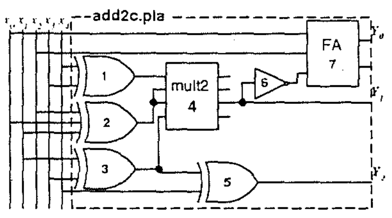


Figure 5.5: Evolved two-bit adder design (**B**); # logic gates = 18; # building blocks = 7; FA is full adder; mult2 is a two-bit multiplier.

In this case, the two-bit multiplier has been added to the functional set of logic components in order to evolve two-bit adder. One of the most common structures that appeared in this series of experiments is illustrated in Fig. 5.5 (design B). This circuit requires 18 primitive logic cells or 7 building blocks to be implemented. Analysing the circuit structure we can notice that only the third output of the two-bit multiplier is used. This fact proves that EA considers each output of a complex building block separately and in some case it does not employ all outputs. The circuit implementing the third output of the two-bit multiplier requires at least 3 primitive logic gates (see Appendix C). The optimal two-bit multiplier contains 7 logic gates. This means that the circuit in question contains 14 non-redundant primitive logic gates. Comparing the circuit in design B with the circuit in design A, we notice that this structure is far from optimal (14 primitive logic gates compared to 10). This example shows that using different set of building block, different circuit structures can be evolved.

### A two-bit multiplier

Evolution of two-bit multiplier at function-level is another problem that is interest us. A two-bit multiplier design problem has been actively studied recently in the area of extrinsic EHW. This circuit has been evolved by different researchers at gate-level.

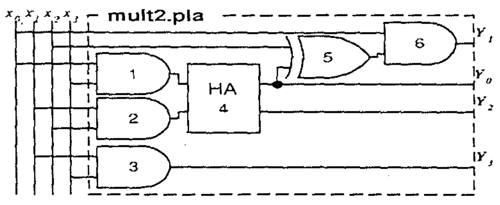


Figure 5.6: Evolved two-bit multiplier design (**A**); # logic gates = 7; # non-redundant primitive logic gates = 7; # building blocks = 6; HA is the half adder.

Our interest in this function lies in the possibility of evolving the two-bit multiplier at function level. The two-bit multiplier has been evolved using half and full adders. Interesting results occurred during the analysis of evolved circuit structures. First, no circuit structure containing one-bit full adders have been evolved. Approximately 50% of circuit structures contain no multi-input multi-output building blocks. In other words, basically these circuits are evolved at gate-level, even if some complex functions have been defined in a functional set of logic gates. Approximately 20% of two-bit multipliers evolved at function level have the structure shown in Fig. 5.6 (design A), 20% of two-bit multipliers encompass a circuit structure shown in Fig. 5.7 (design B). These two circuit designs are the most optimal two-bit multipliers evolved at function-level. Approximately 10% of fully functional circuits evolved consist of more than 10 primitive logic gates. There is no interest in these circuits, since they are not optimal in terms of the number of primitive non-redundant logic gates involved in the circuit structure.

Let us consider in details the circuit designs A and B. These circuits are similar to each other in terms of the type logic circuits employed. In both cases, 4 AND, 1 EXOR and 1 Half adder are used. The difference is in the connectivity of these elements. Note that the implementation of outputs $Y_2$ and $Y_0$ is identical for both designs in question. The output $Y_3$ is implemented separately in both case, and it

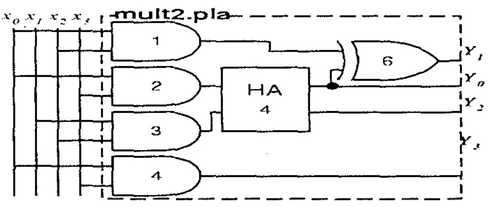


Figure 5.7: Evolved two-bit multiplier design (B); # logic gates = 7; # non-redundant primitive logic gates = 7; # building blocks = 6; HA is the half adder.

employed only one AND logic gate. But in design A, EA defines that output $Y_3$ depends on the variables $x_1$, $x_3$ and in the case of design B the same logic function depends on variables $x_0$, $x_3$. Another interesting aspect of these two structure is in the implementation of output $Y_1$. Again, in order to realize this circuit the same set of functional logic gates has been used: 3 AND, 1 EXOR, 1 half adder. In this case the order of AND and EXOR gates has been changed. This can be done by a human designer as well.

So, these two examples demonstrate that the similar circuit structures with variations of connections can be evolved. In this case EA considers each of the evolved solutions as a unique one and does not recognizes the similarities in evolved circuits. The knowledge of well-known logic algebras has to be used at some stage of the evolutionary process, if one desires to evolve different circuit structures in terms of logic algebra equality.

### 3-bit multiplier

One of the most interesting problem currently investigated by researchers in the area of EHW is evolving a three-bit multiplier. The functional set of logic functions used in circuit evolution contains the logic functions from which it is easily to construct a 3-bit multiplier circuit. Let us consider in details some of the circuit structures

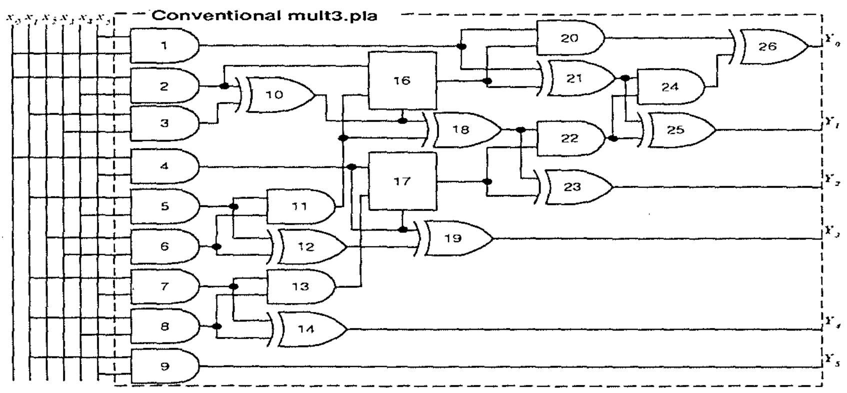


Figure 5.8: Most efficient conventional gate-level three-bit multiplier (# logic gates = 34; # non-redundant primitive logic gates = 34; # building blocks = 21).

evolved at function level.

The 3-bit multiplier can calculate the product of two integers $a$ and $b$ in the range 0-7. The conventional 3-bit multiplier depicted at Fig. 5.8 requires 34 primitive logic gates (if NOT is considered as a separate logic gate), 26 logic gates (if MUX is considered as a separate building block) [9]. Some of sub-circuits used in this design represent half bit adder. Five half-adder structures has been found. Considering the size of circuit in terms of building blocks, the design requires 21 building blocks, if half adder and multiplexer are considered as separate building blocks. The most efficient conventional 3-bit multiplier synthesised using an one-bit full adder and a half adder contains 16 building blocks.

The most efficient 3-bit multiplier evolved at gate-level, requires only 23 gates [135]. Note that this structure contains 3 AND gates with one inverted input. In our approach we count only the number of NOT, AND, EXOR and OR gates. Therefore in our calculations, AND with one inverted input requires 2 primitive logic gates: AND and NOT. From this point of view the circuit structure reported in [135] contains 26 primitive logic gates.

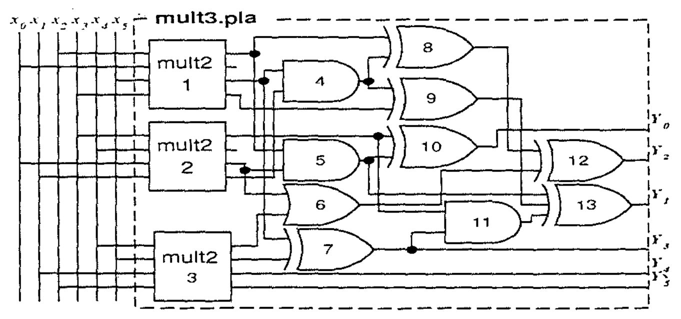


Figure 5.9: Evolved 3-bit multiplier design (**A**): Functional set: {2, 7, 8, 9, 18}; # logic gates = 32; # non-redundant primitive logic gates = 28; # building blocks = 13; *mult2* is the 2-bit multiplier.

One can argue that the primitive logic gate can not be considered as a measure of quality of the evolved circuit since this is optimal only from an algebraic point of view and define nothing from the empirical point of view, when the circuit has to be actually implemented in hardware. Thus, for example one can consider the AND logic gate with inverted input as a unique unit; another user will divide it into 2 sub-units, since there is no manufactured logic gate that can implement this functionality. This has been demonstrated by analysing the quality of a conventional three-bit multiplier. In this work the quality of evolved circuit is estimated in terms of the number of primitive logic gates used in the circuit, since different building blocks with different characteristics can be involved in the circuit design performed at function-level.

The most efficient evolved 3-bit multiplier at function level is depicted in Fig. 5.9 (design A). This circuit requires 32 active primitive logic gates. The cost of this circuit has been calculated without taking into account the outputs used in two-bit multiplier building blocks. Note that the second output of multipliers 1 and 2 is not used. The implementation of the two-bit multiplier without this output requires at least 5 primitive logic gates ([26]). This means that 2 logic gates are employed to implement the second output and are not used in the implementation of other circuit outputs. Therefore, the circuit shown in Fig. 5.9 requires 28 primitive logic gates (32-2*2=28), such as AND, OR and NOT. Therefore, we can conclude that circuit structures evolved at function- and gate-level are comparable. At the same time the circuit structure in question is the most compact circuit implementation of a 3-bit multiplier. The circuit structure is very compact in comparison with conventional design. This design can be mapped onto an rectangular array of 4 columns and 4 rows, which is impossible to carry out with conventional circuit design. Also, this circuit requires only 13 building blocks in comparison with at least 16 building blocks in conventional design. This shows that the circuit is optimal in terms of the number of building blocks required to implement it. The circuit has been mapped directly from the genotype to symbolic representation. This means that no reductions performed by human has been executed. The chromosome representation introduced in this work allows us to use multi-input one-output building blocks of fixed functionality. One of these blocks is used in design A. Building block labeled 13 contains 3 inputs. This cell contains two primitive logic cells but is considered as an unique unit in the chromosome genotype. Using such type of building blocks in the circuit design allows us to reduce the number of building blocks used.

It can be seen from the analysis of the circuit structure depicted in Fig. 5.9 that the quality of evolved circuit in terms of the number of logic gates $cost(\mathcal{N})$ (see Eq. 5.2.5) can be defined by 3 different metrics:

- $M_1$: the number of primitive logic gates;
- $M_2$: the number of non-redundant primitive logic gates;

- $M_3$: the number of building blocks.

This means that the actual cost of building block $cost(\mathcal{B}_j)$ (see Eq. 5.2.6) is calculated differently for each of these three cases. If the building block is uncommitted, different metrics to define the cost of the building block can be applied.

If the first metric $M_1$ is used as a measure of the quality of evolved circuit, $N_j^p$ defines the minimum number of primitive logic cells required to implement the logic function that describes the behaviour of the building block $\mathcal{B}_j$. For example, the 2-nd building block of the circuit shown in Fig. 5.9 describes the two-bit multiplier. In the interpretation mentioned above, the size of this building block is $N_2^p = 7$, since the optimal implementation of two-bit multiplier requires 7 primitive logic gates. This metric can be suitable when the designer wants to know how many primitive logic gates are required to evolve more complex system.

In the case of using the second metric $M_2$ as a measure of the quality of evolved circuits, $N_j^p$ is interpreted in the minimal number of primitive logic cells required to implement the outputs of the building blocks that are actually used in the circuit structure. For instance, the size of logic gate $\mathcal{B}_2$ in design A is 5, since the second output of this building block is not used, $N_2^p = 5$. This metric is very useful, when the goal of circuit design is to evolve the most efficient logic circuit in terms of the number of primitive active logic gates.

The third metric $M_3$ defines the number of building blocks actually used in the circuit regardless of their size. In this case the building block $\mathcal{B}_2$ in design A is considered as one unique unit and, therefore, $N_2^p = 1$. The designer would prefer to use this metric if target implementation technology is FPGA. In this case the designer is interested only in the number of logic cells occupied in the rectangular array.

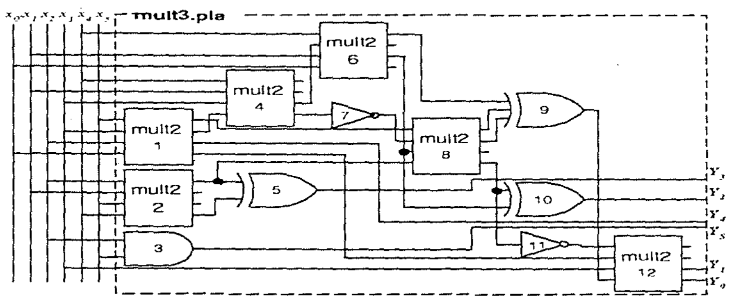


Figure 5.10: Evolved 3-bit multiplier design (**B**): Functional set: {2, 7, 8, 9, 18}; # logic gates = 51; # non-redundant primitive logic gates = 33; # building blocks = 12; *mult2* is the 2-bit multiplier.

As it can be seen from the statements given above, the quality of evolved circuit can be measured differently, if the circuit is evolved at function-level. Using different metrics as an optimisation criteria we can define the optimal circuit structures that can be optimal from one point of view and completely unacceptable from another. The analysis of the three-bit multiplier circuits given below illustrates this phenomena.

Next, we will consider the circuit shown in Fig. 5.10 (design B) that contains 51 active primitive logic cells. This is the optimal circuit in terms of the number of primitive logic cells that has been evolved using the functional set of 2, 7, 8, 9 and 18 logic cells. Also, this is the most efficient circuit in terms of the number of building blocks involved: the circuit requires 12 building blocks. Analysing the circuit structure we can easily notice the presence of large cell redundancy. For example, only two outputs are used in 4 two-bit multipliers. This caused the redundancy and therefore, there is a big difference between the estimation of the quality of evolved circuit using metric $M_1$ and $M_2$. Comparing this design with the circuit design A we can notice that this is optimal in terms of the number of building blocks used. This circuit contains 12 building blocks in comparison with 13 that are required to

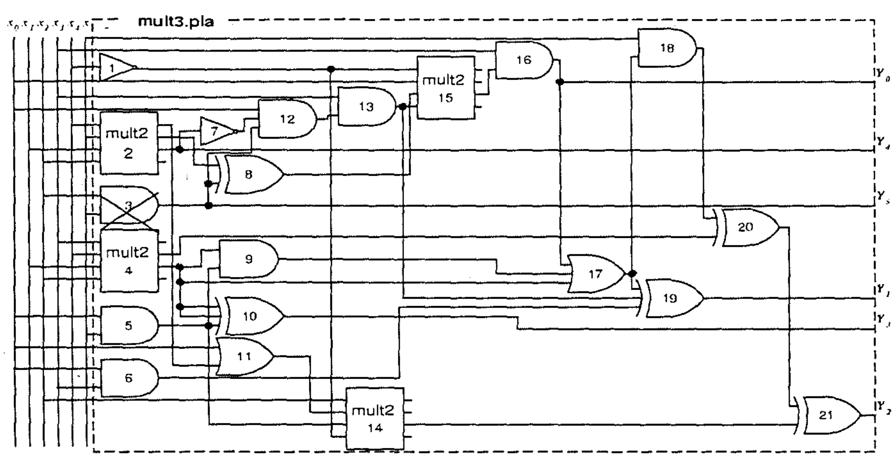


Figure 5.11: Evolved 3-bit multiplier design (C): Functional set: {2, 7, 8, 9, 18}; # logic gates = 50; # non-redundant primitive logic gates = 39; # building blocks = 20; *mult2* is the two-bit multiplier.

implement design A. At the same time, this circuit is far from optimal and even is not comparable with optimal conventional circuit if an estimation is made in terms of the number of primitive logic gates used. The most compact circuit layout required to implement this circuit on FPGA contains 3 rows and 6 columns. Obviously, this circuit will have higher circuit delay then the circuit depicted in Fig. 5.9 (design A), since it requires larger number of columns. It should be noted that less columns in the compact circuit implementation means less delay.

The circuit design C shown in Fig. 5.11 contains 50 active primitive logic cells. This design is far from optimal from all points of view: the number of primitive active logic gates, the number of building blocks and circuit delay. But there is one interesting eventuality that can be noticed during inspection of this circuit. Only the third output is used in two two-bit multipliers. This aspect proves one more time that EA does not consider the building block as a whole unit but examines each output of the building block as a separate sub-function. At the same time, the

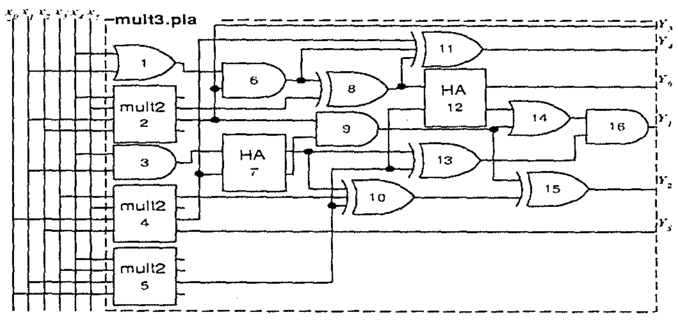


Figure 5.12: Evolved 3-bit multiplier (**D**): Functional set - {2, 7, 8, 9, 18, 22}; # logic gates = 40; # non-redundant primitive logic gates = 31; # building blocks = 16; HA is the half adder, *mult2* is the 2-bit multiplier.

analysis of this structure shows that there is one redundant logic cell labelled 3. A similar cell is implemented inside the structure of building block marked 2 (the 4th output). This can reduce the size of the evolved design. This redundancy illustrates that the evolution of this particular circuit requires more computation time, since some obvious redundancy still can be found.

The three-bit multipliers discussed above have been evolved using two-bit multiplier as building blocks. It has been shown that some of the evolved circuits are optimal in terms of the number of primitive logic gates used or in terms of the number of building blocks utilised. In our further research we would like to answer the question whatever it is possible to evolve more optimal designs using another complex functions as a building blocks and how their efficiency will be compared with ones previously reported. The next two circuit designs depicted in Fig. 5.12 and Fig. 5.13 and are evolved using the two-bit multiplier and the half-adder as building blocks.

Surprisingly, no efficient designs have been evolved when the two-bit multiplier and the half adder were included in the functional set of logic gates. The most efficient design evolved contains 31 non-redundant primitive logic gates, 16 building

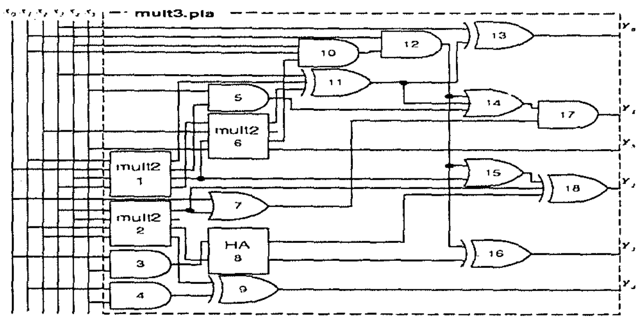


Figure 5.13: Evolved 3-bit multiplier (**E**): Functional set - {2, 7, 8, 9, 18, 22}; # logic gates = 42; # non-redundant primitive logic gates = 37; # building blocks = 18; HA is the half adder, *mult2* is the two-bit multiplier.

blocks and is shown in Fig. 5.12 (design D). It is clear, that these characteristics of evolved three-bit multiplier are comparable with ones defined for the conventional three-bit multiplier, but are far from optimal if we compare these with the most efficient evolved designs. In this design we can notice the presence of redundancy and very low percentage of outputs of building blocks used. For example, only the third output of the two-bit multiplier labeled 5 is used. The circuit illustrated in Fig. 5.13 (design E) is distinctive since this illustrates how EA can use multi-input one-output building blocks. This design contains 4 3-input 1-output logic cells. This shows that evolution defines some sub-functions that can be arranged in the special building blocks even if the functionality of complex building blocks is not defined. Note that if the multi-input one-output building block is used in evolution, the number of inputs in this building block is considered as a separate gene that can be mutated. This means that the functionality of the building block can be changed by varying the number of inputs in it. This shows that the structure of the building block can be defined during evolution and in this case the method starts to behave analogously to the one proposed by J. Koza et al. [51] and named Automatically Defined Functions.

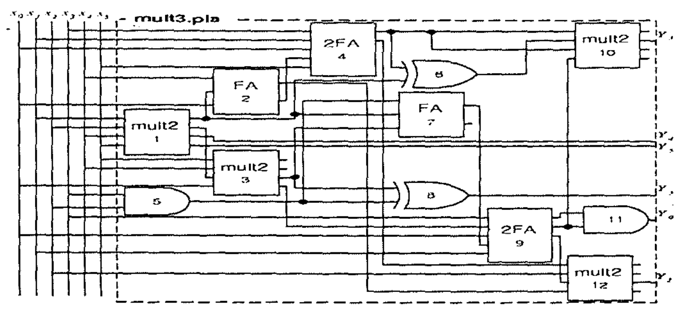


Figure 5.14: Evolved 3-bit multiplier (design **F**): Functional set - {2, 7, 8, 9, 17, 18, 19, 22}; # logic gates = 62; # non-redundant primitive logic gates = 45; # building blocks = 12; FA is the one-bit full adder, 2FA is the two-bit adder, *mult2* is the two-bit multiplier.

This is one of the issue that I will pay attention to my future research.

More interesting circuit design is depicted in Fig. 5.14 (design F). This is optimal design in terms of the number of building blocks used: it contains 12 building blocks. In this case EA discovers at least two optimal designs in terms of the number of building blocks used: designs F and E. Both of these designs consist of 12 building blocks, but the types of building blocks are different. This design contains high level of diversity in terms of functional type of building blocks involved: two-bit multiplier, one- and two-bit adders, AND and EXOR logic gates. Surprisingly we discover that in this design only 4 primitive logic gates labeled 5, 6, 8 and 11 are used separately. This circuit contains the lowest percentage of primitive logic gates used in the circuit structure. This is the best example that illustrates the employment of high-level functionality block efficiency.

## 5.5 Summary

This chapter introduces the function-level extrinsic EHW approach for the first time and demonstrates its applicability to the digital design of arithmetic logic circuits.

The correctness of the genotype after the recombination operation was supported by a connection repair algorithm. In the circuit structure both multi-input one-output and multi-input multi-output logic functions can be used as building blocks. Both the analysis of algorithm performance and the specific features of evolved structures are discussed in detail. The advantage of the proposed chromosome representation is that it allows the use of multi-input multi-output logic functions as a building block. Analysis of circuit structures evolved shows that EA considers each output of complex building block as a separate sub-function that can be potentially involved in the actual circuit structure. Each complex building block encodes some regularity without explicit knowledge of what makes a good sub-function. Another advantage of the proposed method is that it reduces the chromosome length significantly by using the more complex building blocks. For example, the three-bit multiplier can be composed using only 12 building blocks in comparison with 26 primitive logic gates. In this particular case the size of circuit layout required to evolve fully functional three-bit multiplier can be significantly reduced.

A lot of work has to be done in the area of defining the structure of the most reusable building blocks during automatic evolution. This can be done by introducing the internal connectivity genes at the building block level of chromosome representation. The research has to be focused on the identification of useful modules without explicit knowledge of what makes a good module or how to extract task-specific information being placed in the problem solver.

# Chapter 6

# Bidirectional incremental evolution

Although the algorithm performance has been drastically improved with the introduction of the function-level extrinsic EHW approach (see Chapter 5), the problem of evolving the logic functions of large number of inputs and outputs remains. This is because the fitness function in the methods discussed above is calculated using a *complete* truth table. Clearly such a procedure can lead to very slow evolution if the size of the truth table is large. Also, it has been shown previously that during solving complex problems, a direct evolution makes slight progress during the first several thousand of generations and stalls after that. In order to overcome these two problems we propose a *bidirectional incremental evolution* (BIE) that allows us to diminish gradually the complexity of evolved systems. This approach performs incremental evolution in two directions: from complex system to sub-systems with sufficient complexity and from sub-systems to complex system. In this approach, a complex problem is gradually decomposed into some sub-tasks during evolution, and, then, the problem is evolved incrementally, by starting with simpler behaviour and gradually making the task more challenging and general. The system discovers the evaluation tasks and their sequence automatically. Each sub-task is evolved using a self-adaptive function-level extrinsic EHW. The self adaptation is performed in order to evolve circuit layout automatically (see Chapter 4). The function-level EHW is applied to improve the algorithm performance (see Chapter 5). The evaluation of each sub-task is performed using a dynamic fitness function introduced in Chapter 3. The method is tested using standard benchmarks and compared with direct evolution. The bidirectional incremental approach evolves more effective and more general circuits and should also scale up to harder tasks.

## 6.1 Introduction

Evolvable Hardware (EHW) has been introduced as a target architecture for complex system design based on evolution. However, the evolution of complex system turns out to be very difficult task. There are several reasons for this.

One is the problem of evolving systems based on a long chromosome string. Previous approaches of this problem include the use of variable length chromosome [114], function-level evolution [17], [4], Chapter 5, automatically defined functions [51] and the divide-and-conquer approach (so called increased complexity evolution) [54], [55]. The complexity of evolved system depends on the number of inputs and outputs in the implemented logic function. The higher the number of outputs, the higher the system complexity, if the number of inputs remains the same. Similar conclusions can be made regarding to the number of inputs. The larger the number of inputs in the implemented function, the higher the complexity of the evolved system, if the number of outputs remains the same. All methods except the divide-and-conquer approach become unapplicable once the complexity of the system in terms of either the number of outputs or the number of inputs is increased. The divide-and-conquer method deals very well with complex systems of large number of outputs and small number of inputs. This is because the basic idea of this approach is to divide the system by outputs and evolve each sub-system separately. Once each sub-system is evolved, the complex system is assembled. The disadvantage of this approach is that it is unable to deal with systems of large number of inputs, since the number of input combinations in the truth table increases exponentially with increasing the number of inputs in the logic function.

Another reason is that the evolutionary process stalls after several thousand of generations. In this case the number of generations required to perform a task increases drastically with increasing the complexity of task. Thus, a two-bit multiplier (4 inputs, 4 outputs) can be easily evolved after 5 000 generations [9]. At the same time, the evolution of fully functional three-bit multiplier (6 inputs, 6 outputs) at gate-level requires from 3 000 000 generations [9] up to 10 000 000 generations [116]. According to this data we can predict that billions of generations are required to evolve a fully functional 4-bit multiplier (8 inputs, 8 outputs). This example shows the usefulness of an extrinsic EHW approach applied to evolve the higher complexity functions, since the number of generations required to perform increases drastically. A similar problem occurs in the evolutionary robotic area. In order to overcome this problem, several researchers have demonstrated that incremental evolution can be successfully applied to stochastic dynamic problems that are implemented using neural networks [117], [59], [118]. In incremental evolution, the neural network learns incrementally the complex general behaviour by starting with simpler behaviour and gradually making the task more general.

Finally, a more complex problem requires more computational effort to be solved.

The computational effort mostly depends on the three following parameters: (1) the dimension of the circuit layout; (2) the number of generations required to solve the given problem, and (3) the size of the truth table. The dimension of the circuit layout can be referred to the long chromosome string. The second parameter can be reduced once the problem of stalling the evolution is solved, and, the size of final parameter can be abated by solving the sub-tasks given by truth tables of smaller size. This problem can be resolved by applying the schemes to speed up the EA computation time that involves the fitness computation in parallel or a partitioned population evolved in parallel [136], [115]. Also, it can be solved by using the incremental evolution.

Therefore, the direct application of an extrinsic EHW approach to design *practical* digital circuits involves three major problems:

1. limitation in the chromosome length;
2. "stalling" effect in the evolutionary process;
3. restriction of the computation time.

All of the problems mentioned above can be solved using *bidirectional incremental evolution*. The proposed approach significantly differs from the incremental evolution approaches proposed in the past. It incorporates two main approaches: Divide-and-Conquer and Incremental evolution. The principle of our approach is to *divide* a complex task into simpler sub-tasks, to *evolve* each of these sub-tasks and then *merge* incrementally the evolved sub-systems, reassembling a new evolved complex system. The evaluation tasks and their circuit layout dimensions can be identified using either standard decomposition methods or an EHW-oriented decomposition. In our method first the evolution defines the partitioning vectors of any given complex system. These vectors are used to decompose the system into a number of sub-systems. Each of sub-systems is then evolved separately. If a sub-systems is not sufficiently partitioned, new partitioning vectors are defined and the subsystem is decomposed further. This may continue until a final evolvable sub-system set is defined. Once all sub-systems have been evolved, they are merged incrementally into one cost-optimised system. So, the bidirectional incremental evolution contains two processes: (1) evolution towards an modularised system; and (2) evolution towards an optimised system.

**Stage 1: Evolution towards a modularised system.** The decomposition of complex systems into sub-systems can be performed using either the standard decomposition methods, or the EHW-oriented decomposition. Any *standard functional decomposition method* can be used to divide a system into several sub-systems. In this case the incremental evolution is performed in one direction: from sub-systems to more complex system. The sequence of evolving the sub-tasks is defined by standard functional decomposition methods. One disadvantage of this approach can be the fact that the standard functional decomposition methods are designed in most cases for some particular architectural structures, such as PLA, FPGA, etc. and in some cases are not applicable to EHW approaches. Another problem is that the standard decomposition methods are developed using a specific logic algebra. In some cases EHW exploits the larger search space rather than one, limited by logic algebra representation. The standard decomposition methods discussed in the past take into account no specific features of EHW approach. In order to overcome these problems the *EHW-oriented decomposition* is proposed in this Chapter. The evolution from a complex system into another one is carried out by means of the EHW-oriented

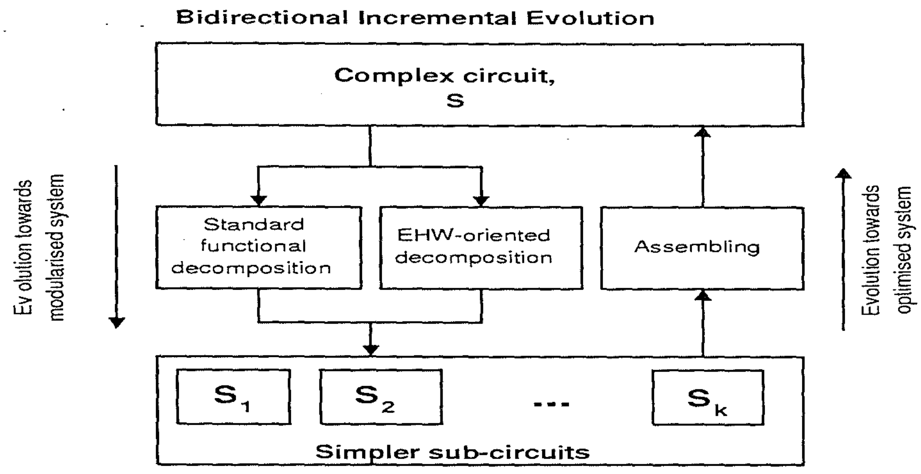

Bidirectional Incremental Evolution
Complex circuit, S
Standard functional decomposition
EHW-oriented decomposition
Assembling
Ev olution towards modularised system
Evolution towards optimised system
$S_1$
$S_2$
...
$S_k$
Simpler sub-circuits

Figure 6.1: The structure of bidirectional incremental evolution (BIE) Incremental evolution performs in two directions: from complex system to sub-systems (complex system decomposition) and from sub-system back to the complex system (complex system optimisation).)

decomposition. The sub-tasks and their sequence of evolution are defined automatically. In this case the system is evolved using bidirectional incremental evolution. The evolution is undertaken on a complex system, in order to define the partitioning vectors. And then, the system is decomposed into a number of sub-systems. Each sub-system is evolved separately. Once all sub-systems have been evolved, they can easily be assembled to the fully functional system. After the first stage of bidirectional incremental evolution, the fully functional complex system can be assembled.

**Stage 2: Evolution towards an optimised system.** During this stage of bidirectional incremental evolution, the system is assembled from simpler sub-systems into a more complex one and is evolved in order to reduce the size of the system. The complexity of assembled sub-systems increases gradually. The bidirectional incremental evolution stops once the given complex system has been assembled and evolved to optimise its size. Thus, this method not only evolves the fully functional system gradually, but also decreases this size incrementally.

The basic principles and main approaches to bidirectional incremental evolution are summarised in Fig. 6.1.

All possible approaches to perform bidirectional incremental evolution will be discussed in this chapter.

## 6.2 Basic idea of bidirectional incremental evolution applied to digital logic design

The basic idea of incremental evolution is that the system *learns* the new behaviour starting with an easy task and the task complexity is gradually increased. Most of incremental evolution approaches reported in the past are used to evolve neural network architecture. In this case the neural network learns first simple task and with evolution the complexity of task incrementally increases.

Incremental evolution occurs under the following circumstances [117]:

1. Incremental evolution, when the problem itself changes over time; for instance a sequence of related problems of increasing complexity may be tackled by a continuing evolutionary algorithm.

2. Cases where the genotype length may change in the course of evolution, particularly where increasingly complex solutions require increasing genotypes in order to specify them.

3. When an initially random population becomes genetically converged after very few generations.

Incremental evolution consists in gradually evolving a complex task on a series of tasks of increasing complexity [57], [58].

Suppose that our goal is to generate a fully functional complex cost-optimised system. The cost of system is defined by the number of active logic gates. In this case the circuit synthesis has to contain two main evolutionary processes that (1) find a fully functional solution, $\gamma(F_1)$, and (2) optimise a number of active logic gates in the circuit, $\gamma(F_2)$. Therefore, the evaluation process can be performed using the dynamic fitness function introduced in Chapter 3. It is extremely difficult to evolve a logic function of large number of inputs and outputs using direct evolution. Therefore, the incremental evolution is applied to solve this problem. In this case incremental evolution operates in two directions specified by evolutionary processes mentioned above. In order to avoid the "stalling" effect appearing in solving complex dynamic problems, incremental evolution is used instead of direct evolution. Therefore, the bidirectional incremental evolution takes place to evolve complex circuits.

The easiest and quickest way to obtain fully functional solution is to decompose a complex logic function into several sub-functions and evolve each sub-function separately. The fully functional complex circuit is assembled from the evolved fully functional sub-circuits. This is the basic idea of Divide-and-Conquer method. Hence, the evolution of fully functional solution, $\gamma(F_1)$ can be performed using "divide-and-conquer" approach.

The minimisation of evolved fully functional solution can be performed using incremental evolution. Each sub-function is first evolved in order to minimise the number of active logic gates in it. Then, several sub-functions are assembled and the evolution is undertaken to evolve more complex task. The complexity of evolved circuit is increased gradually. The process is repeated until the most complex system has been assembled and has been evolved. This method can be defined as "conquer-and-evolve" approach.

Therefore, the circuit evolution is performed in two directions:

1. from complex system to sub-systems;

2. from simple sub-systems to complex system.

In both directions, incremental evolution can be applied in order to evolve a complex system. Therefore, in the proposed method, namely bidirectional incremental evolution (BIE), different fitness criteria are applied in each direction. At the same time during evolution of one increment (one sub-task) the dynamic fitness function can be applied as well. Also, during the first stage of BIE the complexity of the evolved problem is incrementally decreased and during the second stage the complexity of evolved problem is incrementally increased.

There are two basic approaches that can be applied to implement bidirectional incremental evolution.

First approach uses a priori knowledge of the complex system and divides the complex system into sub-systems using standard functional decomposition methods. For example, in combinational circuit design the system can be decomposed according to Shannon's theorem, Ashenhurst and Curtis decompositions, etc. In other applied problems, any a priori-based decomposition methods can be applied in order to divide the system. Since in the standard functional decomposition methods the decomposed sub-functions have the similar number of inputs and outputs, the extrinsic EHW approach with homogeneous circuit layout can be applied (see Chapter 4). In order to speed up the evolutionary process, evolution can be performed at function level (see Chapter 5).

Another approach defines the sub-functions automatically during evolution. The sub-functions extracted from the complex funciton are easily evolvable. The EHW parameters required to evolve a specified sub-function are defined during evolution as well, since the complexities of extracted functions are different. In this case the sub-functions are evolved using the self-adaptive extrinsic EHW discussed in Chapter 4. The evolutionary process can be speeded up if a function-level EHW approach is applied at this stage (see Chapter 5). Thus, self-adaptive function-level extrinsic EHW approach is used to evolve each sub-function. Hence, there are two automatic processes that can be executed during this stage: (1) automatically defined easily evolved tasks, and (2) automatically defined EHW parameters (Circuit layout evolution has been introduced in Chapter 4).

In the following sections the specific features of these two approaches applied to bidirectional incremental evolution are discussed in detail.

## 6.3 BIE with standard functional decomposition

### 6.3.1 Statement of problem

The essential idea of bidirectional incremental evolution is to *scale* the evaluation function (i.e., the "fitness function against which, say, a complex circuit is evolved) over time, with the aim of minimizing the overall time spent evolving a circuit that achieves the prescribed task. In the BIE with standard decomposition method, the

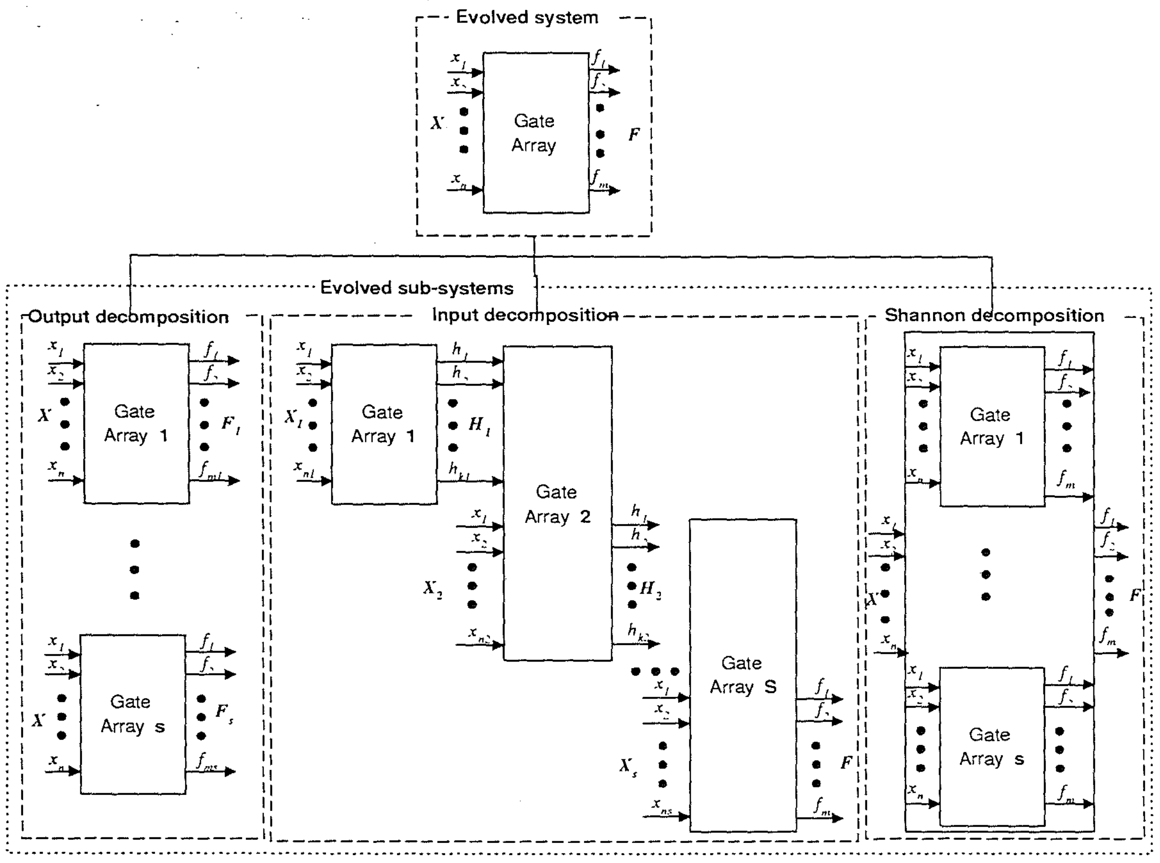


Figure 6.2: The bidirectional incremental method applied for designing digital systems of $n$ inputs and $m$ outputs.

evolution towards a modularised system is performed in two stages. First, the standard functional decomposition is undertaken to define the truth tables for each sub-circuit. Then, each sub-circuit is evolved. So, this stage of BIE is executed as in the Divide-and-Conquer method. Once all sub-circuits have been evolved, the evolution towards to an optimised circuit is undertaken.

Any of the methods from the decomposition theory can be applied in order to divide a complex system into sub-systems. Interactive decomposition, cascaded decomposition, joint decomposition, etc. can be employed to decompose the complex system. Then, each sub-circuit is evolved. Therefore, for the BIE with standard decomposition the internal connectivity of sub-circuits is known in advance (see Fig. 6.2). For example, the sub-circuits are connected iteractively, if the iteractive disjoint decomposition is applied to complex circuit (Fig. 6.2). The connectivity between the sub-circuits can be changed when the evolution towards to optimised system is applied.

Suppose the problem of interest is stated as follows: define the configuration of circuit $\mathbb{C}$ implementing the truth table $T$. In this case the complexity of evolution task is defined by the complexity of truth table. Each chromosome representing a circuit is evaluated according to the truth table $T$.

Once the standard decomposition is undertaken to divide complex truth table $T$, a sequence of smaller truth tables are generated: $\mathcal{T} = (T_0, T_1, ..., T_{k-1} = T)$. Let $G_i$ be the number of generations required to evolve a fully functional sub-circuit. Since in standard decomposition methods the sub-circuits are generated with similar complexity, the number of generations required to evolve all sub-circuits generated after the standard decomposition method is defined as $G^1 = kG_i$. Each sub-task $T_i$ is evolved separately. The initial population for each of the sub-tasks is generated randomly. The evolution towards an optimised system is performed as follows. The population of circuits is sequentially evolved by assembling $h$ sub-circuits at once and using evaluation function $T_{k-1+\Sigma\frac{]k[}{h}}$ for time $G_{k-1+\Sigma\frac{]k[}{h}}$, beginning with $T_k$ for time $G_k$ in order to optimise the size of evolved fully functional circuit. The evolution of each sub-task is stopped once the "stalling" effect has appeared. Each sub-function is evolved using self-adaptive function-level extrinsic EHW approach, since the complexity of the circuit is increased incrementally, the parameters for EHW approach are unique for each sub-function.

## 6.3.2 Standard functional decomposition methods

The evolution towards a modularised system is undertaken using standard functional decomposition methods. Any of the well-known decomposition methods can be applied to divide complex system into sub-systems. Some of the most popular standard functional decomposition methods applied to combinational logic design are listed below.

*Decomposition* means breaking a large logic block into several relatively smaller ones.

The *functional decomposition* (also called input decomposition) is a technique to break a logic function with many variables into several functions with fewer variables. The functions with fewer variables can be designed independently, and are relatively easier to design. Functional decomposition of a logic function is used for any algebraic representation of given function. It allows us to choose any functionally complete algebra for analysis. Therefore it is applicable for our approach.

**Definition 6.3.1.** An logic function $f(X)$ is said to have a *simple decomposition* with respect to set $X_1$ if there exist logic functions $h$ and $g$ such that

$$f(X) = g(h(X_1), X_2), \tag{6.3.1}$$

where $(X_1, X_2)$ is a partition of $X$, $X_1$ and $X_2$ are the bound and free set respectively.

**Definition 6.3.2.** An logic function $f(X)$ is said to have a *simple generalized decomposition* with respect to set $X_1$ if there exist logic functions $h_1, h_2, \cdots, h_k$ and $g$ such that

$$f(X) = g(h_1(X_1), h_2(X_1), \cdots, h_k(X_1), X_2), \tag{6.3.2}$$

where $(X_1, X_2)$ is a partition of $X$.

The number of elements in the bound and free set will be denoted by $n_1$ and $n_2$ respectively. The decomposition is said to be *trivial* if $n_1$ is 1 or $n$. A function that has a nontrivial generalized decomposition is said to be *decomposable*.

**Definition 6.3.3.** An logic function $f(X)$ is said to have a *multiple decomposition* if there exist logic functions $h$, $t$ and $g$ such that

$$f(X) = g(h(X_1), t(X_2), X_3), \tag{6.3.3}$$

and *interactive decomposition* if

$$f(X) = g(h(t(X_1), X_2), X_3), \tag{6.3.4}$$

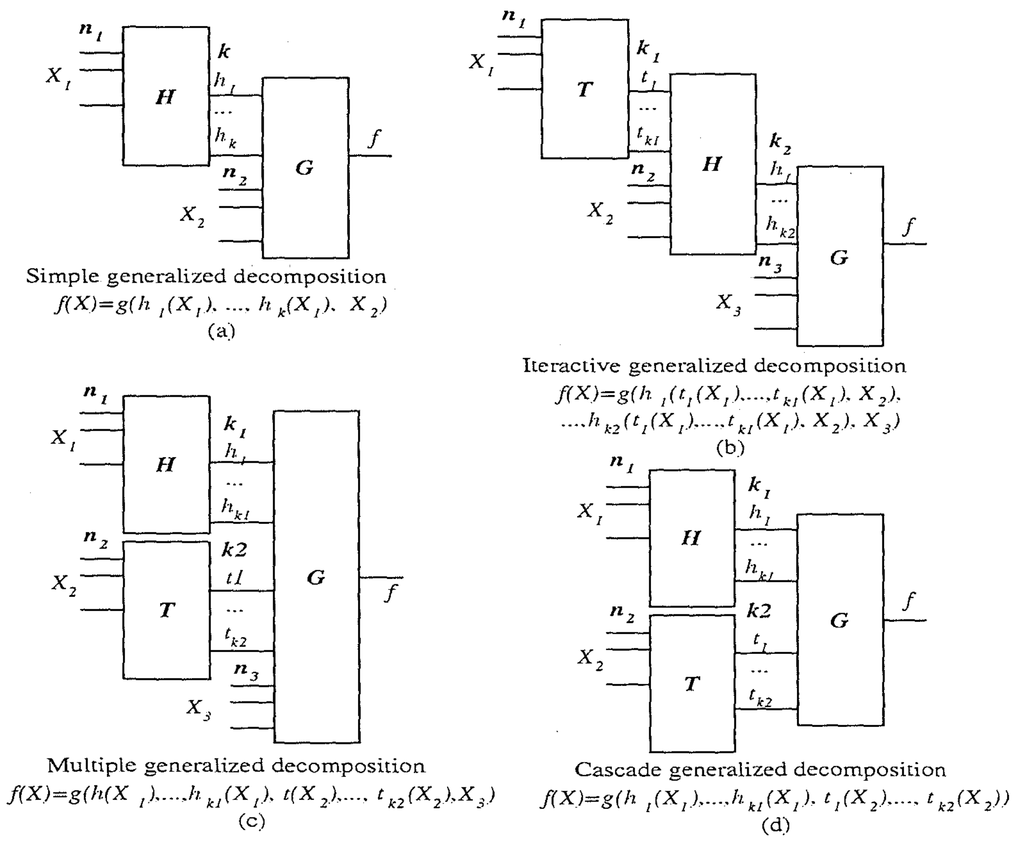
where $(X_1, X_2, X_3)$ is a partition of $X$.

*Disjoint decompositions* are those that decompose function $f(X)$ to two sub-functions $g$ and $h$ that have disjoint sets of input variables. Using non-joint sets of input variables such that $X_1 \cup C$ and $X_2 \cup C)$ leads to the *non-disjoint decomposition*. Every variable in $C$ is called a repeated variable (called also a shared variable). In the case of non-disjoint decomposition, the repeated variable map (RVM) that is a Karnaugh map in which variables from a non-empty set $C$ are repeated are applied. The functions with repeated variables can be represented in any known data structure that allows for don't cares.

According to the Shannon's expansion theorem the logic function can be decomposed as follows. Let $f(x_1, x_2, ..., x_n)$ be any Boolean expression of $n$ variables. Then Shannon's theorem in the SOP (sum-of-products) form says that

$$f(x_1, x_2, ..., x_n) = f(1, x_2, ..., x_n) \cdot x_1 + f(0, x_2, ..., x_n) \cdot \overline{x_1}. \tag{6.3.5}$$

The circuit diagrams for different functional decompositions of logic functions and their systems are shown in Fig. 6.3. The details of these methods can be found in any book on logic design.



n1
X1
H
k
h1
...
hk
n2
X2
G
f
Simple generalized decomposition
f(X)=g(h1(X1), ..., hk(X1), X2)
(a)
n1
X1
T
k1
t1
...
tk1
n2
X2
H
k2
h1
...
hk2
n3
X3
G
f
Iteractive generalized decomposition
f(X)=g(h1(t1(X1),...,tk1(X1), X2), ...,hk2(t1(X1),...,tk1(X1), X2), X3)
(b)
n1
X1
H
k1
h1
...
hk1
n2
X2
T
k2
t1
...
tk2
n3
X3
G
f
Multiple generalized decomposition
f(X)=g(h(X1),...,hk1(X1), t(X2),..., tk2(X2),X3)
(c)
n1
X1
H
k1
h1
...
hk1
n2
X2
T
k2
t1
...
tk2
G
f
Cascade generalized decomposition
f(X)=g(h1(X1),...,hk1(X1), t1(X2),..., tk2(X2))
(d)

Figure 6.3: Circuit diagrams of functional decomposition. (a) Simple generalized decomposition; (b) Iteractive generalized decomposition; (c) Multiple generalized decomposition; (d) Cascade generalized decomposition.

The *decomposition by outputs* is a technique to break a logic function with many outputs into several functions with fewer outputs. The functions with fewer outputs can be designed independently.

Table 6.1: Initial data: Evolving the sub-functions of sqn_d.pla synthesized using Shannon's decomposition and function-level extrinsic EHW.

| | |
|---|---|
| **Circuit layout, $N_{cols} \times N_{rows}$** | 1x35 |
| **Connectivity parameter, $N_{connect}$** | 35 |
| **Functional set** | $\{2, 7, 8, 9\}$ |
| **Gate distribution** | Proportional |
| **Type of layout** | Homogeneous |
| **Type of logic cell** | Multi-Input One-Output |
| **Maximum number of inputs in cell** | 4 |
| **Population size** | 5 |
| **Number of generations** | 10000 |
| **Number of ES runs** | 100 |
| **Circuit mutation rate** | 0.03 |
| **Fitness type** | $\mathcal{F}_1 + \mathcal{F}_2$ |

### 6.3.3 Experiment A: BIE with Shannon's decomposition

In order to investigate how BIE behaves when different types of functional decomposition are chosen, a number of experiments have been performed. Two main types of standard decomposition have been chosen to decompose any logic function: (1) decomposition by outputs; and (2) Shannon's decomposition. In this chapter we will consider in details how BIE with Shannon's decomposition evolves combinational logic functions.

The evolved function sqn_d.pla is taken from benchmark library. 7 inputs and 3 outputs function is given with 84 input combinations. No fully functional solutions have been evolved using direct function-level EHW during 150 000 generations with population of 5 individuals. The initial data is given in Table 6.1. The average circuit functionality obtained is 94.0476 (see Table 6.4). This means that the number of generations has to be increased in order to evolve any fully functional solution.

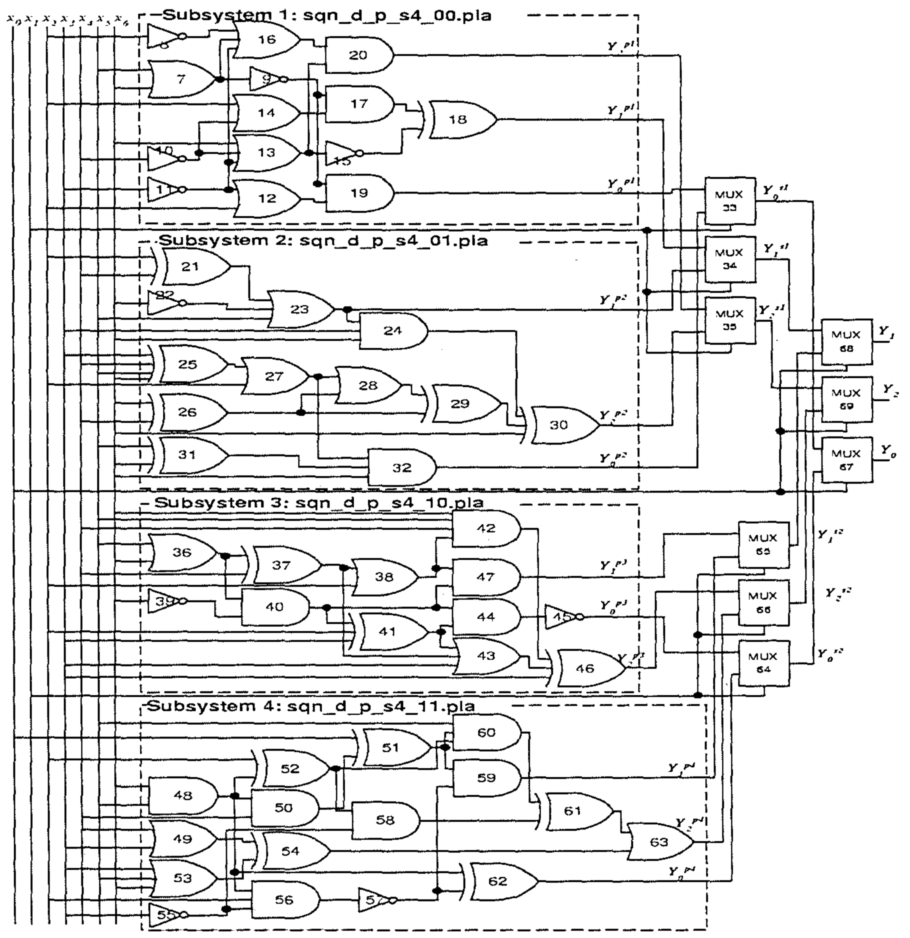
The task of interest to us was to evolve this function during 150 000 generations. In this experiment the function is decomposed using Shannon's decomposition. Note that the Shannon's decomposition has been performed without optimisation by any criteria. We are foreseeing that, by applying the optimisation at this stage, it is possible to achieve much better results than reported in this Chapter.

**Stage 1: Evolution towards a modularised system** The function has been decomposed by two input variables: $x_0$ and $x_1$. Applying Shannon's decomposition, 4 new sub-functions are generated. Thus, the complex truth table is decomposed into four smaller truth tables with 5 inputs and 3 outputs each. Each of sub-systems has been evolved 100 times. The initial data for this experiment are given in Table 6.1. The experimental results are summarised in Table 6.2 and Fig. 6.4. Each of sub-circuits has been evolved separately. Since the complexity of evolved sub-functions remains the same, the evolution has been carried out during 10000 generations for each sub-function.

As a result of evolution, fully functional solutions have been evolved. $F_2(N_f^{opt})$ defines the number of active logic gates in the most optimal design evolved for given sub-circuit. The total number of active logic gates in the evolved four sub-circuits is 70. Since the Shannon's decomposition is applied, the evolved sub-circuits have to be connected to each other using 9 multiplexers. Three logic gates are required to implement each multiplexer. Therefore, the total number of logic gates in the evolved fully functional circuit is 97. The circuit structure synthesised from the logic sub-functions evolved using function-level EHW is illustrated in Fig. 6.4. In order

Table 6.2: sqn_d.pla: Specification of sub-circuits evolved by BIE with standard decomposition. OD and SD are the output and the Shannon's decompositions (by inputs) respectively; FEHW is the function-level EHW; $n$, $m$ and $p$ are the number of inputs, outputs and input combinations in the subsystem evolved; $k$ is the number of additional logic gates involved to connect sub-systems; $F_2(N_f^{opt})$ is the number of active logic gates in the most efficient evolved circuit.

| **Method** | **Sub-system** | | | | **ES performance** | | | |
|---|---|---|---|---|---|---|---|---|
| | **Sub-circuit** | $n$ | $m$ | $p$ | $\overline{F_1^{bf}}$ | $\overline{F_2(\mathcal{N}_f)}$ | $\mathcal{R}(\mathcal{N}_f)$ | $F_2(N_f^{opt})$ |
| | | | | **sqn_d.pla** | | | | |
| FEHW | sqn_d | 7 | 3 | 84 | 94.0476 | - | 0 | 0 |
| SD & FEHW | sqn_d_s4_00 | 5 | 3 | 15 | 99.6375 | 17.5 | 75 | 16 |
| | sqn_d_01 | 5 | 3 | 23 | 97.84375 | 24.6 | 25 | 19 |
| | sqn_d_p_10 | 5 | 3 | 23 | 97.9999 | 24.75 | 31 | 17 |
| | sqn_d_p_11 | 5 | 3 | 23 | 96.98312 | 26.2 | 12 | 18 |
| | Assembled system: $F_2 = F_2^{s0} + F_2^{s1} + F_2^{s2} + F_2^{s3} + k$ | | | | | | | **97** |
| | Total $N_{gen} = 40.000$ | | | | | | | |
| | Optimised system: $F_2^{opt}$ | | | | | | | **94** |
| | Total $N_{gen} = 50.000$ | | | | | | | |
| OD & FEHW | sqn_d_o1.pla | 7 | 1 | 84 | 94.6190 | 15.6 | 20 | 10 |
| | sqn_d_o2.pla | 7 | 1 | 84 | 98.9796 | 19.75 | 66 | 16 |
| | sqn_d_o3.pla | 7 | 1 | 84 | 99.1667 | 27.1429 | 7 | 23 |
| | Assembled system: $F_2 = F_2^{o0} + F_2^{o1} + F_3^{o2}$ | | | | | | | **49** |
| | Total $N_{gen} = 110.000$ | | | | | | | |
| | Optimised system. $F_2^{opt}$ | | | | | | | **32** |
| | Total $N_{gen} = 160.000$ | | | | | | | |
| | | | | **m1_d.pla** | | | | |
| FEHW | m1_d.pla | 6 | 12 | 32 | | | | |
| OD & FEHW | m1_d_o1.pla | 6 | 6 | 32 | 99.4792 | 19.4667 | 85 | 19 |
| | m1_d_o2.pla | 6 | 3 | 32 | 99.8625 | 13.8 | 95 | 13 |
| | m1_d_o3.pla | 6 | 3 | 32 | 98.7254 | 34.6667 | 7 | 29 |
| | Assembled system: $F_2 = F_2^{o0} + F_2^{o1} + F_2^{o2}$ | | | | | | | **61** |



Subsystem 1: sqn_d_p_s4_00.pla
Subsystem 2: sqn_d_p_s4_01.pla
Subsystem 3: sqn_d_p_s4_10.pla
Subsystem 4: sqn_d_p_s4_11.pla
MUX 33
MUX 34
MUX 35
MUX 68
MUX 69
MUX 67
MUX 65
MUX 66
MUX 64

Figure 6.4: sqn_d.pla: Designed circuit using Shannon's decomposition and function-level EHW (Stage 1. Evolution towards a modularised system). The number of primitive logic gates: 97.

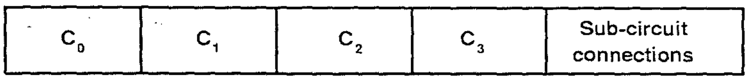


Figure 6.5: Sub-circuits allocation in the complex circuit genotype (*Sub-circuit connections* include the logic gates that link sub-circuits with each other.

to obtain the fully functional solution of desired circuit, the sub-circuits have to be assembled according to restrictions defined by the applied standard decomposition. For example, in the given case, the fully functional complex circuit is assembled using 27 additional primitive logic gates. This shows that it is very important to choose the right type of decomposition, since in some cases additional logic gates may be required to assemble the generated sub-functions together.

**Stage 2: Evolution towards an optimised system** During this stage of evolution the evolved sub-systems are assembled in a more complex form and evolution is performed towards optimised system. The new chromosome genotype for more complex system is defined by assembling the chromosome genotypes of sub-circuits. Merging the truth tables of sub-tasks generates the truth table for more complex task.

The new chromosome genotype is generated as follows. Let $\mathbb{C}_0$, $\mathbb{C}_1$, $\mathbb{C}_2$ and $\mathbb{C}_3$ be chromosome genotypes of the evolved 4 sub-circuits. Let $F_2(\mathbb{C}_i)$ be the number of active building blocks in the $i$-th circuit $\mathbb{C}_i$. Each sub-circuit contains 3 outputs and the decomposition is carried out by 2 input variables, hence $3^2$ multiplexers are required to implement a fully functional circuit. A multiplexer can be implemented in one building block. Hence, the number of active logic gates in the complex circuit is defined as follows: $F_2(\mathbb{C}') = F_2(\mathbb{C}_0) + F_2(\mathbb{C}_1) + F_2(\mathbb{C}_2) + F_2(\mathbb{C}_3) + 9$. The algorithm performs very well if there is a specific percentage of redundant gates in the chromosome genotype (see Chapter 4). Thus, the circuit layout of a complex circuit can be defined as follows: $F_2(\mathbb{C}) = 0.1F_2(\mathbb{C}')$.

The logic gates in the new chromosome genotype are located according to the principle shown in Fig. 6.5. The outputs of logic gates in sub-circuits $\mathbb{C}_1$, $\mathbb{C}_2$ and $\mathbb{C}_3$ are re-encoded as shown in Fig. 6.4. The logic gates are directly mapped to the chromosome genotype. Let us remember that the logic gates located at the left part of the chromosome are the most reusable logic gates in the genotype. Therefore, the logic gates of sub-circuit $\mathbb{C}_0$ are the most reusable if the scheme shown in Fig. 6.5 is applied to generate the new chromosome genotype. This means that the logic gates in sub-circuits $\mathbb{C}_0$ and $\mathbb{C}_1$ have a higher possibility of being re-used if the circuit structure is optimised. Note that in this circuit genotype, there is not much redundancy between multiplexers. So, the generated new chromosome genotype can be characterised by the following properties: (1) 5% of redundant logic gates; (2) the sub-circuits are assembled according to the principle shown in 6.5; (3) the building block can implement a multiplexer since the multiplexer logic function is included in functional set of logic gates.

The assembled new chromosome specifies the initial population in evolution towards an optimised system. The evolution is executed during 10000 generations. Consequently, the circuit illustrated in Fig. 6.6 is evolved. The size of the circuit is reduced by 4 logic gates. Let us compare the circuit structures shown in Fig. 6.4 and Fig. 6.6. First it is necessary to notice that the connectivity of multiplexers, implementation of sub-circuit $\mathbb{C}_0$ remain the same. In sub-circuit $\mathbb{C}_1$ the connections of logic gate labeled 23 are changed. The second input of this logic gate employs the sub-function generated in sub-circuit $\mathbb{C}_0$. More changes can be noticed in the sub-circuit $\mathbb{C}_2$. Thus, the logic gate labelled 38 has become redundant. Massive changes are made in terms of connections of logic gates inside this sub-circuit. Considering

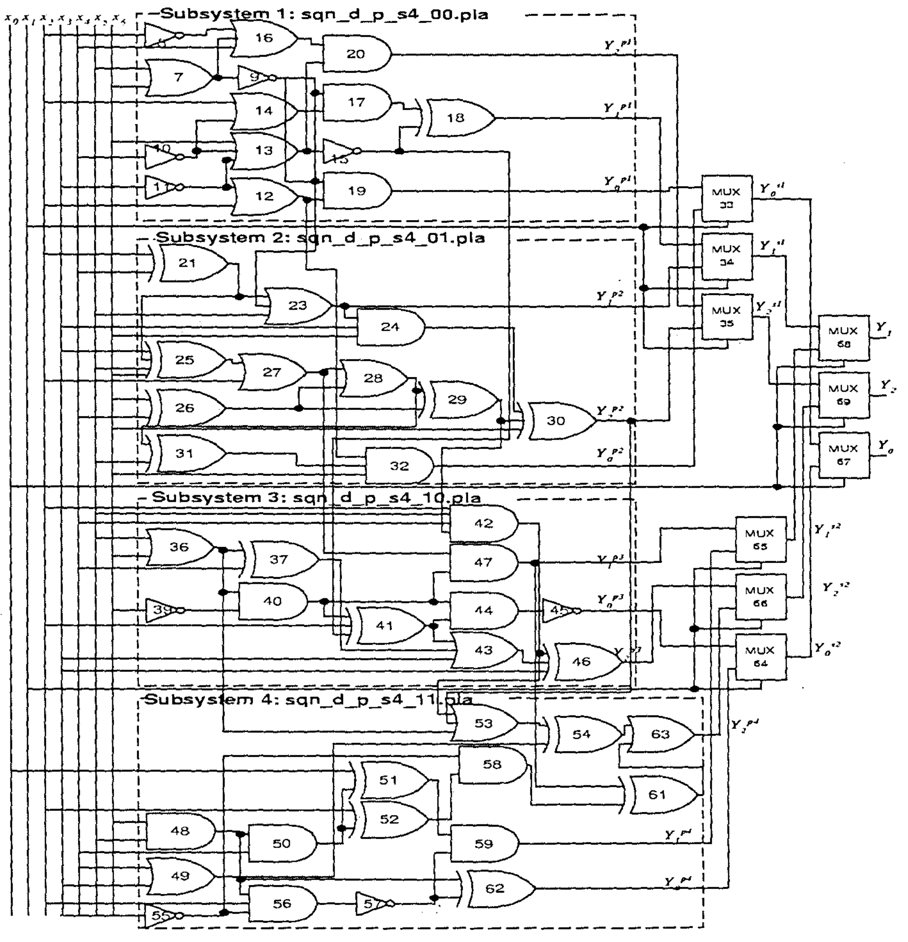

Subsystem 1: sqn_d_p_s4_00.pla
Subsystem 2: sqn_d_p_s4_01.pla
Subsystem 3: sqn_d_p_s4_10.pla
Subsystem 4: sqn_d_p_s4_11.pla
MUX 33
MUX 34
MUX 35
MUX 68
MUX 69
MUX 67
MUX 65
MUX 66
MUX 64

Figure 6.6: sqn_d.pla: Designed circuit using Shannon's decomposition and function-level EHW (Stage 2. Evolution towards an optimised system). The number of primitive logic gates: 93.

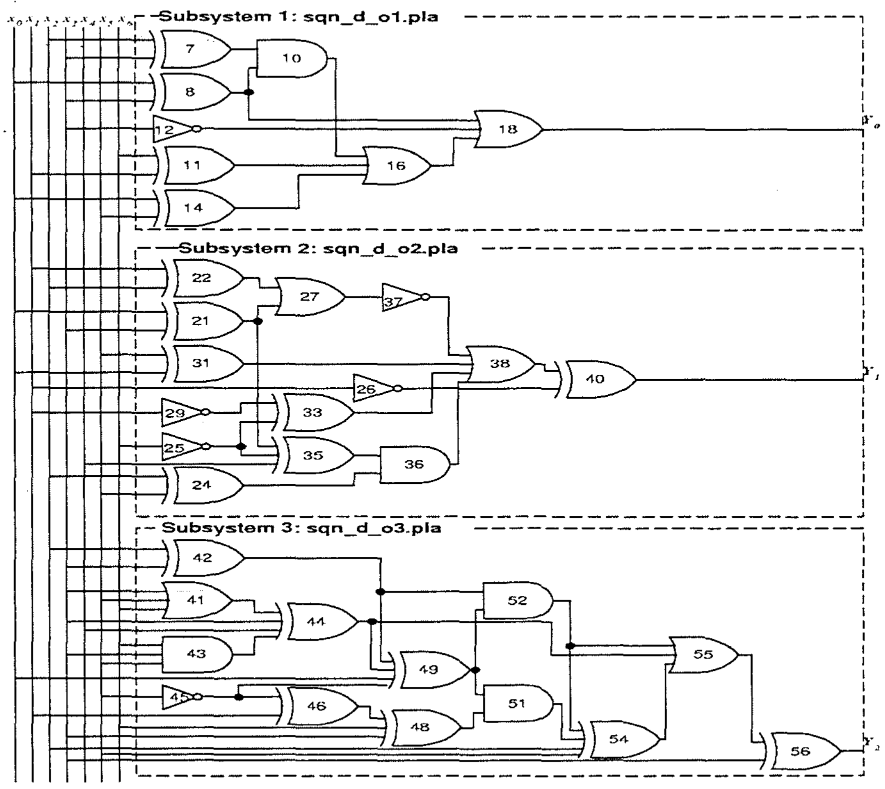

Subsystem 1: sqn_d_o1.pla
Subsystem 2: sqn_d_o2.pla
Subsystem 3: sqn_d_o3.pla

Figure 6.7: sqn_d.pla: Circuit design using BIE with output decomposition and function-level EHW (Stage 1. Evolution toward a modularised system). The number of primitive logic gates: 50.

the new structure of sub-circuit $\mathbb{C}_3$, one can easily notice that the logic gates are rearranged, and some of logic gates are connected to logic gates in sub-circuits $\mathbb{C}_0$, $\mathbb{C}_1$ and $\mathbb{C}_3$. Thus, the circuit structure has been changed drastically in order to reduce the size of logic circuit by 4 logic gates.

### 6.3.4 Experiment B: BIE with output decomposition

In this experiment, we will consider how the logic circuits are evolved using BIE with decomposition by outputs.

### sqn_d.pla

The sqn_d.pla circuit shown in Fig. 6.7 has been evolved using BIE with output decomposition. In this case, each output has been evolved separately as it has been suggested in [55], [54]. This is particular case of the increased complexity evolution, that performs as a first stage of BIE. The experimental results of evolving each of the sub-functions during 100 runs are summarised in Table 6.2. It is necessary to note that the function-level EHW evolves the output $Y_2$ with some difficulty (only 7 fully functional solutions have been achieved out of 100 runs). At the same time the outputs $Y_1$ and $Y_2$ have been evolved very easy.

**Stage 1: Evolution towards a modularised system** Let us consider the circuit structure evolved during evolution towards a modularised system. The circuit shown in Fig. 6.7 is assembled from the sub-circuits evolved separately according to the principle discussed in the previous section. The difference is that the decomposition by output does not require additional logic gates to assemble the evolved sub-circuits together. The logic gates in sub-circuits $\mathbb{C}_1$ and $\mathbb{C}_2$ are re-encoded according to their new location in chromosome genotype. 5% of logic gate redundancy is used to generate the fully functional circuit.

Analysing this circuit we found that 3 logic gates can be removed from the final circuit: (cells 2 and 3 (sub-system 2), cell 1 (sub-system 3). Therefore, the circuit implementing the sqn_d.pla function and shown in Fig. 6.7 requires 46 primitive logic gates instead of 49. Comparing the circuits evolved using BIE with Shannon's and output decompositions, we notice that the most compact logic circuit is evolved when the complex circuit is decomposed by outputs. Thus, the optimised circuit evolved using BIE with Shannon's decomposition contains 93 logic gates. The modularised

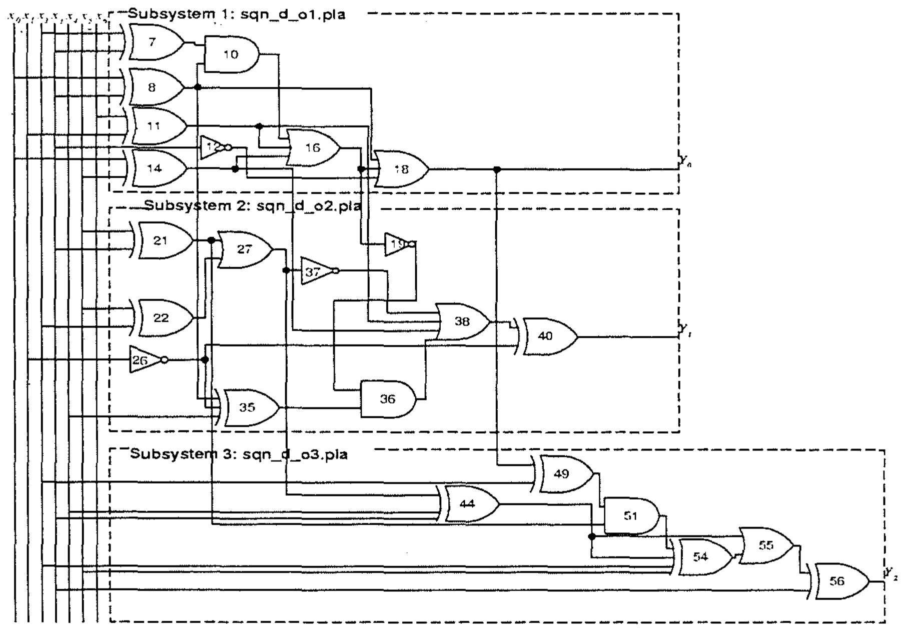

Subsystem 1: sqn_d_o1.pla
Subsystem 2: sqn_d_o2.pla
Subsystem 3: sqn_d_o3.pla
7
10
8
11
14
16
18
21
27
37
22
26
38
40
36
35
49
44
51
54
55
56
$y_0$
$y_1$
$y_2$

Figure 6.8: sqn_d.pla: Circuit design using BIE with output decomposition and function-level EHW (Stage 2. Evolution towards an optimised circuit). The number of primitive active logic gates: 34.

logic circuit sqn_d.pla evolved using BIE with decomposition by outputs consists of 50 logic gates. So, clearly we can see that in this particular case the BIE with decomposition by outputs performs much better than the BIE with Shannon's decomposition.

**Stage 2: Evolution towards an optimised system** The initial population is generated from the chromosome discussed in the previous sub-section. Approximately 5% of logic gate redundancy is used in the circuit structure. As a result of evolution towards an optimised system, the logic circuit shown in Fig. 6.8 has been evolved. This circuit contains 34 logic gates. So, the size of final logic circuit has been reduced by 16 logic gates. Analysing the circuit structure evolved we can notice that the logic gates, located in sub-circuit $\mathbb{C}_0$ are actively used in sub-circuits $\mathbb{C}_1$ and $\mathbb{C}_2$. Similar

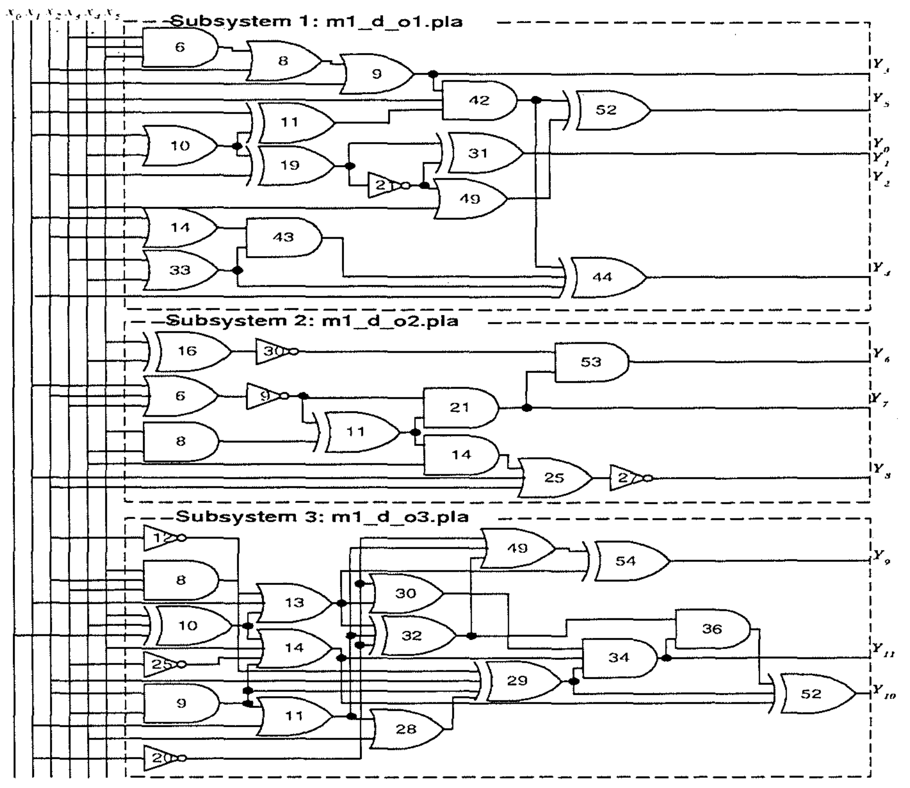


Figure 6.9: m1_d.pla: Circuit design using BIE with output decomposition and function-level EHW (Stage 1: Evolution towards a modularised system). The number of primitive logic gates is 61.

conclusions can be made about the re-usable features of logic gates located in the sub-circuit $\mathbb{C}_1$. The structure of sub-circuit $\mathbb{C}_2$ is totally changed compared to the one shown in Fig. 6.7. Thus, the logic gates in sub-circuits $\mathbb{C}_0$ and $\mathbb{C}_1$ are actively used in sub-circuits $\mathbb{C}_2$ and $\mathbb{C}_3$. Therefore, we can conclude that the final structure of evolved fully functional circuit depends on how the sub-circuits within the new chromosome genotype are assembled. Thus, if the sub-circuits are assembled according to the principle shown in Fig. 6.5, then the logic gates in sub-circuits located in the left part of the chromosome are more often used in the rest of the circuit.

**m1_d.pla**

In order to demonstrate more clearly how the BIE with decomposition by outputs works, the m1_d.pla function has been evolved (see Fig. 6.9). In this case each synthesised sub-circuit contains more than one output. This function of 6 inputs and 12 outputs has 32 input combinations. Three subsystems have been obtained after applying the output decomposition. The sub-system 1 has 6 outputs, the sub-system 2 has 3 outputs and the sub-system 3 has also 3 outputs. Each sub-system has been evolved separately during 100 runs. The most efficient circuit structures evolved have been included in final circuit shown in Fig. 6.9. Four logic gates have been proved to be redundant: cell 3 (sub-system 1), cells 1 and 3 (sub-system 2), cell 5 (sub-system 3). Hence, the circuit requires 57 primitive logic gates.

This example shows that each sub-circuit decomposed by outputs can contain more than one output and can be successfully evolved. It is necessary to point out that no fully functional circuits have been evolved for the functions mentioned above using function-level EHW even after 150 000 generations ($3 \times 50000$ generations for three sub-circuits generated using BIE).

## 6.4 BIE with EHW-oriented decomposition

The main advantage of the method is that evolution is not carried out in one operation on the complete evolvable hardware unit, but rather is a bottom-up and a top-down ways. The number of inputs and outputs, in an evolved sub-system can be limited to allow faster evolution. The two main stages of evolution are discussed in detailed in the following sub-sections.

### 6.4.1 Stage 1. Evolution towards a modularised system using EHW-oriented decomposition

EHW-oriented decomposition discovers the evaluation tasks as well as their sequence automatically. The main idea of this approach is to define the *easily evolved* sub-functions and evolve them separately. Note that the fully functional or nearly fully functional design has to be obtained for each decomposed sub-function.

The truth table of an $n$-input $m$-output logic function given for $p$ input combinations contains an input matrix $I(n \times p)$ and an output matrix $O(m \times p)$ and can be described by this pair of matrices as $(I(n \times p), O(m \times p))$. The logic function is completely specified if $p = 2^n$, i.e. it is given on *all* input combinations. For instance, the completely specified 6-input 7-output logic function can be described by truth table $(I(6, 64), O(7, 64))$, where $n = 6$, $m = 7$ and $p = 64$.

The following metrics can be used in order to define the quality of an evolved circuit.

The percentage of correct output bits corresponding to the $j$-th output, $\mathbf{y_j}$ is calculated as follows:

$$f_{\mathbf{y_j}^o} = \frac{\sum_{i=1}^{p} |y_j - d_j|}{p} * 100; \qquad (6.4.1)$$

where $|y_i - d_i|$ is the absolute difference between the actual output $y_i$ and the desired output $d_i$; $\mathbf{y_j}$ is the vector of the $j$-th circuit output. If $f_{\mathbf{y_j}^o} = 100.0$ the circuit implements the output $y_j$ completely.

The percentages of correct output bits in the $j$-th output for input combinations with $x_i = 0$ and $x_i = 1$ are computed as follows:

$$f_{\mathrm{y_j}^o}|_{x_i=0} = \frac{\sum_{i=1}^{p_{x_i=0}} |y_j - d_j|}{p_{x_i=0}} * 100; \tag{6.4.2}$$

$$f_{\mathrm{y_j}^o}|_{x_i=1} = \frac{\sum_{i=1}^{p-p_{x_i=0}} |y_j - d_j|}{p - p_{x_i=0}} * 100; \tag{6.4.3}$$

where $p_{x_i=0}$ is the number of input combinations in the truth table with $x_i = 0$. If $f_{\mathrm{y_j}^o}|_{x_i=0} = 100.0$, the circuit completely implements the truth table generated from the output $y_j$ with condition that $x_i = 0$.

The EHW-oriented decomposition is performed to evolve an $n$-input $m$-output logic function for a given $p$ input combinations as follows:

1. Define the termination condition of evolution, for example, the number of generations $N_{gen}$.

2. Evolve the initial complex system implementing the truth table $T_0 = (I(n \times p), O(m \times p))$ during $N_{gen}$ (gate- or function-level EHW with randomly generated initial population and with circuit layout evolution).

3. Keep the result of evolution: the genotype of the best chromosome.

4. Calculate $f^o_{\mathrm{y_j}}$, $(j = 1, ..., m)$ and $f_{\mathrm{y_j}^o}|_{x_i=0}$, $f_{\mathrm{y_j}^o}|_{x_i=1}$, $(i = 1, ..., n)$.

5. Choose the output partitioning vector $\mathbf{v_y}$ defined by the circuit outputs with higher $f^o_{\mathrm{y_j}}$;

6. IF $p$ is large, THEN choose the input partitioning vector $\mathbf{v_i}$ defined by the higher value of $f_{\mathrm{y_j}^o}|_{x_i=0}$ or $f_{\mathrm{y_j}^o}|_{x_i=1}$ for the circuit outputs specified in step 5, generate the product partitioning vector $\mathbf{v_p}$ determined by input combinations with $x_i = 0$ or $x_i = 1$, ELSE Go to step 7.

7. Generate the 3 (if step 6 has been executed) or 2 (if step 6 has not been executed) truth tables of sub-functions: (1) The easily evolved function is defined by the truth table with $\mathbf{v_o}$ and $\mathbf{v_p}$ partitioning vectors: $T_1 = (I(n \times |\mathbf{v_p}|), O(|\mathbf{v_o}| \times |\mathbf{v_p}|))$; (2) The second sub-system contains the remaining output combinations for the best chosen product partitioning vector $\mathbf{v_p}$: $T_2 = (I(n \times p - |\mathbf{v_p}|), O(|\mathbf{v_o}| \times p - |\mathbf{v_p}|))$; (3) The third sub-system contains the remaining data from the truth table $T_0$: $T_3 = (I(n \times |\mathbf{v_p}|), O(m - |\mathbf{v_o}| \times |\mathbf{v_p}|))$.

8. Evolve separately the sub-systems defined in the previous step. The initial population of the sub-system described by the truth table $T_1$ is the final population of EHW executed at step 2. The sub-system $T_2$ is evolved using either the final population obtained at step 2 or a randomly generated population. The initialisation process depends on the parameters $f^o_{\mathbf{y_j}}$, $f_{\mathbf{y_j}^o}|_{x_i=0}$ and $f_{\mathbf{y_j}^o}|_{x_i=1}$ for this part of the truth table in $T_0$. The sub-system $T_3$ is evolved using the randomly generated initial population during $N_{gen}$ generations.

9. IF the complexity levels for $T_2$ and $T_3$ are not sufficient, THEN Consider the sub-system as a complex system and proceed to steps 3-8 for each sub-system, ELSE Go to step 10.

10. The evolvable sub-systems are generated.

The sub-system $S_1(T_1)$ is generated with the best metrics. If these metrics are equal to 100, the fully functional circuit can be extracted from the system $S_0(T_0)$. In other cases, the initial population is generated based on chromosome genotype specified by $S_0(T_0)$. The fitness function of chromosomes in the initial population is relatively high, since it is defined by chosen metrics. Therefore, the sub-system

$S_1(T_1)$ is easily evolvable. In this case the genetic material obtained during the previous evolutionary process of complex task is used in the evolution of less complex task.

The diagram of EHW-oriented decomposition described above is given in Fig. 6.10. The block "EVOLVE" contains two inputs: (1) The truth table, $T$ specifying which function is evolved during the evolutionary process, and (2) Initialisation to define how the initial population is generated. The output of this block contains the genotype of the best evolved chromosome $\mathbb{C}_{bc}(T)$. This chromosome will be used to generate the initial population. This initial population further participates in the evolution of easily evolved sub-systems. The initial population of sub-system $S_3$ is generated randomly, because the truth table $T_3$ contains the worst values of $f_{\mathrm{y_j}^o}$ and either $f_{\mathrm{y_j}^o}|_{x_i=0}$ or $f_{\mathrm{y_j}^o}|_{x_i=10}$ is taken from truth table $T_0$. This also brings some diversity in evolution of the complex system and allows the evolution to find a better "start" point for the next decomposed functions.

In the scheme of EHW-oriented decomposition the evolution is terminated after a fixed number of generations, $N_{gen}$. We chose this termination condition because the fitness function is usually improved significantly during the first generations. Alternatively, the evolution can be terminated once the fitness value $(\mathcal{F}_{start} + \Delta\mathcal{F})$ is achieved, where $F_{start}$ is the best fitness generated at the initial population; $\Delta F$ is the value on which the fitness function is expected to be improved.

Another problem that can arise in EHW-oriented decomposition is the definition of circuit layout for each sub-task that can be solved. This can be avoided by using the circuit layout evolution together with circuit functionality proposed first in [1].

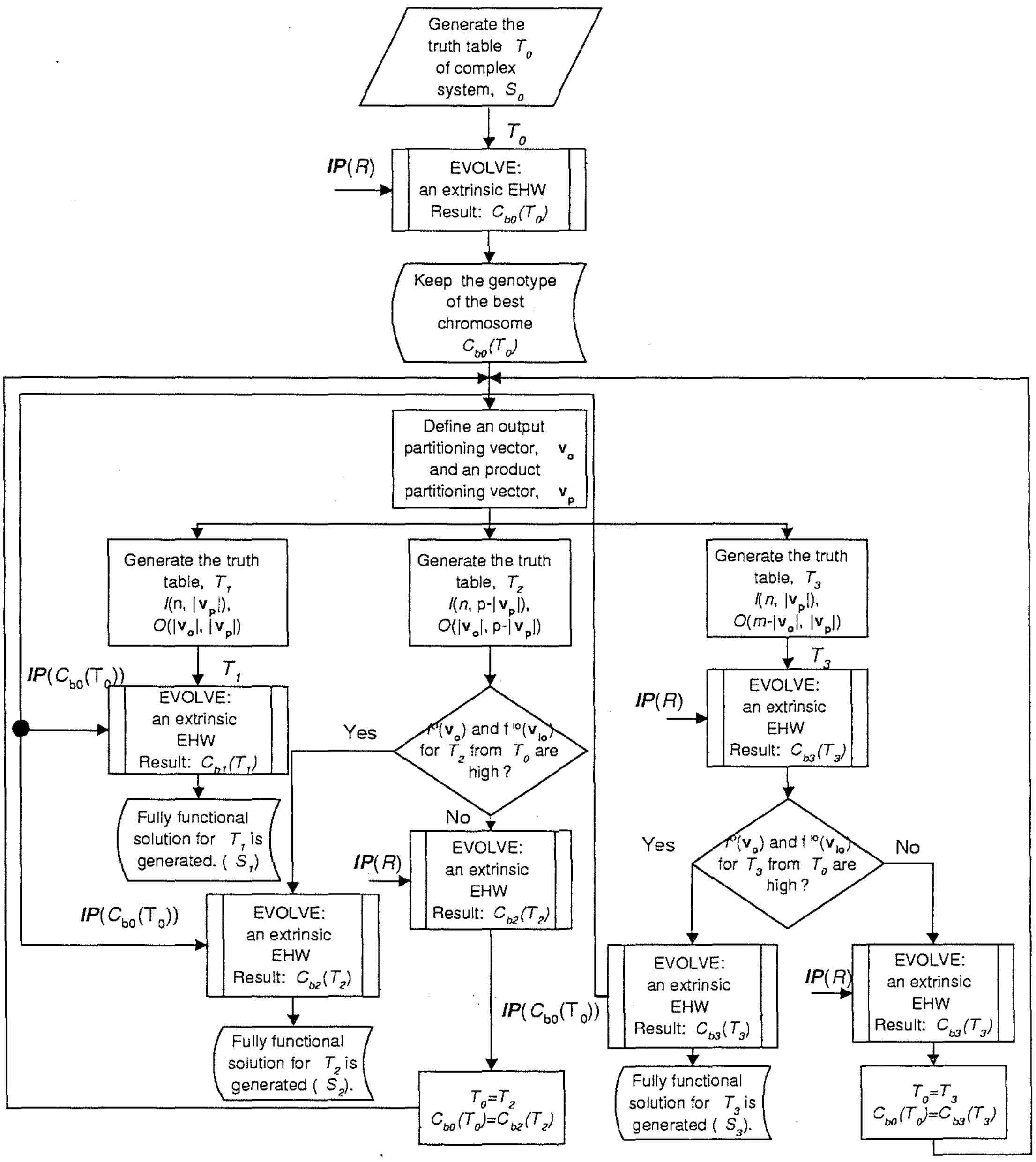

Generate the truth table $T_0$ of complex system, $S_0$
$T_0$
IP(R)
EVOLVE: an extrinsic EHW Result: $C_{b0}(T_0)$
Keep the genotype of the best chromosome $C_{b0}(T_0)$
Define an output partitioning vector, $\mathbf{v}_o$ and an product partitioning vector, $\mathbf{v}_p$
Generate the truth table, $T_1$ $I(n, |\mathbf{v}_p|)$, $O(|\mathbf{v}_o|, |\mathbf{v}_p|)$
Generate the truth table, $T_2$ $I(n, p-|\mathbf{v}_p|)$, $O(|\mathbf{v}_o|, p-|\mathbf{v}_p|)$
Generate the truth table, $T_3$ $I(n, |\mathbf{v}_p|)$, $O(m-|\mathbf{v}_o|, |\mathbf{v}_p|)$
$IP(C_{b0}(T_0))$
$T_1$
EVOLVE: an extrinsic EHW Result: $C_{b1}(T_1)$
Fully functional solution for $T_1$ is generated. ( $S_1$)
Yes
$f^{io}(\mathbf{v}_o)$ and $f^{io}(\mathbf{v}_{io})$ for $T_2$ from $T_0$ are high ?
No
IP(R)
EVOLVE: an extrinsic EHW Result: $C_{b2}(T_2)$
$IP(C_{b0}(T_0))$
EVOLVE: an extrinsic EHW Result: $C_{b2}(T_2)$
Fully functional solution for $T_2$ is generated ( $S_2$).
$IP(C_{b0}(T_0))$
$T_0=T_2$ $C_{b0}(T_0)=C_{b2}(T_2)$
$T_3$
IP(R)
EVOLVE: an extrinsic EHW Result: $C_{b3}(T_3)$
Yes
$f^{io}(\mathbf{v}_o)$ and $f^{io}(\mathbf{v}_{io})$ for $T_3$ from $T_0$ are high ?
No
EVOLVE: an extrinsic EHW Result: $C_{b3}(T_3)$
Fully functional solution for $T_3$ is generated ( $S_3$).
IP(R)
EVOLVE: an extrinsic EHW Result: $C_{b3}(T_3)$
$T_0=T_3$ $C_{b0}(T_0)=C_{b3}(T_3)$

Figure 6.10: The diagram of EHW-oriented decomposition. **IP** is the initial population; **IP**(R) denotes the randomly generated initial population; $T_i$ is the $i$-th truth table; $C_{bi}(T_i)$ is the best chromosome genotype evolved using an extrinsic EHW for the function given by the truth table $T_i$; $\mathbf{IP}(C_{bi}(T_i))$ is the initial population generated using the best chromosome genotype obtained after the evolutionary process for the $i$-th truth table.

### 6.4.2 Stage 2. Evolution towards an optimised system

The assembling of the system is based on the specific features of the output and functional decompositions. For example, the output decomposition guarantees that each sub-system is synthesized separately and it is *completely* independent. In the case of functional decomposition, the corresponding outputs generated for different input combinations in different sub-systems have to be connected together using an one-control multiplexer. An analysis of experimental results shows that it is reasonable to assemble the sub-systems decomposed by functional decomposition first and then the sub-systems separatly using output decomposition.

### 6.4.3 Experiment C: BIE with output decomposition

In this section we will consider how the bidirectional incremental evolution performs. The 7-input 10-output logic function (z5xp1_d.pla) has been evolved using both direct evolution and bidirectional incremental evolutions. Hence, $n = 7$, $m = 10$ and $p = 128$. This function is taken from standard benchmark library for combinational logic design. The search has been performed using a rudimentary $(1 + \lambda)$ evolutionary strategy with dynamic fitness function, uniform mutation and population of 5 individuals.

In bidirectional incremental evolution, the system is first evolved towards its modularisation. Once the system has been decomposed into sub-systems with sufficient complexity, the system is evolved towards its optimisation. These two processes are considered in detail in the following sub-sections for two types of experiments.

In this experiment the complex system is divided into sub-systems by means of output decomposition only.

**Stage 1. Evolution towards a modularised system** The evolution of sub-systems is performed during no more than 15000 generations.

During this stage of bidirectional incremental evolution, the circuit structure remains the same, but the evaluation parameters are changed. For instance, the chromosome $\mathbb{C}$ is evaluated by truth table $T_1$. $T_1$ can be decomposed into smaller truth tables $T_2$ and $T_3$. In this case, the chromosome $\mathbb{C}$ can be estimated by truth tables $T_2$ and $T_3$. Obviously, in this case the evaluation by one of the tables will show better results.

The bidirectional incremental evolution begins with attempts to evolve the complex system. The evolved circuits are evaluated by a given truth table of 7 inputs and 10 outputs. The evolution is terminated after 5000 generations and the best chromosome defines the first generated sub-system $S_0$ that is evaluated according to the metrics given in Eq. 6.4.1– Eq. 6.4.2. The result of this evaluation is given in Table 6.5.

Analysing these data we observe that $f^o_{y_7} = f^o_{y_8} = f^o_{y_9} = 100.0$. This means that the circuit $S_0$ is fully functional if it is evaluated by the truth table $T_1$ generated from all input combinations of outputs $y_7$, $y_8$ and $y_9$. Therefore, we can define the first sub-system that can be evaluated according to truth table $T_1$. From this two truth tables $T_1$ and $T_2$ can be generated.

Because the circuit $S_0$ evaluated by the truth table $T_1$ is fully functional, the evolution of system $S_1$ can start with an initial population generated from the circuit $S_0$. In this case, the evolutionary process will tend to reduce the number of active gates in the circuit.

Since $f^o_{y_j}$ for the outputs $y_0$, $y_1$, $y_2$, $y_3$, $y_4$, $y_5$ and $y_6$ are not equal to 100.0, the circuit $S_1$ evaluated by the truth table $T_2$ is not fully functional. Therefore, the next sub-systems have to be defined.

As a result of EHW-oriented decomposition the following sub-systems can be defined:

1. $S_0$: $T_0 = (I(7,128), O(10,128))$; $Y = \{y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9\}$. *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: Random. Best chromosome - $\mathbb{C}_0$. **Conclusion**: Decomposition by outputs: $S_1$ - easily evolved; $S_2$.

2. $S_1$: $T_1 = (I(7,128), O(3,128))$; $Y = \{y_7, y_8, y_9\}$. *Purpose of evolution:* reduce the number of active gates in circuit; *Initialisation* - $\mathbb{C}_0$ (fully functional circuit). Best chromosome - $\mathbb{C}_1$. **Conclusion**: a fully functional circuit with reduced number of active gates is evolved.

3. $S_2$: $T_2 = (I(7,128), O(7,128))$; $Y = \{y_0, y_1, y_2, y_3, y_4, y_5, y_6\}$. *Purpose of evolution:* evolving fully functional circuit; *Initialisation:* Random. Best chromosome - $\mathbb{C}_2$. **Conclusion**: Decomposition by outputs: $S_3$ - easily evolved; $S_4$.

4. $S_3$: $T_3 = (I(7,128), O(3,128))$; $Y = \{y_0, y_5, y_6\}$. *Purpose of evolution:* reduce the number of active gates in circuit; *Initialisation* - $\mathbb{C}_2$ (fully functional circuit). Best chromosome - $\mathbb{C}_3$. **Conclusion**: a fully functional circuit is evolved.

5. $S_4$: $T_4 = (I(7,128), O(4,128))$; $Y = \{y_1, y_2, y_3, y_4\}$. *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: Random. Best chromosome - $\mathbb{C}_4$. **Conclusion**: Decomposition by outputs: $S_5$ - easily evolved; $S_6$.

6. $S_5$: $T_5 = (I(7,128), O(1,128)); Y = \{y_1\}$; *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: $\mathbb{C}_4$. Best chromosome - $\mathbb{C}_5$. **Conclusion**: A fully functional circuit with reduced number of active logic gates is evolved.

7. $S_6$: $T_6 = (I(7,128), O(3,128))$; $Y = \{y_2, y_3, y_4\}$. *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: Random. Best chromosome - $\mathbb{C}_6$. **Conclusion**: Decomposition by outputs: $S_7$ - easily evolved; $S_8$.

8. $S_7$: $T_7 = (I(7,128), O(1,128)); Y = \{y_2\}$; *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: $\mathbb{C}_6$. Best chromosome - $\mathbb{C}_7$. **Conclusion**: A fully functional circuit with reduced number of active logic gates is evolved.

9. $S_8$: $T_8 = (I(7,128), O(1,128)); Y = \{y_3\}$; *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: $\mathbb{C}_6$. Best chromosome - $\mathbb{C}_8$. **Conclusion**: A fully functional circuit with reduced number of active logic gates is evolved.

10. $S_9$: $T_9 = (I(7,128), O(1,128)); Y = \{y_4\}$; *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: $\mathbb{C}_6$. Best chromosome - $\mathbb{C}_9$. **Conclusion**: A fully functional circuit with reduced number of active logic gates is evolved.

Table 6.3: z5xp1_d.pla: Metrics of sub-systems obtained during evolution. The systems are decomposed according to the metrics shown in bold. For example, after analysis of the system $S_2$, the easily evolved system $S_3$ is composed from the outputs $y_0$, $y_5$ and $y_6$. The remaining outputs are evaluated in the following system $S_4$.

| $S_j$ | $f^o_{y_j}$ | | | | | | |
|---|---|---|---|---|---|---|---|
| | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ | $y_6$ |
| $S_2$ | **99.22** | 89.06 | 88.28 | 70.31 | 59.37 | **84.37** | **93.75** |
| $S_4$ | | **92.18** | 85.19 | 83.59 | 87.5 | | |
| $S_6$ | | | **88.28** | **95.31** | **90.62** | | |

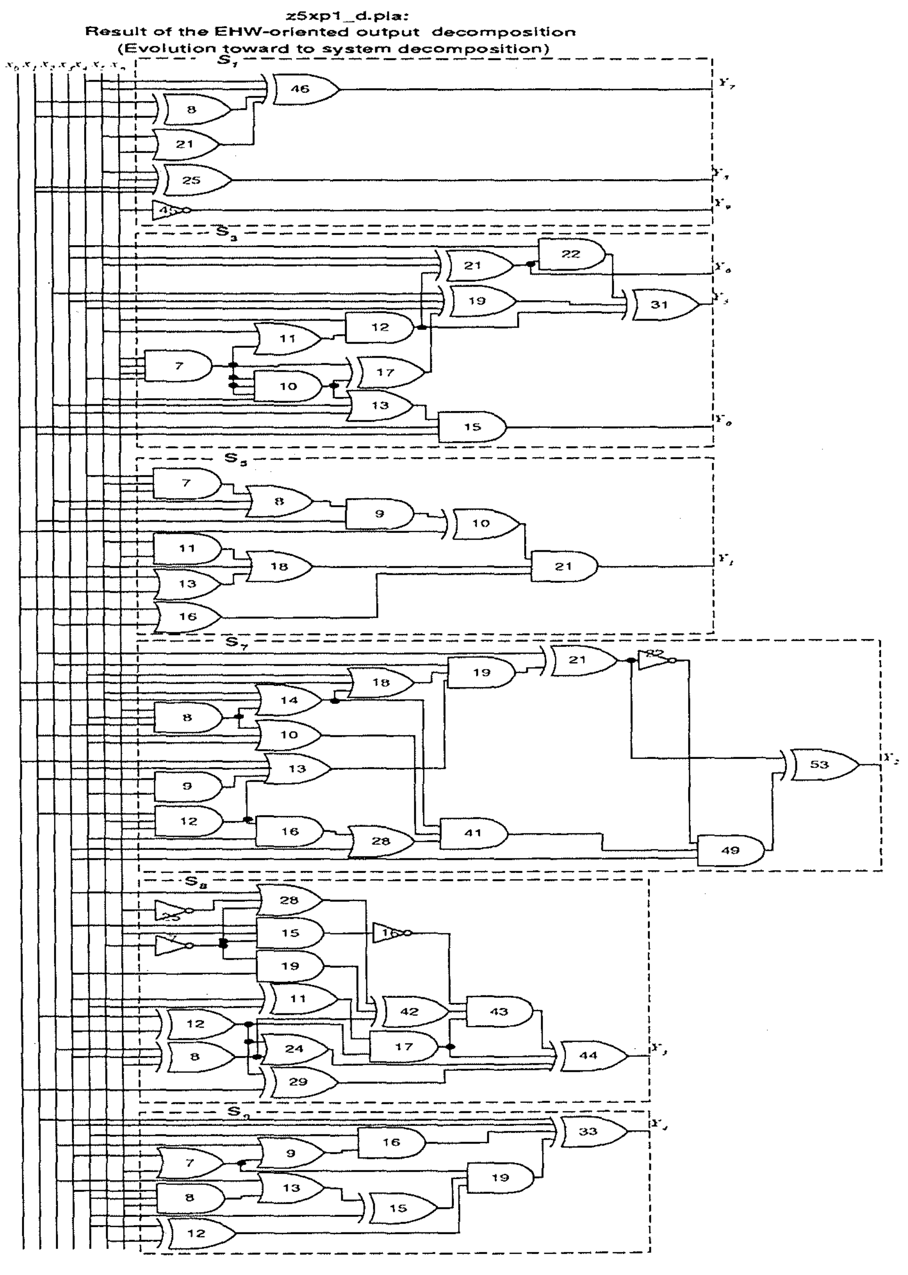

z5xp1_d.pla:
Result of the EHW-oriented output decomposition
(Evolution toward to system decomposition)
S1
S3
S5
S7
S8
S9
46
8
21
25
22
19
31
12
11
7
17
10
13
15
9
18
16
14
28
41
53
49
42
43
44
24
29
33

Figure 6.11: z5xp1_d.pla: Bidirectional incremental evolution (Stage 1: Evolution towards to a modularised system).

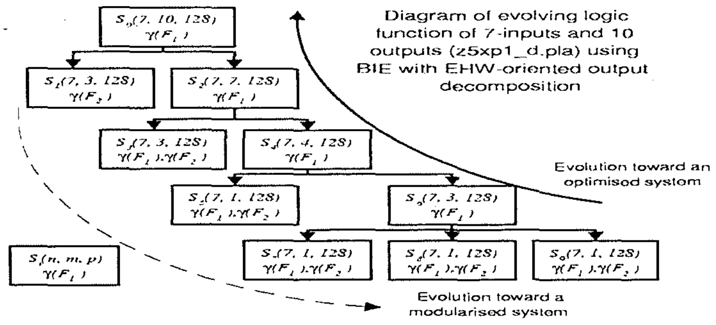

$S_0(7, 10, 128)$ $\Upsilon(F_1)$
$S_1(7, 3, 128)$ $\Upsilon(F_2)$
$S_2(7, 7, 128)$ $\Upsilon(F_1)$
$S_3(7, 3, 128)$ $\Upsilon(F_1).\Upsilon(F_2)$
$S_4(7, 4, 128)$ $\Upsilon(F_1)$
$S_5(7, 1, 128)$ $\Upsilon(F_1).\Upsilon(F_2)$
$S_6(7, 3, 128)$ $\Upsilon(F_1)$
$S_7(7, 1, 128)$ $\Upsilon(F_1).\Upsilon(F_2)$
$S_8(7, 1, 128)$ $\Upsilon(F_1).\Upsilon(F_2)$
$S_9(7, 1, 128)$ $\Upsilon(F_1).\Upsilon(F_2)$
$S_i(n, m, p)$ $\Upsilon(F_1)$
Diagram of evolving logic function of 7-inputs and 10 outputs (z5xp1_d.pla) using BIE with EHW-oriented output decomposition
Evolution toward an optimised system
Evolution toward a modularised system

Figure 6.12: z5xp1_d.pla: Diagram of performing a bidirectional incremental evolution (Experiment C). The $i$-th system $S_i$ is evaluated according to the truth table of $n$ inputs, $m$ outputs and $p$ input-output combinations. This evaluation process can be defined as $S_0(n, m, p)$. Dynamic fitness function [1] contains evaluation of: (1) circuit functionality, $F_1$; (2) the number of logic gates used in the circuit, $F_2$. $\Upsilon(F_j)$ defines the evaluation process performed using criteria $F_j$.

Once the EHW-oriented decomposition is complete, the fully functional circuit can be generated as shown in Fig. 6.11. This can be done if the synthesis of the optimised circuit is not required to be performed. Since, only output decomposition is allowed in this example, each of the sub-circuits is implemented independently and realizes the set of outputs from the complex system. Obviously, this circuit structure is not optimal, since the circuit optimisation has not been performed.

So, the bidirectional incremental evolution schedule can be defined as a decision tree and can be summarised as shown in Fig. 6.12. This sequence of tasks forces the evolution first to develop its partitioning vectors and then to optimise the structures of assembled sub-tasks. The mapping of the performance of these tasks into the scale of evolutionary process is shown in Fig. 6.13. Note that once stage 1 of the bidirectional incremental evolution is completed, the fully functional circuit can be assembled without any optimisation process. Fig. 6.13 depicts the performance of direct and bidirectional incremental evolution. The comparison is made for the evolutionary process when the circuit functionality has been evolved. Therefore, the evolution of sub-system $S_1$ is not taken into account, since it is fully functional after the evolution of system $S_0$. The fully functional circuit can be generated after 75 000 generations. During these generations, the problems with less complexity have been evolved. The fully functional circuit implemented z5xp1_d.pla has been evolved using direct evolution after 5 000 000 generations. The best functionality circuit evolved after 150 000 generations using direct evolution is 91.40625. That is relatively low. Also, during the first stage of bidirectional incremental evolution, the size of the fully functional circuit evolved has been reduced from 157 logic gates to 85 (see Table 6.4). This shows that even during the first stage of bidirectional incremental evolution the size of circuit can be reduced if a dynamic fitness function is used in the evaluation process.

**Stage 2. Evolution towards an optimised system** The main purpose of this evolution stage is to optimise the size of the evolved circuit. During this stage of bidirectional incremental evolution, the sub-circuit of larger complexity is assembled from different sub-circuits of lower complexity, and the truth table of this circuit is also assembled from the two truth tables describing the behaviour of sub-systems. Evolution is performed under the new chromosome genotype with new evaluation criteria.

For example, the sub-circuits $\mathbb{C}(T_2)$ and $\mathbb{C}(T_3)$ evaluated by truth tables $T_2$ and $T_3$ are evolved separately. More complex task can be defined by the truth table $T$ that is assembled from the truth tables $T_2$ and $T_3$: $T \in T_2 \cup T_3$. This task is described by chromosome $\mathbb{C}(T_2, T_3)$. The fully functional circuit implemented the truth table $T$ is generated based on the sub-circuits $\mathbb{C}(T_2)$ and $\mathbb{C}(T_3)$. In this case, the chromosome genotype $\mathbb{C}(T_2, T_3)$ is synthesized based on chromosome genotypes $\mathbb{C}(T_2)$ and $\mathbb{C}(T_3)$.

The truth tables of these two sub-circuits are assembled into a larger one. The generated new genotype creates the initial population for the optimisation process. The evolutionary process is first undertaken on the new generated circuit. Then a circuit of larger complexity is generated and evolution repeats again. The process will continue until the complex circuit is assembled and optimised using evolutionary process.

This process can also reduce the size of the evolved circuit drastically. For example, the size of circuit synthesised after the first stage of bidirectional incremental evolution has been reduced from 85 logic gates to 54.

Thus, the actual size of fully functional z5xp1 circuit has been reduced from 157 logic gates to 54 (see Table 6.4). In other words the optimisation process reduced the number of logic gates by a factor of about 3.

### 6.4.4 Experiment D: BIE with Shannon's and output decompositions

In this experiment the complex system is divided into sub-systems by means of output and Shannon's decompositions. In order to compare the differences between the evolutionary processes, the evolution of first system $S_0$ is performed for both experiments.

**Stage 1: Evolution towards modularised system** The sub-system $S_1$ is generated according to the same specification given in experiment A. Some of the metrics $f^{io}_{y_j}$ in system $S_0$ are equal to 100.0.For example, $f^{io}_{y_0}|_{x_1=0} = f^{io}_{y_1}|_{x_1=0} = 100.0$, i.e. the circuit implements completely the truth table generated from the outputs $y_0$ and $y_1$

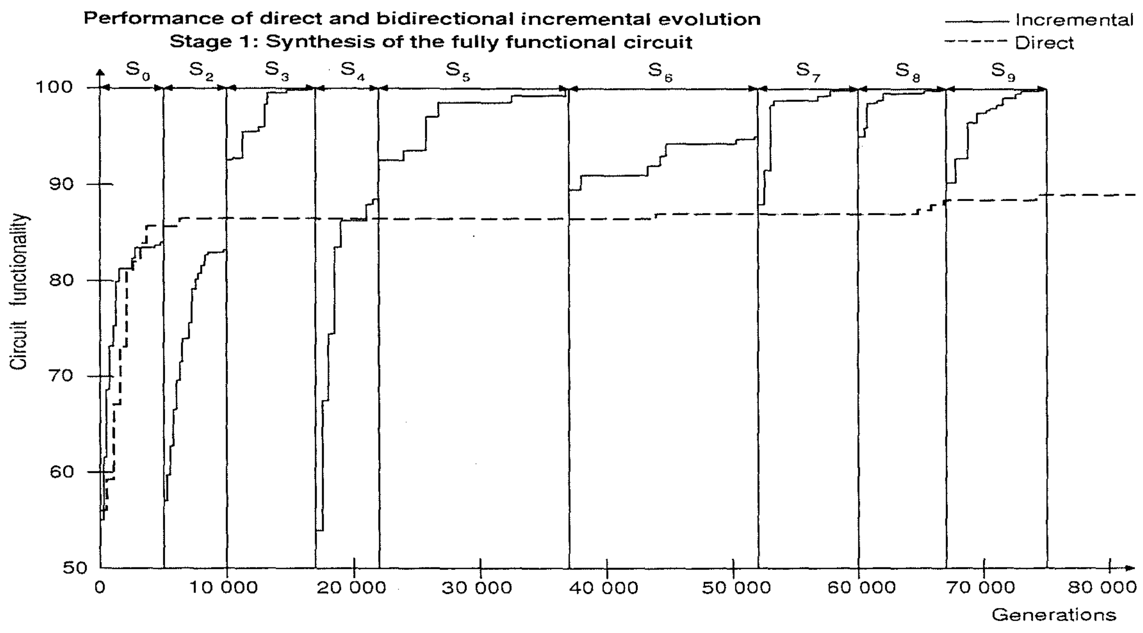


Figure 6.13: z5xp1_d.pla: Performance of direct and bidirectional incremental evolution in the circuit design problem.The maximum fitness per generation is plotted for each of the two approaches. The direct evolution (dotted line) makes slight progress at the first and stalls after about 10 000 generations. The plot is an average of 100 simulations. Incremental evolution, however, proceeds through several task transitions (seen as abrupt drop-offs in the plot), and eventually solves the goal-task. The incremental plot is a result of one simulation.

Table 6.4: z5xp1_d.pla: History of the incremental evolution with the EHW-oriented output decomposition.

| **Sub-system** | | | | | **ES performance** | | |
|---|---|---|---|---|---|---|---|
| $S_i$ | $n$ | $m$ | $p$ | $Y$ | $N_{gen}$ | $F_2(N_f)$ | $F_2(N_f^{opt})$ |
| **Direct evolution** | | | | | | | |
| $S$ | 7 | 10 | 128 | $Y$ | 150 000 | - | - |
| $F_1^{bf} = 91.40625$ | | | | | | | |
| **Stage 1: Evolution towards to a modularised system** | | | | | | | |
| $S_0$ | 7 | 10 | 128 | $Y$ | 5 000 | - | - |
| $S_1$ | 7 | 3 | 128 | $\{y_7, y_8, y_9\}$ | 5 000 | 20 | 9 |
| $S_2$ | 7 | 7 | 128 | $Y$ | 5 000 | - | - |
| $S_3$ | 7 | 3 | 128 | $\{y_0, y_5, y_6\}$ | 15 000 | 25 | 21 |
| $S_4$ | 7 | 4 | 128 | $Y$ | 5 000 | - | - |
| $S_5$ | 7 | 1 | 128 | $\{y_1\}$ | 15 000 | 23 | 17 |
| $S_6$ | 7 | 3 | 128 | $Y$ | 15 000 | - | - |
| $S_7$ | 7 | 1 | 128 | $\{y_2\}$ | 15 000 | 30 | 25 |
| $S_8$ | 7 | 1 | 128 | $\{y_3\}$ | 15 000 | 26 | 24 |
| $S_9$ | 7 | 1 | 128 | $\{y_4\}$ | 15 000 | 23 | 13 |
| **Total:** $\sum F_2(N_f^{opt})$ | | | | | 110 000 | 157 | 85 |
| **Stage 2: Evolution towards to an optimised system** | | | | | | | |
| **Total:** $\sum F_2(N_f^{opt})$ | | | | | 200 000 | 85 | 54 |

and for $x_1 = 0$. This truth table contains 6 inputs, 2 outputs and 64 input combinations. Therefore, if the Shannon's decomposition is allowed to be performed, the next fully functional sub-circuit contains the structure defined by this truth table. The truth table generated, given that $x_1 = 1$, has the same parameters. Obviously, it is easier to evolve a circuit with fewer input combinations.

Note that this is not the only possible set of sub-circuits that can be generated by analysing the metric data of $S_0$. One can decompose the system in terms of variable $x_0$, since $f_{y0}^{io}|_{x0=0} = 100.0$ or in terms of variable $x_3$, since $f_{y0}^{io}|_{x3=1} = 100.0$. We choose the partitioning vector mentioned above because in this case we divide the sub-system not only in terms of the variable $x_1$, but also here, the two outputs $y_0$ and $y_1$ can be

Table 6.5: Parameters of $S_0$ sub-system obtained during evolution of z5xp1_d.pla. The metrics specified in Eq. 6.4.1 and Eq. 6.4.2 are calculated for sub-system $S_0$. $f^o_{y_j}$ defines that the sub-function described by output $y_j$ is fully functional. Metrics $f^{io}_{y_j}|_{x_i=0}$ and $f^{io}_{y_j}|_{x_i=1}$ correspond to the Shannon's decomposition by variable $x_i$. Hence, $f^{io}_{y_j}|_{x_i=0} = f^{io}_{y_j}|_{x_i=1} = 100$ for all variables $x_i$. If $f^o_{y_j} < 100$ and $f^{io}_{y_j}|_{x_i=0} = 100$ then the sub-circuit described the sub-function with $x_i = 0$ is fully functions. The same implied for metric $f^{io}_{y_j}|_{x_i=1}$.

| $y_j$ | $f^o_{y_j}$ | $f^{io}_{y_j}$ | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $x_0$ | | $x_1$ | | $x_2$ | | $x_3$ | | $x_4$ | | $x_5$ | | $x_6$ | |
| | | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| $y_0$ | 92.9689 | **100** | 86 | **100** | 86 | 99 | 88 | 86 | **100** | 94 | 93 | 94 | 93 | 94 | 93 |
| $y_1$ | 89.0625 | 80 | 99 | **100** | 79 | 71 | 69 | 54 | 86 | 69 | 71 | 68 | 72 | 71 | 69 |
| $y_2$ | 69.5312 | 61 | 79 | 71 | 69 | 54 | 86 | 69 | 71 | 68 | 72 | 71 | 69 | 69 | 71 |
| $y_3$ | 82.8125 | 77 | 90 | 91 | 75 | 88 | 79 | 88 | 79 | 79 | 88 | 85 | 82 | 82 | 85 |
| $y_4$ | 60.9375 | 60 | 63 | 60 | 63 | 66 | 57 | 63 | 60 | 63 | 60 | 60 | 63 | 66 | 57 |
| $y_5$ | 81.25 | 82 | 82 | 82 | 82 | 82 | 82 | 75 | 88 | 88 | 75 | 88 | 75 | 75 | 88 |
| $y_6$ | 62.5 | 63 | 63 | 63 | 63 | 63 | 63 | 63 | 63 | 75 | 50 | 75 | 50 | 50 | 75 |
| $\mathbf{y_7}$ | **100.0** | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| $\mathbf{y_8}$ | **100.0** | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| $\mathbf{y_9}$ | **100.0** | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |

involved in this partitioning process.

Each sub-circuit that is relatively difficult to evolve using small number of generations has been evaluated in terms of metrics given in Eq. 6.4.1 and Eq. 6.4.2. The result of this evaluation is given in Table 6.6.

Table 6.6: Parameters of sub-systems obtained during evolution of z5xp1_d.pla using the EHW-oriented output and input decompositions (Stage 1. Evolution towards a modularised system).

| $S_k$ | $y_j$ | $f^o_{\mathbf{y}_j}$ | $f^{io}_{y_j}$ | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $x_0$ | | $x_1$ | | $x_2$ | | $x_3$ | | $x_4$ | | $x_5$ | | $x_6$ | |
| | | | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| $S_4$ | $y_2$ | 89.9438 | 93 | 88 | 93 | 88 | 88 | 93 | 94 | 86 | 86 | 94 | 90 | 91 | 91 | 90 |
| | $y_3$ | 79.6875 | 80 | 80 | 79 | 82 | 82 | 79 | 82 | 79 | 79 | 82 | 79 | 82 | 82 | 79 |
| | $y_4$ | 78.125 | 79 | 79 | 79 | 79 | 75 | 82 | 75 | 82 | 82 | 75 | 82 | 75 | 75 | 82 |
| | $y_5$ | 93.75 | 94 | 94 | 94 | 94 | 94 | 94 | **100** | 88 | **100** | 88 | 88 | **100** | **100** | 88 |
| | $y_6$ | **100.0** | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| $S_8$ | $y_2$ | 95.3125 | 96 | 95 | **100** | 91 | 99 | 93 | 97 | 94 | 93 | 99 | 94 | 97 | 96 | 96 |
| | $y_3$ | 96.875 | 97 | 97 | **100** | 94 | 97 | 97 | 94 | **100** | 97 | 97 | **100** | 94 | 97 | 97 |
| | $y_4$ | 93.75 | 94 | 94 | 94 | 94 | 94 | 94 | 88 | **100** | 94 | 94 | **100** | 88 | 94 | 94 |

Returning to the Table 6.6 it may be seen that the sub-circuit $y_6$ is fully functional, since $f^o_{\mathbf{y_6}} = 100$. Also, the system can be decomposed using Shannon's decomposition by any of the following variables: $x_3$, $x_4$, $x_5$ or $x_6$. In this case the partitioning vector can be randomly chosen among these variables, since the other metrics corresponding to these variables are 88 in all cases. For example, the system can be decomposed by variable $x_3$. In this case the sub-system generated according metric $f^{io}_{y_5}|_{x_3=0}$ is fully functional and another system is easily evolvable since the circuit $S_4$ evaluated according to metric $f^{io}_{y_5}|_{x_3=1}$ has 88% correct bits already. A similar analysis is carried out for the sub-system $S_8$.

As a result of EHW-oriented decomposition, the following sub-systems can be defined:

1. $S_0$: $T_0 = (I(7,128), O(10,128))$; $Y = \{y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9\}$. *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: Random. Best chromosome - $\mathbb{C}_0$. **Conclusion**: Decomposition by outputs: $S_1$ - easily evolved; Decomposition by input $x_1$: $S_2$, $S_3$ - easily evolved; $S_4$.

2. $S_1$: $T_1 = (I(7,128), O(3,128))$; $Y = \{y_7, y_8, y_9\}$. *Purpose of evolution:* reduce the number of active gates in circuit; *Initialisation* - $\mathbb{C}_0$ (fully functional circuit). Best chromosome - $\mathbb{C}_1$. **Conclusion**: a fully functional circuit with a reduced number of active gates is evolved.

3. $S_2$: $T_2 = (I(7,64), O(2,64))$; $Y = \{y_0, y_1\}$; $X = \{x_0, 0, x_2, x_3, x_4, x_5, x_6\}$; *Purpose of evolution:* reduce the number of active gates in circuit; *Initialisation* - $\mathbb{C}_0$ (fully functional circuit). Best chromosome - $\mathbb{C}_2$. **Conclusion**: a fully functional circuit with a reduced number of active gates is evolved.

4. $S_3$: $T_3 = (I(7,64), O(2,64))$; $Y = \{y_0, y_1\}$; $X = \{x_0, 1, x_2, x_3, x_4, x_5, x_6\}$. *Purpose of evolution:* evolve a fully functional circuit and reduce the number of active gates in circuit; *Initialisation* - $\mathbb{C}_0$ (fully functional circuit). Best chromosome - $\mathbb{C}_3$. **Conclusion**: a fully functional circuit is evolved after 15 000 generations.

5. $S_4$: $T_4 = (I(7,128), O(5,128))$; $Y = \{y_2, y_3, y_4, y_5, y_6\}$. *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: Random. Best chromosome - $\mathbb{C}_4$. **Conclusion**: Decomposition by outputs: $S_5$ - easily evolved; Decomposition by input $x_3$: $S_6$, $S_7$ - easily evolved; $S_8$.

6. $S_5$: $T_5 = (I(7,128), O(1,128))$; $Y = \{y_6\}$. *Purpose of evolution:* reduce the number of active gates in circuit; *Initialisation* - $\mathbb{C}_4$ (fully functional circuit).

Best chromosome - $\mathbb{C}_5$. **Conclusion**: a fully functional circuit with a reduced number of active gates is evolved.

7. $S_6$: $T_6 = (I(7,64), O(1,64))$; $Y = \{y_5\}$; $X = \{x_0, x_1, x_2, x_3, x_4, x_5, 0\}$. *Purpose of evolution:* reduce the number of active gates in circuit; *Initialisation* - $\mathbb{C}_4$ (fully functional circuit). Best chromosome - $\mathbb{C}_6$. **Conclusion**: a fully functional circuit with reduced number of active gates is evolved.

8. $S_7$: $T_3 = (I(7,64), O(1,64))$; $Y = \{y_5\}$; $X = \{x_0, x_1, x_2, x_3, x_4, x_5, 1\}$;. *Purpose of evolution:* evolve a fully functional circuit and reduce the number of active gates in circuit; *Initialisation* - $\mathbb{C}_4$ (fully functional circuit). Best chromosome - $\mathbb{C}_7$. **Conclusion**: a fully functional circuit is evolved.

9. $S_8$: $T_8 = (I(7,128), O(3,128))$; $Y = \{y_2, y_3, y_4\}$; $\mathcal{F} = 95.3125$; $f^o_{\mathbf{y_2}} = 95.3125$; $f^o_{\mathbf{y_3}} = 96.875$; $f^o_{\mathbf{y_4}} = 93.75$. *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: Random. Best chromosome - $\mathbb{C}_8$. **Conclusion**: Decomposition by inputs and outputs. Easily evolved: $S_9$, $S_{11}$; $S_{10}$, $S_{12}$.

10. $S_9$: $T_9 = (I(7,64), O(2,64))$; $Y = \{y_3, y_4\}$; $X = \{x_0, x_1, x_2, x_3, x_4, 0, x_6\}$. *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: $C_8$. Best chromosome - $\mathbb{C}_9$. **Conclusion**: a fully functional circuit with a reduced number of active gates is evolved.

11. $S_{10}$: $T_{10} = (I(7,64), O(2,64))$; $Y = \{y_3, y_4\}$; $X = \{x_0, x_1, x_2, x_3, x_4, 1, x_6\}$. *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: Random. Best chromosome - $\mathbb{C}_{10}$. **Conclusion**: a fully functional circuit with a reduced number of active gates is evolved.

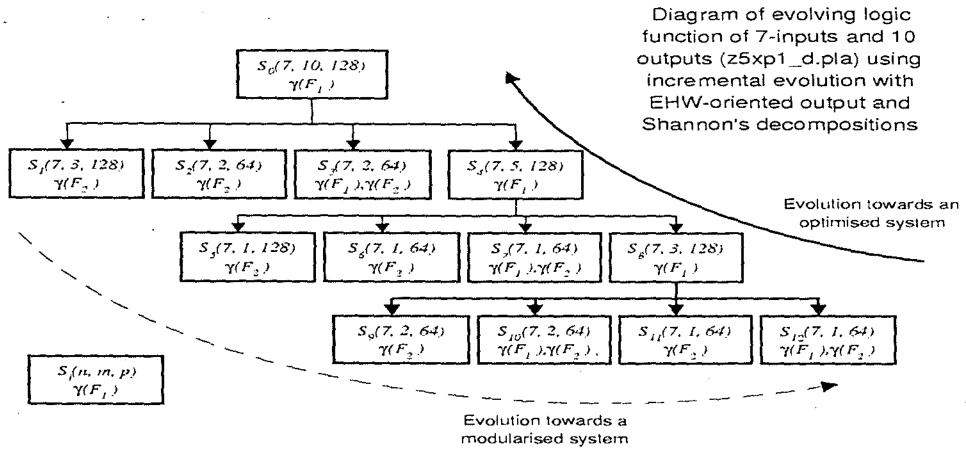


Diagram of evolving logic function of 7-inputs and 10 outputs (z5xp1_d.pla) using incremental evolution with EHW-oriented output and Shannon's decompositions
S0(7, 10, 128) γ(F1)
S1(7, 3, 128) γ(F2)
S2(7, 2, 64) γ(F2)
S3(7, 2, 64) γ(F1),γ(F2)
S4(7, 5, 128) γ(F1)
Evolution towards an optimised system
S5(7, 1, 128) γ(F2)
S6(7, 1, 64) γ(F2)
S7(7, 1, 64) γ(F1),γ(F2)
S8(7, 3, 128) γ(F1)
S9(7, 2, 64) γ(F2)
S10(7, 2, 64) γ(F1),γ(F2)
S11(7, 1, 64) γ(F2)
S12(7, 1, 64) γ(F1),γ(F2)
Si(n, m, p) γ(F1)
Evolution towards a modularised system

Figure 6.14: z5xp1_d.pla: Diagram of performing bidirectional incremental evolution with output and Shannon's decompositions (Experiment D).

12. $S_{11}$: $T_{11} = (I(7,64), O(1,64))$; $Y = \{y_2\}$; $X = \{x_0, 0, x_2, x_3, x_4, x_5, x_6\}$. *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: $C_8$. Best chromosome - $\mathbb{C}_{11}$. **Conclusion**: a fully functional circuit with a reduced number of active gates is evolved.

13. $S_{12}$: $T_{12} = (I(7,64), O(1,64))$; $Y = \{y_2\}$; $X = \{x_0, 1, x_2, x_3, x_4, x_5, x_6\}$. *Purpose of evolution*: evolving a fully functional circuit; *Initialisation*: Random. Best chromosome - $\mathbb{C}_{12}$. **Conclusion**: a fully functional circuit with a reduced number of active gates is evolved.

Fig. 6.14 shows bidirectional incremental evolution allowing both output and Shannon's decompositions. This diagram is different from the one illustrated in Fig. 6.12. 13 sub-systems have been generated in order to perform the first stage of bidirectional incremental evolution. The generated tasks are easier then the one discussed in Experiment A, since they evolve functions with fewer number of inputs and therefore a smaller number of input/output combinations. Another difference is that this process can be performed in parallel since 3 or more sub-systems can be evolved at the same time. Therefore, the evolution with output and Shannon's decompositions can be implemented using a parallel system, which will decrease the computation time drastically.

If the task is to generate fully functional circuit only. The complex system can be assembled after the first stage. Thus, the complex circuit structure defined after the first stage of evolution is given in Fig. 6.15. The circuit contains 139 primitive logic gates. Since the Shannon method of decomposition is employed, the size of final complex system before optimisation stage depends also on the number of multiplexers used. During first stage of our method the size of circuit has been reduced partially, since the sub-circuits are evolved using dynamic fitness functions. This means that if the number of generations is large enough to perform both synthesis and optimisation of circuit during evolutionary process, the size of sub-circuit can be reduced. For example, during evolution of sub-circuit $S_1$ the size of circuit has been reduced from 20 primitive logic gates to 9. This optimisation is performed by evolutionary process $\Upsilon_{\mathcal{F}_2}$. Similar reductions can be noticed in the evolution of other sub-circuits. The summary of this is given in Table 6.7. As can be seen, the size of sub-circuits has been reduced even during first stage of BIE.

Thus, as a result of using dynamic fitness function during evolution of sub-circuits within the first stage of our method the size of circuit has been reduced from 245 logic gates to 139. This is still very large circuit in comparison with the one generated in experiment C (85 logic gates). 110 000 generations are required to complete the first stage. In order to compute the total number of generations required to evolve sub-circuits $S_4$ and $S_8$ are taken into account.

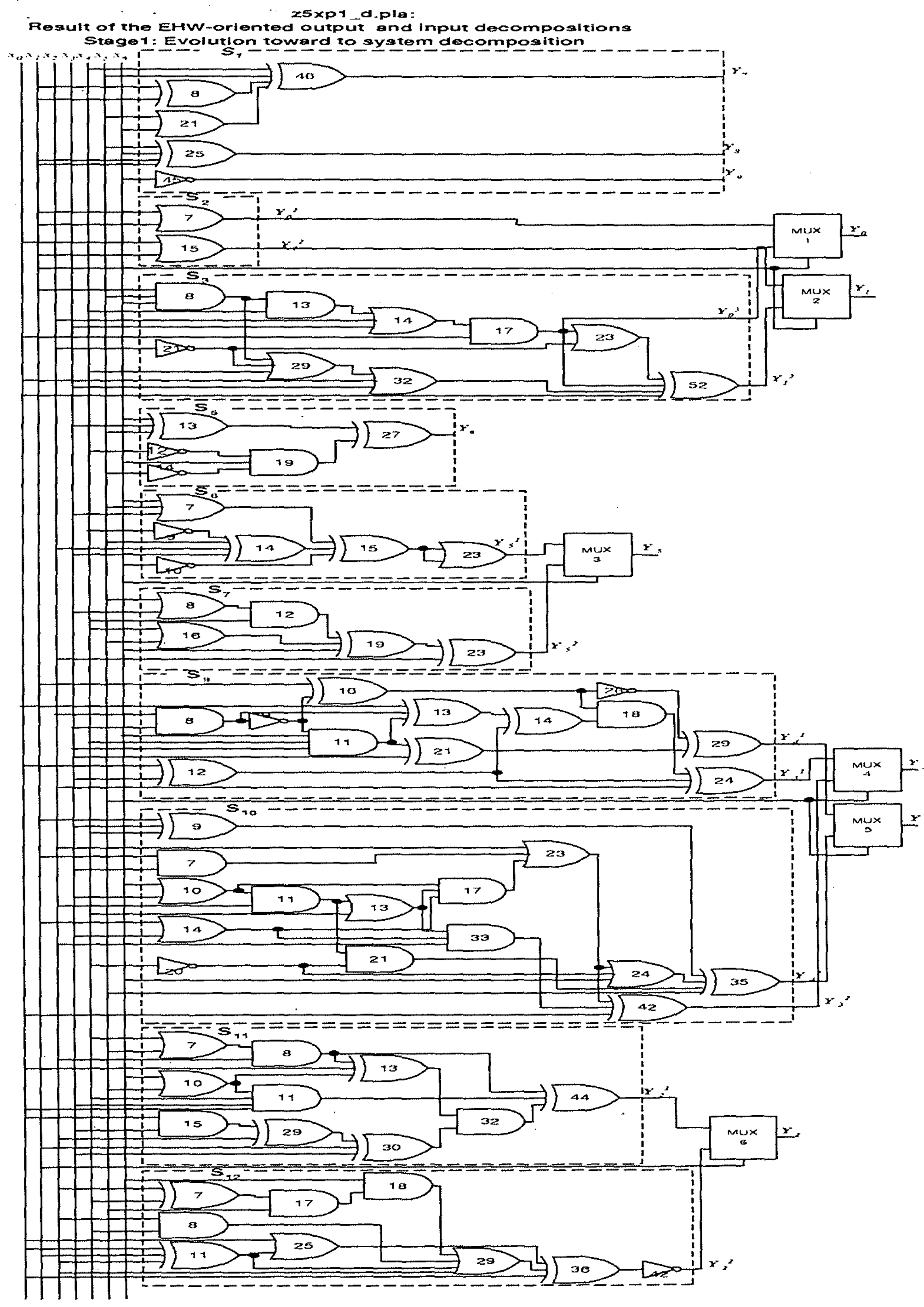

z5xp1_d.pla:
Result of the EHW-oriented output and input decompositions
Stage1: Evolution toward to system decomposition
S1
S2
S3
S5
S6
S7
S9
S10
S11
S12
MUX 1
MUX 2
MUX 3
MUX 4
MUX 5
MUX 6

Figure 6.15: z5xpl_d.pla: The EHW-oriented output and Shannon's decompositions (Stage 1: Evolution towards a modularised system) (Experiment D).

Table 6.7: z5xp1_d.pla: History of the bidirectional incremental evolution with the EHW-oriented output and input decomposition.

| **Sub-system** | | | | | | **ES performance** | | |
|---|---|---|---|---|---|---|---|---|
| $S_i$ | **Sub-circuit** | $n$ | $m$ | $p$ | $Y$ | $N_{gen}$ | $F_2(N_f)$ | $F_2(N_f^{opt})$ |
| **Stage 1: Functionality evolution** | | | | | | | | |
| $S_1$ | z5xp1_d_s1_do_3.pla | 7 | 3 | 128 | $\{y_7, y_8, y_9\}$ | 5 000 | 20 | 9 |
| $S_2$ | z5xp1_d_s2_do2_dpx1_0.pla | 7 | 2 | 64 | $\{y_0, y_1\}$ | 1 000 | 13 | 2 |
| $S_3$ | z5xp1_d_s3_do2_dpx1_1.pla | 7 | 2 | 64 | $\{y_0, y_1\}$ | 15 000 | 26 | 14 |
| $S_4$ | z5xp1_d_s4.pla | 7 | 5 | 128 | - | 5 000 | - | - |
| $S_5$ | z5xp1_d_s5_do1_dp.pla | 7 | 1 | 128 | $\{y_6\}$ | 5 000 | 9 | 7 |
| $S_6$ | z5xp1_d_s6_do1_dpx6_0.pla | 7 | 1 | 128 | $\{y_5\}$ | 5 000 | 14 | 10 |
| $S_7$ | z5xp1_d_s7_do1_dpx6_1.pla | 7 | 1 | 128 | $\{y_5\}$ | 5 000 | 10 | 7 |
| $S_8$ | z5xp1_d_s8.pla | 7 | 3 | 128 | - | 5 000 | - | - |
| $S_9$ | z5xp1_d_s9_do2_dpx5_0.pla | 7 | 2 | 64 | $\{y_3, y_4\}$ | 5 000 | 22 | 16 |
| $S_{10}$ | z5xp1_d_s10_do2_dpx5_1.pla | 7 | 2 | 64 | $\{y_3, y_4\}$ | 20 000 | 50 | 22 |
| $S_{11}$ | z5xp1_d_s11_do2_dpx1_0.pla | 7 | 1 | 64 | $\{y_2\}$ | 20 000 | 32 | 15 |
| $S_{12}$ | z5xp1_d_s12_do2_dpx1_1.pla | 7 | 1 | 64 | $\{y_2\}$ | 20 000 | 23 | 13 |
| **Total:** $\sum F_2(N_f^{opt})$ | | | | | | | 245 | 139 |
| **Total:** $\sum N_{gen} = 110000$ | | | | | | | | |
| **Stage 2: Circuit optimisation** | | | | | | | | |
| $OS_0$ | z5xp1_d_s2s3_do2_dpx1.pla | 7 | 2 | 128 | $\{y_0, y_1\}$ | 15 000 | 25 | 18 |
| $OS_1$ | z5xp1_d_s6s7_do1_dpx6.pla | 7 | 1 | 128 | $\{y_5\}$ | 15 000 | 21 | 12 |
| $OS_2$ | z5xp1_d_s9s10_do2_dpx5.pla | 7 | 2 | 128 | $\{y_3, y_4\}$ | 15 000 | 45 | 12 |

**Stage 2: Evolution towards an optimised system** The circuit optimisation has been carried out during 100 000 generations. A circuit of 60 logic gates has been synthesised. This circuit is larger than the one evolved in experiment C by four logic gates.

First, let us synthesise the connectivity diagram that reflects all connections between sub-circuits. This structure shown in Fig. 6.16 helps to define the assembling points during second stage of incremental evolution. For example, from this

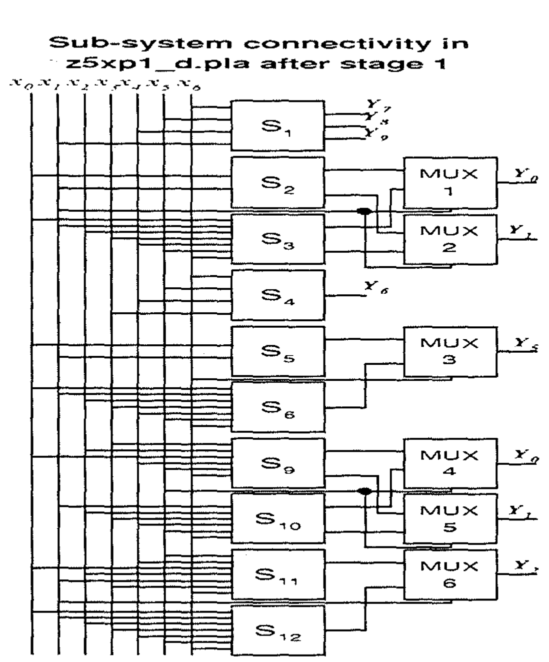


Figure 6.16: Sub-system connectivity in z5xp1_d.pla: The EHW-oriented output and Shannon's decompositions (Stage 1: Evolution towards system decomposition).

connectivity diagram clear, that in order to assemble the complex system, the sub-systems decomposed using Shannon's decomposition have to be merged. Thus, the sub-systems $S_2$ and $S_4$ can be assembled in more complex system. Similarly, the following three more complex sub-systems can be defined: $\{S_5, S_6\}$, $\{S_9, S_{10}\}$ and $\{S_{11}, S_{12}\}$.

So, in order to evolve the more complex system, the sub-systems have to be assembled in one chromosome. In this section the sub-systems are assembled according to the principles shown in Fig. 6.5. The multiplexers are represented as a collection of AND, OR, NOT logic gates instead of one building block. Once the more complex system is synthesised, the evolution is undertaken in order to optimise the system. For example, the size of sub-system assembled from the sub-circuits $S_2$ and $S_3$ has been reduced from 25 primitive logic gates to 18. The circuit structure of the

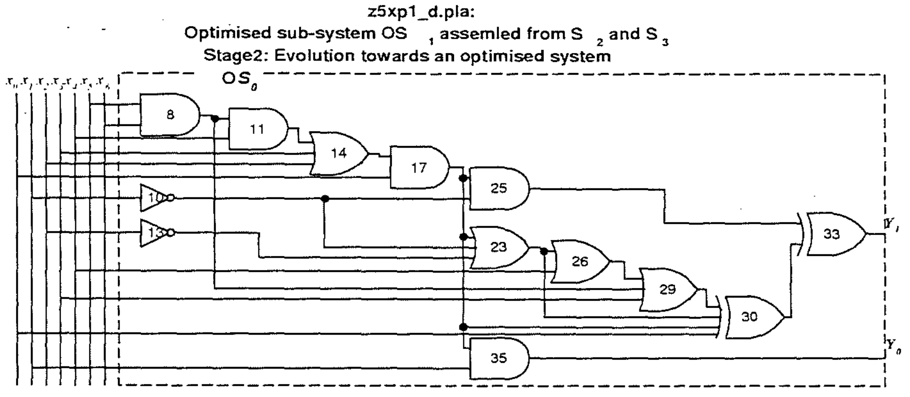

z5xp1_d.pla:
Optimised sub-system OS 1 assemled from S 2 and S 3
Stage2: Evolution towards an optimised system
OS0
8
11
14
17
25
23
26
29
30
33
35
Y1
Y0

Figure 6.17: z5xp1_d.pla: Optimised sub-system $OS_0 = \{S_2, S_3\}$. The number of primitive active logic gates is 18.

more complex system synthesised is given in Fig. 6.17. It may be seen, that after optimisation process the structure of circuit has been changed significantly. Thus, in this design it is difficult to define the main multiplexers, that have been presented in the initial circuit structure. This example shows that the circuit structure can be significantly changed if the multiplexers connected the sub-circuits are represented as a collection of primitive logic gates. Let us evolve more complex system, assembled from sub-circuits $S_6$ and $S_7$. Let us represent the multiplexer in chromosome genotype as it has been discussed in previous example. The circuit structure optimised in this circuit is given in Fig. 6.18. In this case the size of logic circuit has been reduced from 21 primitive logic gates to 12.

So, this example show that the complex circuit can be generated using BIE with output and Shannon's decompositions. The combination of these two decompositions allows us to deal with smaller circuits rather then in case of using output decomposition only. By this reason the evolutionary process performs quicker, since the smaller circuit layout is required to implement sub-circuits and smaller truth tables are used to evaluate circuits.

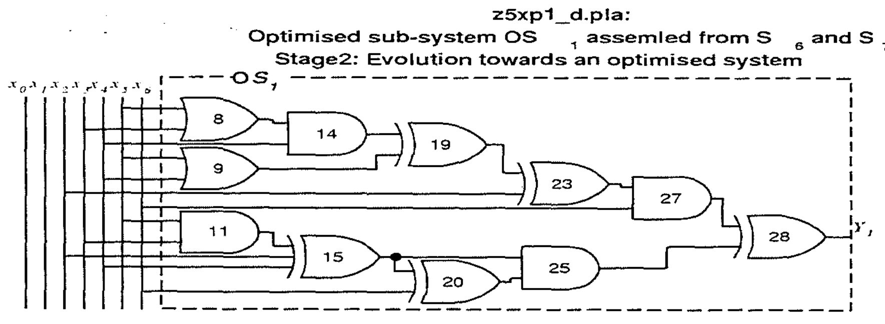

z5xp1_d.pla:
Optimised sub-system OS 1 assemled from S 6 and S 7
Stage2: Evolution towards an optimised system
x0 x1 x2 x3 x4 x5
OS1
8
9
11
14
15
19
20
23
25
27
28
Y1

Figure 6.18: z5xp1_d.pla: Optimised sub-system $OS_1 = \{S_6, S_7\}$. The number of primitive active logic gates is 12.

The analysis of evolved structures shows that representing the multiplexer by collection of logic gates (AND, NOT, OR) allows us to change the circuit structure drastically during optimisation phase.

## 6.5 Summary

This chapter shows that even when a task is too difficult to evolve directly, evolution can be applied incrementally in two directions to achieve the desired complex behaviour. In this approach, evolution performs in two directions: from complex system to sub-systems and from sub-systems to complex system. The experimental results show that bidirectional incremental evolution performs significantly better then direct evolution. Also, it has been demonstrated how the evolutionary process can be different when the different types of decomposition are allowed. The approach should be applicable to many real EHW applications, where often a natural hierarchy of behaviours from simple to complex exists.

The empirical investigation of circuit evolved using both BIE with standard decomposition methods and BIE with EHW-oriented decomposition method. In both case, it is possible to achieve the fully functional solution of complex system within a relatively small number of generations. In case of using EHW-oriented decomposition, it has been shown that the partitioning vectors can be defined according to the result of the evolution carried out during previous step.

It has been shown that more complex tasks can be evolved using bidirectional incremental evolution, since it is not restricted by the size of truth table or "stalling" effect of direct evolution.

The experiments on evolving digital logic functions constitute a starting point for research on methods that evolve complex general behavior in two directions and on methods that investigate the functional interconnectivity inside a given complex problem. There are many tasks that have to be solved in this direction. A major direction of future work will be to apply bidirectional incremental evolution to real world application tasks solved intrinsically and extrinsically.

It has been shown empirically that the proposed bidirectional incremental approach evolves more effective and more general circuits and should also scale up to harder tasks.

# Chapter 7

# Multi-valued logic circuit design

The basic idea of the work reported in this chapter is to show the universality of proposed method in terms of application tasks and their behaviour. The self-adaptive function-level EHW approach discussed in Chapter 4 and Chapter 5 is applied to the multiple-valued logic design. The behaviour of EHW approach for this application is discussed in detail. The empirical study proves the universality of developed methods. Evolvable hardware has been proposed as a new method for designing the systems in the complex applications. The basic idea of this approach is to evolve a fully functional circuit using an initially randomly generated one. In this chapter we report on the application of evolvable hardware techniques to multiple-valued circuit design. We choose a rectangular array structure to represent the internal connectivity of logic gates. Any set of multi-valued primitive operator or any multi-input multi-output logic function can be included in the functional set of logic gates. We consider both function- and gate-level extrinsic evolvable hardware methods that allow us to synthesise the $n$-input $m$-output $r$-valued logic functions. A number of experiments have been carried out in order to investigate the specific features of proposed method. It has been discovered that the algorithm performance depends on the circuit layout (the number of columns and rows in rectangular array together with connectivity restrictions) and the functional set of logic gates chosen to be used. The experimental results show that function-level evolvable hardware method performs better than the gate-level evolvable hardware in terms of the number of fully functional circuits evolved.

## 7.1 Introduction

*Evolvable Hardware* (EHW) refers to the generation of electronic circuits using evolutionary algorithms. A central idea of this is that each possible electronic circuit is represented as a chromosome in an evolutionary process in which the standard genetic operations over the circuits, such as initialisation, recombination, elitism, selection are carried out. The evolving circuits may be evaluated using software simulation models [108], [111], [97], [137], [138], [25], [12] (so called *extrinsic EHW*) or, alternatively, evolved entirely in hardware [139], [36], [140] (so called *intrinsic EHW*). The evolution of circuits using primitive logic gates performs at *gate-level EHW*. *Function-level EHW* employs multi-input multi-output logic functions to describe the behaviour of building blocks in the evolving circuit.

There are two main approaches for the synthesis of combinational logic circuits using evolutionary algorithms. The first optimises the formal representation of function and synthesises the circuit based on the optimal function representation. In this case a functionally complete basis is chosen and the genetic algorithm is applied to optimise the form of the function representation. Using the obtained optimised representation the circuit structure is synthesised. It is clear that the circuit design is obtained by the application of algebraic rules associated with the relevant algebra.

An overview of evolutionary methods using the design of quaternary digital circuits can be found in [141]. The design of multiple-valued logic (MVL) circuits based on cost-table techniques using evolutionary algorithms is discussed in [142], [143]. The second approach [108], [111], [1], [36] begins from randomly connected and randomly chosen gates and gradually evolves the target functionality. The particular set of gates used is fixed in advance, but whether or not any particular gate is used, or, how many times a gate is used, is entirely free. The advantage of this approach is that it allows us to synthesize the circuit using *any* set of logic gates. Consequently, it permits the synthesis of compact and unusual circuit structures. In this way we can abandon the restrictions associated with conventional design [9].

This chapter presents a method for the synthesis of combinational multiple-valued circuits. The approach is an extension of the evolvable hardware method for binary logic circuits proposed in [111], [25]. The first attempts to evolve multiple-valued arithmetical combinational circuits were discussed in [8]. A discussion concerning a suitable set of multiple-valued logic gates was given in [8] and some specific features of the circuit layout were reported in [7]. The dynamic fitness which allows us to improve the quality of evolved circuits in terms of the number of active gates used in evolution was proposed in [1] and Chapter 3. Efforts to evolve the circuit layout together with circuit functionality were discussed in [1]. Some circuit structures evolved for a 3-valued half–adder were examined in [9]. In this Chapter we consider notable features of gate- and function-levels extrinsic evolvable hardware approaches discussed in Chapter 3, Chapter 4 and Chapter 5. Thus, we examine the optimal rectangular dimensions, which allow the evolution of fully functional circuits for an $n$-input $m$-output $r$-valued logic function. We show experimentally that certain sets of multiple-valued gates allow much easier evolution of fully functional circuits. A number of evolved circuit designs for some arithmetic functions such as a half adder, an one-digit full adder, an one-digit multiplier, etc. are discussed. These designs were evolved using both gate- and function-level EHW. Some of the logic designs evolved cannot be proved to be equal to similar designs implementing the same logic function. A notable feature of this work is that it shows how it is possible to produce circuits which combine different functionally complete sets of multiple-valued gates within a multiple-valued circuit design and proves the universality of approach discussed in this dissertation.

## 7.2 Idea of EHW approach

The basic idea of the proposed approach is as follows. A combinational logic circuit is represented as a rectangular array of building blocks. A *building block* can implement any primitive logic operator or multi-input multi-output logic function behaviours, which is specified in advance, or can be calculated based on the interactive connectivity principles of logic gates. A *function-level extrinsic evolvable hardware approach* uses building blocks of any complexity. For example, a multi-input multi-output logic function can describe the behaviour of a building block. *Gate-level extrinsic evolvable hardware* is employed, when the building blocks are defined by primitive logic operators. In subsequent discussion we will refer to the function behaviour which is known of in advance as the *standard* logic function. A two-digit adder is an example of a standard logic function. In terms of building block implementation we assume that the optimal structure has been chosen. Therefore, the search space of the algorithm is the combinational functions computed by the building blocks and the interconnections among these building blocks. The *circuit layout* defined by the number of columns and rows in a rectangular array is fixed. The degree of connectivity in the circuit is called the *connectivity parameter* and is defined by the number of columns of building blocks to the left of the current column, which can have their outputs connected to the inputs of the current cell, and, by implying to the final circuit outputs. The experimental results show that it is easier to evolve a circuit using a suitable functional set of multi-input multi-output building blocks.

## 7.3 Background and notations

Let $R = \{0, 1, \cdots, r-1\}$ be the set of the $r$-valued logic values, and let $X = \{x_0, x_1, \cdots, x_{n-1}\}$ be an input vector of $n$ variables, where $x_k$ takes on values from $R$, $k \in \{0, ..., n-1\}$. The cardinality of $X$ defines the number of members of $X$ and denoted as $|X|$. Then, $|X| = n$. An $r$-valued $n$-variable logic function $f(X)$ is a mapping $f : R^n \longrightarrow R$. Then, an $r$-valued $n$-input $m$-output logic function $F(X)$ is defined by the output vector of $m$ logic functions $\{f_0, f_1, \cdots, f_{m-1}\}$ and is a mapping $f : R^n \longrightarrow R^m$. The primitive multi-valued two-variable operators are defined as follows:

1. *Disjunction operator*, MIN [144]: $x_1 \wedge x_2 = x_1 \cdot x_2 = min(x_1, x_2)$;
2. *Conjunction operator*, MAX [144]: $x_1 \vee x_2 = max(x_1, x_2)$;
3. *Modular sum operator*, MODSUM [145]: $x_1 \oplus x_2 = x_1 + x_2 \pmod r$;
4. *Modular product operator*, MODPRODUCT [145]: $x_1 \otimes x_2 = x_1 * x_2 \pmod r$;
5. *Truncated sum operator*, TSUM [146]: $x_1 \Diamond x_2 = min(x_1 + x_2, r-1)$;

6. *Truncated product operator*, TPRODUCT [147], [148]: $x_1 \star x_2 = max(x_1 + x_2 - (r-1), 0)$;

7. *Operator of Sheffer* [149]:

$$x_1 | x_2 = \begin{cases} (1+x_1) \pmod{r}, & x_1 = x_2 \\ 0, & x_1 \neq x_2 \end{cases}$$

8. *Sum operator over GF(r)* [150];

9. *Product operator over GF (r)* [150];

10. *Truncated difference operator* [151]:

$$x_1 - x_2 = \begin{cases} (x_1 - x_2), & x_1 >= x_2 \\ 0, & otherwise. \end{cases}$$

where "+" is an ordinary addition. Note, that TPRODUCT operator has been introduced as two-variable logic operator. The notation of this operator given above can be easily extended to the $r$-valued $n$-variable logic operator as follows:

$$x_0 \star x_1 \star \dots \star x_{n-1} = max(x_0 + x_1 + \dots + x_{n-1} - (n-1)(r-1), 0).$$

The most commonly encountered unary operators are considered below.

1. *Complement of a logic level $l$* called also *negation*, NOT [152]: $!x_1 = \overline{x_1} = (r-1) - x_1$;

2. *Clockwise cycle operator* [153]: $x_1^{\rightarrow k} = x + k \pmod{r}$. When $k = 1$, this is a single-variable addition operator called also SUCCESSOR [154]: $x_1^{\rightarrow 1} = ?x_1 = x_1 + 1 \pmod{r}$;

3. *Counter clockwise cycle operator* [153]: $x_1^{\leftarrow k} = x - k \pmod r$;

4. *Literal of a x variable* [144]:

$$x_s = \begin{cases} (r-1), & x = s \\ 0, & x \neq s; \end{cases} \qquad s \in \{0, 1, \cdots, r-1\}.$$

Some of functionally complete sets of MVL operators represented any given logic function are given below:

| | | |
|---|---|---|
| $\{x^{\rightarrow 1}$, MAX$\}$ | 1921 | Post [154] |
| {MODSUM, MODPRODUCT} | 1924 | Bernstein [145] |
| {operator of Sheffer} | 1935 | Webb [149] |
| $\{x^0, x^1, \cdots, x^{r-1}$, MIN, MAX$\}$ | 1968 | Allen and Givone [144] |
| $\{x^{\rightarrow k}$, MIN, MAX$\}$ | 1970 | Vranesic et al. [153] |
| $\{\overline{x_1}, MIN, MAX\}$ | 1970 | Vranesic et al. [153] |
| {+, x (over GF($r$))} | 1984 | Hurst [150] |
| $\{x^0, x^1, \cdots, x^{r-1}$, MIN, MODSUM$\}$ | 1986 | Dueck and Miller [155] |
| $\{x^0, x^1, \cdots, x^{r-1}$, MIN, TSUM$\}$ | 1988 | Onneweer et al. [146] |
| $\{x^0, x^1, \cdots, x^{r-1}$, MIN, TSUM$\}$ | 1991 | Pelayo et al. [156] |

The multi-valued operators have been implemented using different technologies:

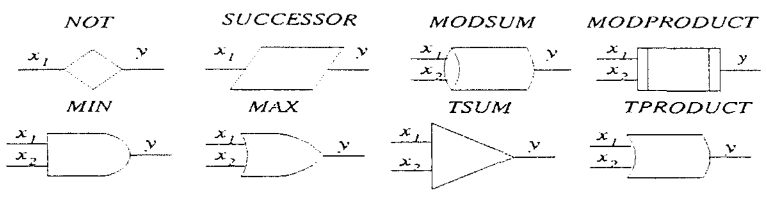


Figure 7.1: Symbols of the two-input $r$-valued logic gates

| | | | |
|---|---|---|---|
| NOT(MIN($x_1$, $x_2$)) | 1977 | CMOS | Vranesic, Smith [157] |
| + GF(4) | 1978 | $I^2L$ | Pugsley, Silio [158] |
| + GF(4) | 1981 | $T^2L$ | Davio, Deschamps [151] |
| TDifference | 1981 | $I^2L$ | Davio, Deschamps [151] |
| $(x+1)$ mod 3,4 | 1984 | LSI | Hurst [150] |
| TSUM, MIN, MAX | 1986 | CCD | Abd-El-Barr, Vranesic [159] |
| $x^s$ | 1989 | CMOS | Sasao [160] |
| $x^s$, TSUM | 1991 | CMOS | Sasao [160] |
| $x^{\rightarrow k}, x^{\leftarrow k}$, MIN | 1993 | CMOS | Jain, Bolton, Abd-El-Barr [161], [162] |
| MAX, TSUM, $x^s$ | 1993 | CMOS | Jain, Bolton, Abd-El-Barr [161], [162] |
| +, × GF (4) | 1993 | CMOS | Zilic, Vranesic [163] |
| +, × GF (4) | 1994 | CMOS | Pierzchala, Perkowski, Grydel [164] |
| MIN, TSUM, $x^s$ | 1994 | RTT | Deng et al. [165] |
| $x^{\rightarrow k}, x^{\leftarrow k}$ | 1996 | Current Mode | Abd-El-Barr, Hasan [166] |
| TDifference | | CMOS | |

In this chapter we do not restrict ourselves to a specific functionally complete set of multiple-valued operators. We can combine different functionally complete bases. The symbolic representation of MVL operators used in this paper is shown in Fig. 7.1.

The logic gates implementing the primitive logic functions mentioned above are used as a building blocks for the synthesis of complete MVL circuits at the gate-level EHW.

## 7.4 An extrinsic EHW

In order to synthesize logic circuits a rudimentary $(1+\lambda)$ evolutionary strategy (ES) with uniform mutation was implemented. During evolution, the circuit layout, as well as the input functionality gene, was not allowed to be changed. Thus, only logic gates with primary inputs were used. During initialisation, the initial genotypes of chromosomes were generated randomly.

### 7.4.1 Encoding

In order to define the chromosome representation the following notations have been adopted:

| | |
|---|---|
| $N_{cols}^{max}, N_{rows}^{max}$ | the maximum number of columns and rows in rectangular array respectively; |
| $N_{cols}, N_{rows}$ | the number of columns and rows in rectangular array, respectively, $N_{cols} \in \{1, \cdots, N_{cols}^{max}\}$ and $N_{rows} \in \{1, \cdots, N_{rows}^{max}\}$; |
| $N_{connect}$ | the *connectivity parameter* representing the number of columns on the left to which a cell in a particular column $c_{col}$ or an output may be connected and $N_{connect} \in \{1, \cdots, N_{cols}\}$; |
| $N_{in}^{max}, N_{out}^{max}$ | the maximum number of inputs and outputs in any building block respectively; |
| $N_{in}(\mathcal{B}), N_{out}(\mathcal{B})$ | the number of inputs and outputs in the building block $\mathcal{B}$ respectively; |

Table 7.1: Gate functionality according to the $b_0(z)$ gene in chromosome

| Gene functionality, $b_0(z)$ | Gate function |
|---|---|
| 0 | Logic constant |
| 1 | $SUCCESSOR$ :$?x_0$ |
| 2 | $NOT$ :$!x_0$ |
| 3 | Literal: $L(x_0, c)$ |
| 4 | Clockwise Operator: $c \leftarrow x_0$ |
| 5 | Counter Clockwise Operator: $c \rightarrow x_0$ |
| 6 | Wire: $x_0$ |
| 7 | $MIN : x_0 \cdot x_1$ |
| 8 | $MAX : x_0 \vee x_1$ |
| 9 | $MODSUM : x_0 \oplus x_1$ |
| 10 | $MODPRODUCT : x_0 \otimes x_1$ |
| 11 | $TSUM : x_0 \diamond x_1$ |
| 12 | $TPRODUCT : x_0 * x_1$ |
| 13 | $x_0 + x_1$ (over $GF(r)$ |
| 14 | $x_0 * x_1$ (over $GF(r)$) |
| 15 | Multiplexer |
| 16 | 1-digit multiplier |
| 17 | 1-digit full adder |
| 18 | 2-digit multiplier |
| 19 | 2-digit full adder |
| 20 | 3-digit multiplier |
| 21 | 3-digit full adder |
| 22 | Half adder |

**Functional set.**

The functional set of all possible logic functions that can be used in evolution is shown in Table 7.1. Let $T_f^{all}$ be the set of integers defining the codes of logic functions reported in Table 7.1. $|T_f^{all}|$ defines the maximum number of logic functions allowed to be used in evolution. We can assign any $n$-input $m$-output logic function to describe the behaviour of a building block. In given case we choose arithmetic functions such as a half adder, a full adder and a multiplier. Any of logic functions mentioned in Table 7.1 can be involved in evolution. The set of logic functions actually used in evolution and encoded as shown in Table 7.1, is defined as follows: $T_f = \{t_f : t_f \in T_f^{all},\ \text{some } t_f\}$. $|T_f|$ determines the number of primitive and standard logic functions involved in evolution.

We specify subsets of logic functions that influence differently the number of inputs and outputs in building blocks as:

1. the subset of 1-input 1-output functions (the unary operators), $T_f^1$;
2. the subset of 2-input 1-output functions (the primary operators), $T_f^2$;
3. the subset of multi-input multi-output *standard* functions, such as half adder, full adder, multiplier, $T_f^3$.

Based on the notation given above, we can summarise:

$$T_f = T_f^1 \cup T_f^2 \cup T_f^3, \quad T_f^1 \cap T_f^2 \cap T_f^3 = \emptyset. \tag{7.4.1}$$

The behaviour of a building block can be represented by any of the logic functions mentioned above or by the set of 2-input 1-output functions connected as shown in Fig. 7.2 (b). In this case a building block has $k$-inputs and $(k-1)$-outputs. Let us consider a building block $\mathcal{B}$ that can implement any logic function from subsets $T_f^1, T_f^2$ and $T_f^3$. Thus if $\mathcal{B}$ implements the logic function from subset $T_f^1$, then $N_{in}(\mathcal{B}) = 1$, $N_{out}(\mathcal{B}) = 1$. In the case of using a 2-input, 1-output logic function ($T_f^2$), the number of inputs can be variable and can be defined as follows: $N_{in}(\mathcal{B}) \in T_{in} = \{2, \cdots, N_{in}^{max}\}$, $N_{out}(\mathcal{B}) \in T_{out} = \{1, \cdots, N_{in}(\mathcal{B}) - 1\}$. For example, if $\diamondsuit$ defines the 2-input 1-output logic primitive function and $N_{in} = 4$, then $\{i_0, i_1, i_2, i_3\}$ and $\{o_0, o_1, o_2\}$ is the set of inputs and outputs in the building block from $T_f^2$, respectively. The number of outputs in the building block is $N_{out} = N_{in} - 1 = 3$. The primitive logic gates in the building block are connected as illustrated in Fig. 7.2 (b). These outputs can be analytically represented as follows:

$$\begin{aligned} o_0 &= i_0 \diamondsuit i_1; \\ o_1 &= (i_0 \diamondsuit i_1) \diamondsuit i_2; \\ o_2 &= ((i_0 \diamondsuit i_1) \diamondsuit i_2) \diamondsuit i_3. \end{aligned}$$

The number of inputs and outputs are fixed. No changes in the number of inputs and outputs are allowed, if the standard logic function defined in $T_f^3$ describes the building block behaviour.

So, we can define the relation between the type of logic function chosen to describe the behaviour of building block $\mathcal{B}$, the number of inputs and outputs in $\mathcal{B}$ as follows:

$$\begin{aligned} \{(t_f, t_{in}, t_{out}) : \quad & (t_f \in T_f^1 \wedge t_{in} \in \{1\} \wedge t_{out} \in \{1\}) \vee \\ & (t_f \in T_f^2 \wedge t_{in} \in T_{in} \wedge t_{out} \in T_{out}, \text{ some } t_{in}, \text{ some } t_{out}) \vee \\ & (t_f \in T_f^3 \wedge T_f^3 \rightarrow T_{in} \wedge T_f^3 \rightarrow T_{out})\}. \end{aligned} \qquad (7.4.2)$$

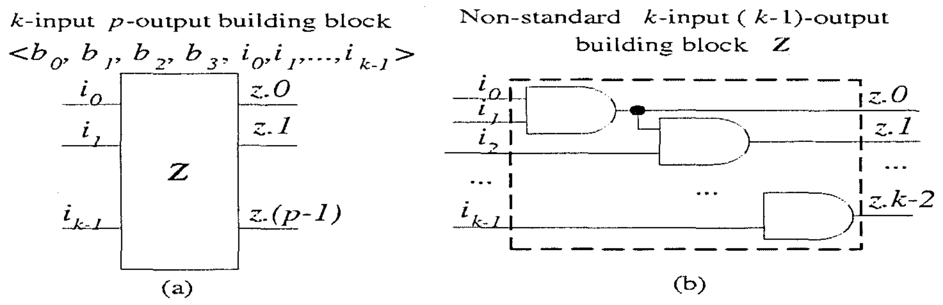

k-input p-output building block
<b0, b1, b2, b3, i0, i1, ..., ik-1>
i0
i1
ik-1
Z
z.0
z.1
z.(p-1)
(a)
Non-standard k-input (k-1)-output building block Z
i0
i1
i2
...
ik-1
z.0
z.1
...
z.k-2
(b)

Figure 7.2: Building block level representation

**Building block level representation.**

Let us consider the building block $\mathcal{B}_z$ labeled as $z$. Let $T_i$ be the set of integers $\{0, 1, \cdots, 2^{N_{in}^{max}}\}$. Let $N_{out}(\mathcal{B}_z)$ be the number of outputs in the building block and $N_{out}^{max}$ is defined to be no more than 10. Denote $V(c_{col})$ as the set of real numbers $v$ such that:

$$\begin{aligned} c_{col} > N_{connect}: \quad & a_{min} = n + (c_{col} - N_{connect}) * N_{rows}; \\ & a_{max} = n + c_{col} * N_{rows} - 1 \\ & a_{min} \leq v \leq a_{max} + 0.1 * N_{out}(\mathcal{B}_{a_{max}}) \\ c_{col} \leq N_{connect}: \quad & a_{min} = 0; \;\; a_{max} = n + c_{col} * N_{rows} - 1; \\ & a_{min} \leq v \leq a_{max} + 0.1 * N_{out}(\mathcal{B}_{a_{max}}) \end{aligned}$$

Any building block $\mathcal{B}_z$ located in the column $c_{col}$ and the row $c_{row}$ can be represented as follows:

$$\mathcal{B}_z = < b_0 \; b_1 \; b_2 \; b_3 \; \mathbb{I} >,$$

where $b_0 \in T_f$ is the *building block functionality gene*, which defines the type of building block $\mathcal{B}_z$; $b_1 \in T_i$ is the *input functionality gene* that determines the type

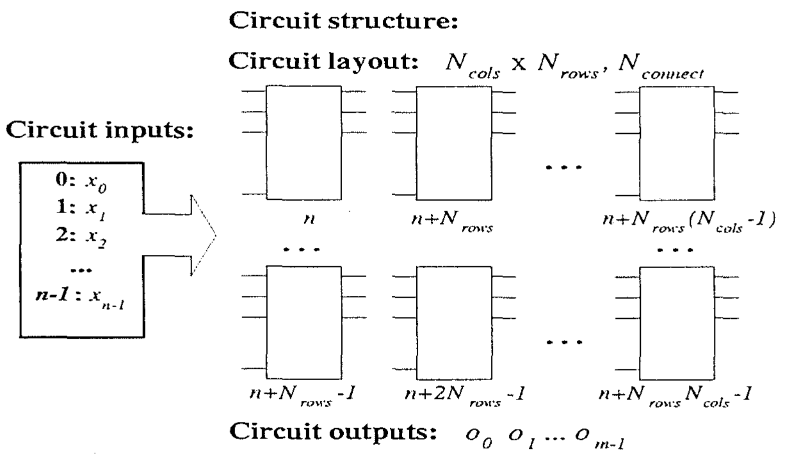


Figure 7.3: Circuit level representation

of inputs of building block $\mathcal{B}_z$ (primary or inverted); $b_2$ and $b_3$ are the number of outputs and inputs of the building block; the set $\mathbb{I} = \{i_0, i_1, \cdots, i_{N_{in}^{max}-1}\}$ defines the connections between building blocks, $\mathbb{I} = \{i \in V(c_{col}),\ \text{some } i\}$. The genes $b_0, b_2$ and $b_3$ are calculated according to (Eq. 7.4.2). Thus, relation $(b_o\ b_3\ b_2)$ is determined in (Eq. 7.4.2). The gene $b_1$, an integer, defines the type of inputs used in the building block. The lowest bit corresponds to the input $i_0$ and the highest bit corresponds to the input $i_{N_{in}^{max}}$. If the corresponding bit is zero the input is primary, otherwise the input is inverted. For example, let $b_1$ be 11. That corresponds to the binary sequence $< 1011 >$. This sequence can be encoded as $< i_3\ i_2\ i_1\ i_0 >$. The inputs $i_0, i_1$ and $i_3$ are considered as primary and the input $i_2$ is inverted.

**Circuit level representation.**

Let us consider the rectangular array $\mathbb{B}$ of the logic building blocks $\{\mathcal{B}(c_{col}c_{row}) : \{\mathcal{B}(c_{col}c_{row}) \in \mathbb{B},\ c_{col} = \{0, \cdots, N_{cols}-1\},\ c_{row} = \{0, \cdots, N_{rows}-1\}\}\}$. The building block $\mathcal{B}(c_{col}c_{row})$, located in the column $c_{col}$ and the row $c_{row}$, is labeled by an integer $(n + N_{rows} * c_{cols} + c_{rows})$. For example, if a 4-input logic function is evolved using

3x4 circuit layout, the building block $\mathcal{B}(00)$ located in the 0-th column and the 0-th row is labeled as 4, $\mathcal{B}_4$. The building block $\mathcal{B}(23)$ located in the 2-nd column and the 3-rd row is labeled 15, $\mathcal{B}_{15}$.

Let us consider how the connectivity parameter influences the circuit structure. For the first column of building blocks in the chosen geometry, the inputs may only be connected to the actual circuit inputs. However, provided that the connectivity parameter $N_{connect}$ is greater then 1, the building blocks from the second column can be connected to the outputs of building block from the first column as well as to the circuit inputs. If $N_{connect}$ is 1, then the building blocks from the second column can be only connected to the outputs of building blocks from the first column. In the case when $N_{connect} = N_{cols}$, the building blocks can be connected to any outputs of building blocks located to the left or to the circuit inputs. Decreasing $N_{connect}$ has the effect of reducing the number of possible circuit solutions that may be found.

Let $\mathbb{O}$ be the set of integers such that $\{o : o \in V(N_{cols})\}$, $|\mathbb{O}| = m$. The set $\mathbb{O}$ defines the circuit outputs. Therefore, the circuit genotype can be represented as follows:

$$C =< N_{cols} \ N_{rows} \ N_{connect} \ \mathbb{B} \ \mathbb{O} > . \tag{7.4.3}$$

**Genotype.**

The value $g(x)$ at position $x$ (measured from the left and starting at 0) is chosen as follows:

**Circuit layout**

| | |
|---|---|
| $x = 0$ | $g(0) = N_{cols} \in \{1, \cdots, N_{cols}^{max}\}$ |
| $x = 1$ | $g(1) = N_{rows} \in \{1, \cdots, N_{rows}^{max}\}$ |
| $x = 2$ | $g(2) = N_{connect} \in \{1, \cdots, N_{cols}\}$ |

**Building block**

$$0 \leq x < |\mathbb{B}| * (4 + |\mathbb{I}|) \quad (g(x)\ \ g(x+2)\ \ g(x+3)),\ \text{if}\ (x-3)\ \mathrm{mod}\ (4+|\mathbb{I}|) = 0;$$

$$g(x) \in T_i,\ \text{if}\ (x-3)\ \mathrm{mod}\ (4+|\mathbb{I}|) = 1;$$

$$g(x) \in V(c_{col}),\ \text{if}\ (x-3)\ \mathrm{mod}\ (4+|\mathbb{I}|) = 4, \cdots, 3+|\mathbb{I}|;$$

**Circuit outputs**

$$|\mathbb{B}| * (4 + |\mathbb{I}|) \leq x \qquad g(x) \in V(N_{cols})$$

**An example.**

An example of the chromosome representation with the actual circuit structure is given in Fig. 7.4. This circuit implements a 3-valued 1-digit full adder. This function has 3 inputs and 2 outputs and is implemented here on a combinational network with a 4x2 circuit layout. The circuit inputs are labeled as follows: 0, 1, and 2 that correspond to the input variables $x_0$, $x_1$ and $x_2$ respectively. Each output of the building block is labeled with an individual address. Thus, the output of an one-output building block located in the 0-th column and the 0-th row is labeled as 3.0. The first output of a 2-output building block located in the 3-rd column and the 0-th

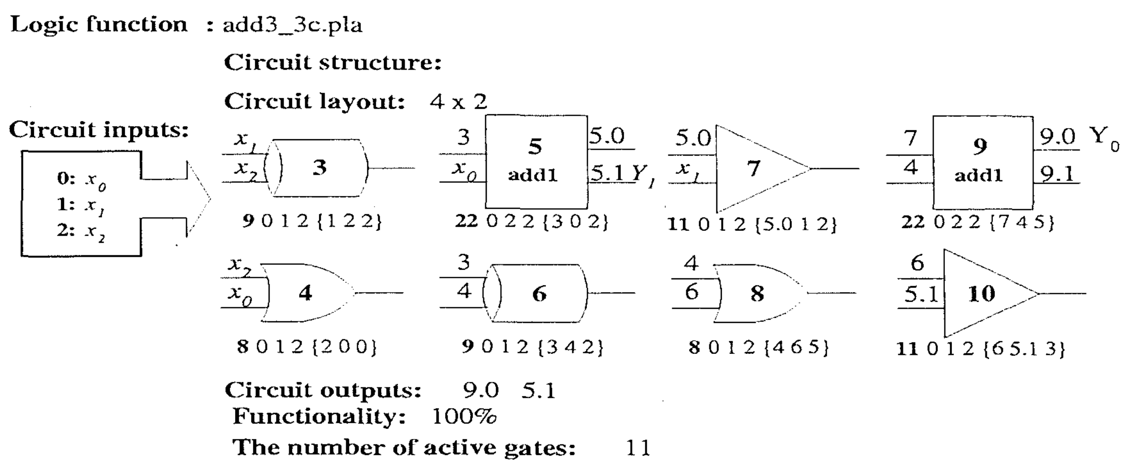


Figure 7.4: An example of the phenotype and corresponding genotype of a chromosome with 4x2 circuit layout

row is labeled as 9.0 and the second output of this block is encoded as 9.1. Thus, the main part of individual address corresponds to the address of a building block and the fraction part defines which output of the building block is utilised. For instance, let us consider a building block labeled 9, $\mathcal{B}_9$ with genotype $< 22\ \ 0\ \ 2\ \ 2\ \mid\ 7\ \ 4\ \ 5 >$.. The functionality gene of $\mathcal{B}_9$ is 22, that corresponds to the half adder according to Table 7.1. The second gene defines the input functionality and equals 0. This means that all inputs in $\mathcal{B}_9$ are primary. The two following genes determine the number of outputs and inputs in the building block. Because the number of outputs is 2, the outputs of $\mathcal{B}_9$ are encoded as 9.0 and 9.1. The number of inputs is also 2, which means that are used only 2 first connectivity genes connected to the building block outputs 7 and 4. The third connection is defined because the maximum number of inputs in the building block has been set up to 3. The circuit outputs are connected to the connections 9.0 and 5.1.

### 7.4.2 Fitness Function

The circuits are evaluated using a *dynamic fitness function.* In this case the evaluation is performed in two stages. First we are trying to find the fully functional circuit ($\mathcal{F}_1$) and, second, we are trying to minimise the number of active gates in the functionally complete circuit ($\mathcal{F}_2$). Thus, $\mathcal{F}_1$ rewards the circuits which have the correct digits in the correct positions for the circuit outputs. $\mathcal{F}_2$ adds a reward for the 100% functional circuits with the minimal number of active building blocks.

To present these fitness functions more formally, we need some definitions. Let function $F(X)$ be represented as a MVL matrix mapping denoted as $X \longrightarrow Y$, where $X$ is a $(r^n \times n)$ matrix of all $n$-variable inputs, $r^n$ is the number of input combinations, and $Y$ is a $(r^n \times m)$ matrix of the corresponding $m$ outputs. Then, the synthesis of MVL functions can be stated as follows. Design a sequence of operations that accomplishes the mapping $X \longrightarrow Y$. This mapping is achieved by applying a sequence of primitive operations. In our case the sequence of primitive operations is defined in the rectangular array of building blocks. Let $\mathcal{N}(\mathbf{x})$ be the output of an $r$-valued network $\mathcal{N}$ on the input combination $\mathbf{x}$, where $\mathbf{x}$ is the $n$-digit $r$-valued vector whose individual digits are the inputs to $N$. Suppose that $\mathcal{C}$ is a correct circuit, so that $\mathcal{C}(\mathbf{x}) = m * r^n$, since $m * r^n$ is the number needed to have the digits in $\mathbf{x}$ correctly sorted for an output combination with $m$ digits.

Our first fitness, $\mathcal{F}_1$ returns the number of correctly sorted digits over all inputs in $X$. Let $\Delta(\mathbf{x}, \mathbf{y})$ be the number of digits in $\mathbf{x}$ and $\mathbf{y}$ which agree with each other (the "opposite" of Hamming distance between $\mathbf{x}$ and $\mathbf{y}$), where $\mathbf{x}$ and $\mathbf{y}$ are $r$-valued vectors of $n$ elements. For example, $\Delta((2300), (2010)) = 2$ since the high order digits are both 2 and the low order digits are 0, but the other digits differ. Then, we can formally define our first fitness function as

$$\mathcal{F}_1(\mathcal{N}) = \frac{\sum_{\mathrm{y}_1 \in Y} \sum_{\mathrm{x} \in X} \Delta(\mathcal{C}(\mathrm{x}), \mathcal{N}(\mathrm{x}))}{m * r^n} * 100. \tag{7.4.4}$$

This definition implies that, if $\mathcal{F}_1(\mathcal{N}) = 100\%$, the circuit evolved is correct, i.e. the evolved network $\mathcal{N}$ is fully functional.

The second fitness, $\mathcal{F}_2$ defines the number of primitive logic cells unused in the circuit. Let $cost(\mathcal{N}(\mathcal{B}))^{max}$ be the maximum number of primitive cells needed to implement the most complex multi-input multi-output building block. Let $cost(\mathcal{N}(\mathcal{B}_z))$ be the number of primitive logic operators needed to implement the building block $\mathcal{B}_z$ and $u$ define the employment of $\mathcal{B}_z$. If $u = 1$, $\mathcal{B}_z$ is actually used in the evolved circuit and if $u = 0$, the building block is uncommitted. The fitness $\mathcal{F}_2$ can be calculated as follows:

$$\mathcal{F}_2 = \sum_{z=0}^{N_{cols} * N_{rows} - 1} cost(\mathcal{N}(\mathcal{B}))^{max} - u * cost(\mathcal{N}(\mathcal{B}_{\ddagger})). \tag{7.4.5}$$

Note, that fitness $\mathcal{F}_2$ is activated when $\mathcal{F}_1 = 100\%$. The dynamic fitness function is calculated as follows:

$$\mathcal{F} = \begin{cases} \mathcal{F}_1, & \mathcal{F}_1 < 100; \\ \mathcal{F}_1 + \mathcal{F}_2, & \mathcal{F}_1 = 100. \end{cases}$$

For example, the fitness function of the circuit, shown in Fig. 7.4, can be defined as follows. This implements a full one-digit adder completely, thus $\mathcal{F}_1 = 100\%$ and we have to compute the number of primitive active logic cell. Analysing the connectivity of the basic blocks it can be found that building blocks $\mathcal{B}_6$, $\mathcal{B}_8$ and $\mathcal{B}_{10}$ are uncommitted. Thus, there are 5 building blocks actually involved in the circuit implementation. Note, that $\mathcal{B}_5$ and $\mathcal{B}_9$ implement the half adder requiring at least 4 primitive logic cells to be implemented. Therefore the number of active primitive logic cells in the circuit is 11. The number of active building blocks is 5. The maximum number of logic cells required to implement the most complex building block is 4, thus $\mathcal{F}_2 = 21$ and $\mathcal{F} = \mathcal{F}_1 + \mathcal{F}_2 = 121$.

## 7.5 Evolved circuit designs for arithmetic circuits

In this section we will consider some experimental results obtained for the half and the full adders and the one-digit multiplier.

### 7.5.1 A half adder, add3_2.pla

The circuit structures depicted in Fig. 7.5 and implemented the 3-valued half adder have been evolved using an extrinsic gate-level EHW. $!x_i$ corresponds to inverted input $\overline{x_i}$. In this series of experiments, inverted inputs are allowed to be considered as circuit inputs as well as primary ones. Analysis of these structures shows that the algorithm adapts to the specific features of logic operators. We chose these circuits to analyse because all of them have the same circuit structure. Four logic gates are involved to implement the 3-valued half adder. In all cases the sum digit is synthesized using the MODSUM logic operator. The implementation of the carry function contains 4 logic gates. The circuit connectivity is the same for all implementations. The difference between the circuits is defined by employing different logic operators to the logic gates 3 and 4 as well as using different input combinations to the logic gate 3. This example shows that, in some cases, algorithm can implement logic function with the same circuit connectivity, but using different logic operators.

Comparing circuit structures, shown in Fig. 7.5 (c) and (d), we can see that they are identical except for the final, fourth gate. In the first case this is the MAX operator and in the second case this is the MIN operator. It is well known that the

TPRODUCT and MIN operators for the 3-valued logic are not identical. Therefore the circuits do not implement the same function from the logic algebra point of view. There are no proofs that the TPRODUCT operator can be used instead of MIN. However in this specific case the EA finds that these operators are "identical", i.e. they can replace each other. Careful investigation of the truth tables of these operators shows that the difference between them is in a single input combination: $1 \wedge 1 = 1$ (the MIN operator) and $1 \star 1 = 0$ (the TPRODUCT operator). Hence, we can conclude that the TPRODUCT and MIN operators are identical except for input combination $x_0 = 1$, $x_1 = 1$. The EA generates input combinations of these gates such that the input combination $x_0 = 1$, $x_1 = 1$ is not used. This type of exploitation of gate function is highly non-intuitive.

The circuit structures shown in Fig. 7.5 (c) and (b) are identical except gate 3. Logic gate 3, given in Fig. 7.5 (b), can be simplifies to the TPRODUCT gate with primary inputs using de-Morgan's law. It is clear from logic algebra (the definitions of these operators) that the MIN and TPRODUCT operators are not identical. The difference in the truth tables of these operators is compensated in the 4-th gate. In other words, when the output combination for $x_0 = 1$ and $x_1 = 1$ do not depend on gate 3 we can ignore the difference in the logic operators. The output for the carry digit will always be 0, because the output of gate 2 is always 0 for $x_0 = 1$ and $x_1 = 1$ and gate 4 represents the MIN operation and, in any case, will give priority for this value.

Let us consider the case mentioned above from the algebraic point of view. Let $A$ and $B$ be the logic variables or functions. Symbols $*$ and $+$ define the logic operations, the truth tables of which are not identical. Then, we can conclude from the above

Table 7.2: Truth tables for the circuits shown in Fig. 7.5 (c), (d) and (b)

| $x_0\ x_1$ | $\overline{x_0}\ \overline{x_1}$ | Fig. 7.5 (c) | | | | Fig. 7.5 (d) | | | | Fig. 7.5 (b) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Logic gates | | | | Logic gates | | | | Logic gates | | | |
| | | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| 00 | 22 | 0 | **2** | **0** | 0 | 0 | **2** | **0** | 0 | 0 | **2** | **0** | 0 |
| 01 | 21 | 1 | **1** | **0** | 0 | 1 | **1** | **0** | 0 | 1 | **1** | **0** | 0 |
| 02 | 20 | 2 | **0** | **0** | 0 | 2 | **0** | **0** | 0 | 2 | **0** | **0** | 0 |
| 10 | 12 | 1 | **1** | **0** | 0 | 1 | **1** | **0** | 0 | 1 | **1** | **0** | 0 |
| 11 | 11 | 2 | **0** | **1** | 0 | 2 | **0** | **1** | 0 | 2 | **0** | **0** | 0 |
| 12 | 10 | 0 | **2** | **1** | 1 | 0 | **2** | **1** | 1 | 0 | **2** | **1** | 1 |
| 20 | 02 | 2 | **0** | **0** | 0 | 2 | **0** | **0** | 0 | 2 | **0** | **0** | 0 |
| 21 | 01 | 0 | **2** | **1** | 1 | 0 | **2** | **1** | 1 | 0 | **2** | **1** | 1 |
| 22 | 00 | 1 | **1** | **2** | 1 | 1 | **1** | **2** | 1 | 1 | **1** | **2** | 1 |

discussion that the proposed methods allows us to synthesize circuit when

$$A * B = A + B.$$

### 7.5.2 An one-digit full adder, add3_3c.pla

**Gate-level EHW:**

In this section we will consider some circuit designs evolved using the extrinsic gate-level EHW. The circuit structure discussed below has been evolved using 1x10 circuit layout, with connectivity parameter equals 10. The evolution of a population of 5 individuals has been carried out during 15,000 generations. The functional set of logic gates include NOT, MIN, MAX, MODSUM, MODPRODUCT, TSUM and TPRODUCT. The typical circuit structure evolved for an one-digit 3-valued full adder is shown in Fig. 7.6(a). This structure contains 6 logic gates and uses MODSUM, MODPRODUCT, TSUM and TPRODUCT. Note that the depth of the circuit is defined by the number of levels in the circuit. For example, there are 4 levels in the circuit shown in Fig. 7.6(a). The logic gates 1 and 2 belong to the first level.

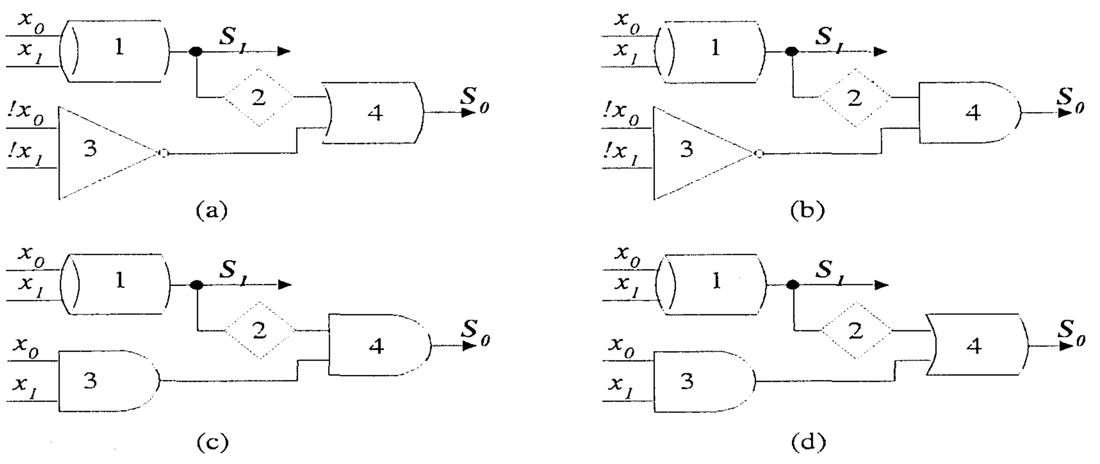


Figure 7.5: The evolved for 3-valued half adder designs.

Logic gates 3 and 4 are located in the second level. The logic gates 5 and 6 are positioned in the 3-rd and the 4-th levels respectively. It is interesting to note that in most circuits observed with 6 and 7 logic gates, the MODPRODUCT logic gate with identical inputs is used. The notable feature of the circuit shown in Fig. 7.6 (b) is that the logic operators from different logic algebras are utilised. The circuit structures have different configurations. In the case when the functional set of logic gates has been changed to {MIN, MAX, MODSUM}, the circuit design requiring 9 logic gates has been obtained (Fig. 7.6 (b)). In this case the depth of the circuit is 6. The implementation of digit $S_1$ is the same as in the previous case and it takes only two logic gates. So, this circuit structure requires more logic gates than the one shown in Fig. 7.6(a). This example illustrates how the functional set of logic gates used in evolution influences the circuit structures evolved.

**Function level EHW:**

Evolving a fully functional one-digit adder using a rectangular array with 10 columns and 1 row and with connectivity parameter equal to 10 proved to be relatively easy

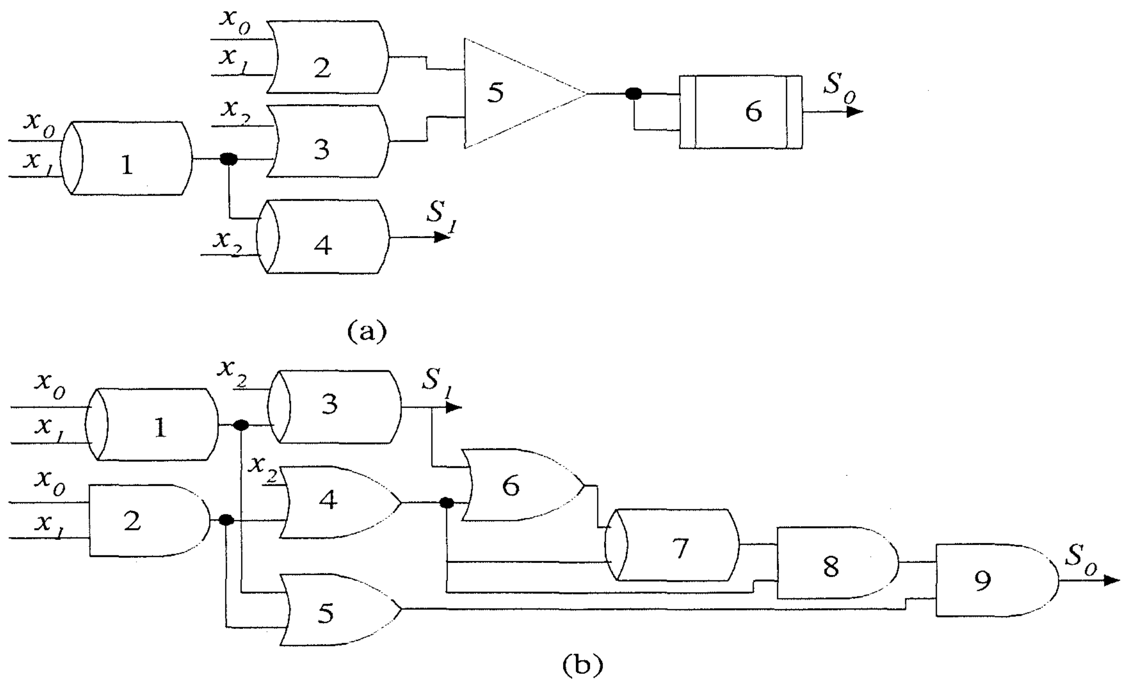


Figure 7.6: The evolved 3-valued full adders.

and the designs shown in Fig. 7.7 were obtained. The circuit shown in Fig. 7.7(a) contains only 3 building blocks and employs 9 primitive logic cells, because the optimal implementation of the half adder contains 4 primitive logic cells [7]. It is interesting to note that, in this structure, all outputs of the half adder have been used actively. When we choose the functional set such that the the MAX gate is not allowed to be used, the circuit structure shown in Fig. 7.7(b) is evolved. This structure has 4 building blocks and contains 10 primitive logic cells. Both structures mentioned above use half adders for all outputs.

### 7.5.3 An one-digit multiplier, mult3_2.pla

An one-digit multiplier multiplies the $r$-valued numbers $A_0$ by $B_0$ to produce the two-digit $r$-valued number $(P_1P_0)$, where $A_0$, $B_0$ and $P_1$ are the most significant digits.

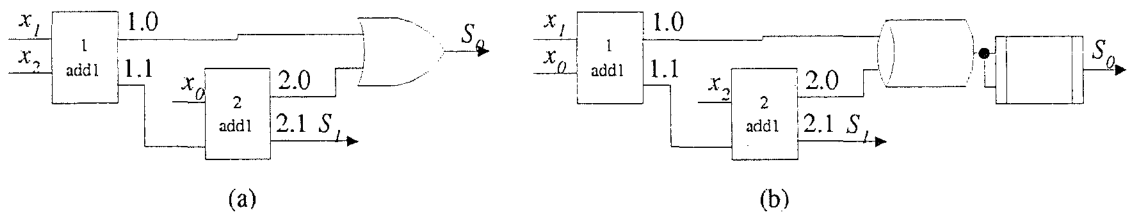


Figure 7.7: The 3-valued 1-digit full adders evolved using the 3-valued half adder and the one-digit multiplier.

Thus, this is a circuit with 2 inputs and 2 outputs and it requires 9 input and output conditions for full specification in the case of 3-valued logic and 16 in the case of 4-valued logic.

Some circuits implemented at gate-level are shown in Fig. 7.8. The smallest circuit contains only 3 logic gates and involves MODPRODUCT and TPRODUCT logic gates. Note that the outputs of this circuit are implemented separately. When the functional set of logic gates has been chosen by the way that contains no MODPRODUCT logic gate, the solution with 7 logic gates is obtained. It is clear from this example that the functional set of logic gates can drastically influence the number of active gates in the circuit.

### 7.5.4 A 1.5-digit multiplier, mult3_3.pla

**Function level EHW:**

A 1.5-digit multiplier multiplies the $r$-valued numbers $(A_1A_0)$ by $B_0$ to produce the two-digit $r$-valued number $(P_1P_0)$, where $A_1$, $B_0$ and $P_1$ are the most significant digits. This is a circuit with 3 inputs and 2 outputs and it requires 27 input and output conditions for full specification in the case of 3-valued logic. Modeling of the

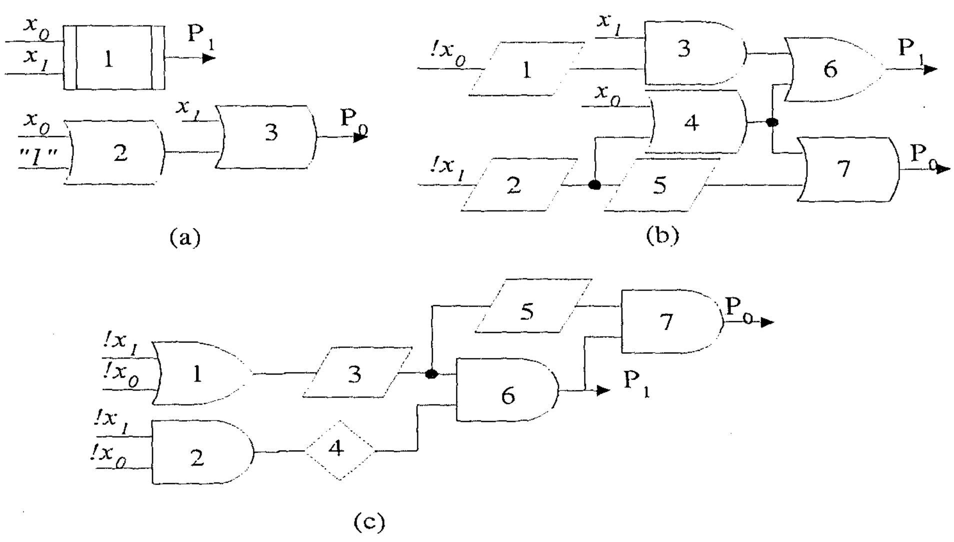


Figure 7.8: The evolved 3-valued one-digit multiplier.

multiplication process on the familiar long-multiplication method is shown below.

$$\begin{array}{cc} A_1 & A_0 \\ & B_0 \\ \hline A_1B_0 & A_0B_0 \end{array}$$

An example of circuit structures evolved using the proposed method are depicted in Fig. 7.9. It is interesting to note that in the case of circuit shown in Fig. 7.9(a) an EA uses the outputs of the half adder and the one-digit multiplier as the sub-functions and some of its outputs are uncommitted. Thus, the first output of the 3-valued 1-digit multiplier labeled 6 is not employed. At the same time, all outputs of the half adder are used. The circuit contains 7 building blocks and involves 12 primitive logic cells, because the implementation of a one-digit multiplier requires least 3 primitive logic cells. Fig. 7.9(b) shows the circuit evolved using functional set containing only a half adder from the standard functions. This circuit requires 6 building blocks and 12 primitive logic cells. Note, that the implementation of digit $P_1$ is the same for

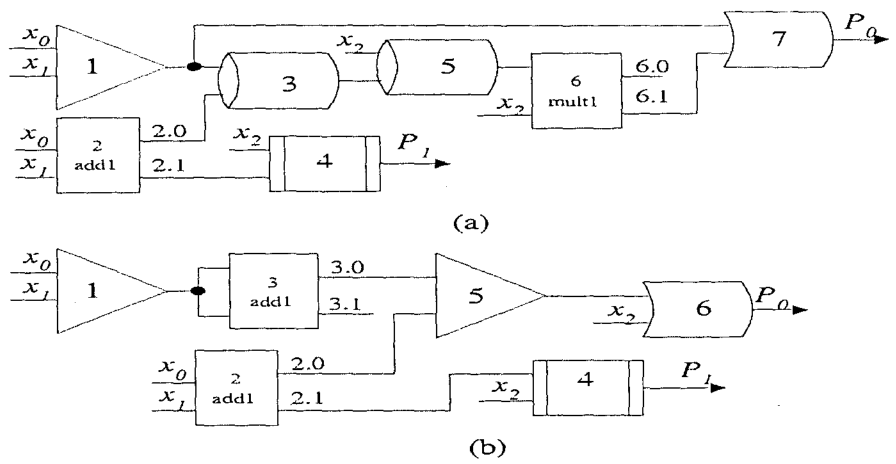


Figure 7.9: The 3-valued 1.5-digit multipliers evolved using the 3-valued half adder and the one-digit multiplier.

both cases. The circuits discussed above can not be obtained using the rules of the standard multiplication process.

## 7.6 Gate-level EHW and algorithm performance

In this section we will consider how the circuit layout and functional set of logic gates used influence the algorithm performance. For this purpose, the functional sets of logic gates containing primitive two-input logic operators have been chosen. Therefore, the gate-level EHW approach has been applied to synthesise multiple-valued logic functions.

### 7.6.1 Circuit layout and algorithm performance

The main purpose of this series of experiments is to investigate how the connectivity parameter and circuit layout affect the performance of the algorithm. The behaviour of the following circuit layout parameters have been investigated:

1. the connectivity parameter;
2. the number of rows in the rectangular array;
3. the number of columns in the rectangular array.

The following criteria are applied to estimate the algorithm performance:

1. the mean functionality fitness of the best chromosome over 100 runs, $av.F_1$, $\overline{F_1^{bf}}$;
2. the mean number of active gates in fully functional designs evolved over 100 runs, $av.100F_2$, $\overline{F_2^{bf}(\mathcal{N}_f)}$;
3. the number of fully functional circuits evolved, $\#100\%$ *cases*, $\mathcal{R}(\mathcal{N}_f)$.

In order to get the first positive results in a reasonable period of time we arrive at the parameters shown in Table 7.3. We perform three different types of experiment. Each of the experiments has different circuit layout parameters. The circuit parameters are listed separately for each type of experiment performed (Table 7.3).

**Connectivity parameter.**

In this series of experiments we investigate how the connectivity parameter influences the algorithm performance. Fig. 7.10 summarises the experimental results. For each set of runs the number of columns and rows has been fixed and the connectivity parameter allowed to vary. The algorithm performance for 3-valued one-digit half and full adders (add3_2.pla and add3_3c.pla), one-digit multiplier (mult3_2.pla) has been investigated. Remember that the connectivity parameter defines the flexibility of the internal connections in a circuit. The higher the connectivity parameter: the

Table 7.3: Initial data: Circuit layout and algorithm performance.

| Circuit | add3_2.pla | add3_3c.pla | mult3_2.pla |
|---|---|---|---|
| **Radix of logic, $r$** | 3 | 3 | 3 |
| **Functional set, $\mathbb{FS}$** | $\{1, 7, 8, 9, 10, 11, 12\}$ | | |
| **Type of algorithm** | $(1+\lambda)$ ES | $(1+\lambda)$ ES | $(1+\lambda)$ ES |
| **Population size, $\lambda$** | 5 | 5 | 5 |
| **Number of generations, $N_(gen)$** | 1000/100 | 15000 | 5000/500 |
| **Number of algorithm runs** | 100 | 100 | 100 |
| **Mutation rate, $p_m$** | 5% | 5% | 5% |
| **Investigation of connectivity parameter** | | | |
| **Circuit layout,** $N_{cols} \times N_{rows}$ | 25x1 | 25x1 | 25x1 |
| **Connectivity parameter,** $(N^{min}_{connect}, N^{max}_{connect}, \Delta N_{connect})$ | (1, 25, 1) | (1, 25, 1) | (1, 25, 1) |
| **Investigation of the number of columns** | | | |
| **Circuit layout,** $(N^{min}_{cols}, N^{max}_{cols}, \Delta N_{cols})$ $N_{rows}$ | (2, 25, 1) 1 | (16, 25, 1) 1 | (2, 25, 1) 1 |
| **Connectivity parameter,** $N_{connect}$ | $N_{cols}$ | $N_{cols}$ | $N_{cols}$ |
| **Investigation of the number of rows** | | | |
| **Circuit layout,** $(N^{min}_{rows}, N^{max}_{rows}, \Delta N_{rows})$ $N_{cols}$ | (2, 25, 1) 4 | - - - | (2, 25, 1) 4 |
| **Connectivity parameter,** $N_{connect}$ | 4 | - | 4 |

higher the flexibility of internal connections in the circuit. We examine how, with increasing the flexibility of the internal connections in the circuit, the mean fitness functions and the number of fully functional circuits evolved behave.

Fig. 7.10(a), (b) show how ES performance depends on the connectivity parameter after 100 and 1000 generations for add3_2.pla. Analysing this data we can conclude that, by increasing the number of generations, the algorithm performance is improved in terms of the number of the fully functional circuit evolved. Thus, in the case where the connectivity parameter is 25, the number of fully functional circuits evolved after 1, 000 generations is 5 times higher than after 100 generations. Increasing the number of generations, we can obtain 100 fully functional solutions out of 100. Another important issue of these results is that the number of fully functional circuits evolved depends on the connectivity parameter. Thus, in case of the one-digit half adder, the number of fully functional designs evolved is significantly increased, when the connectivity parameter is more than 12. It is necessary to note that very few fully functional designs have been evolved with a connectivity parameter less then 5. A similar conclusion can be made analysing experimental data for other two logic functions (Fig. 7.10(c), (d), (e)). In all cases the algorithm performance has been improved in terms of the number of fully functional solutions evolved.

Let us consider how the quality of fully functional circuits evolved depends on the connectivity parameter. The quality of evolved circuits can be defined by the number of active primitive logic gates used in circuit ($F_2$ criteria). It is interesting to note that there is a domain where circuits with a relatively high number of active gates are evolved. This has been observed for all functions evolved. Thus, in the case of the one-digit half adder, circuits with more than 10 logic gates have been evolved with $2 \leq N_{connect} \leq 12$. This is a very large number of logic gates in comparison with optimal design, which involves only 4 primitive logic gates.

This can be explained as follows. A smaller connectivity parameter provides a higher level of constrains on the internal connectivity of the circuits and the circuit outputs. In this case a limit number of logic gates can be specified as circuit output: $N_{connect} * N_{rows}$. Let the "self-reproductive" logic circuit be the circuit that acts as a wire. For example, the 3-valued circuit "$NOT(NOT(x_1))$" is "self-reproductive", because it implements itself and acts as a wire. The depth of logic circuit defines how many levels is required to be implemented. Thus, the depth of $S_1$ is 1 and the depth of $S_2$ is 3 in add3_2.pla (see Fig. 7.5 (c)). Therefore, the minimal depth of a circuit is 1 and the maximal one is 3. The following statements can be made:

*Statement 1.* If the minimal depth of fully functional design is $N_{levels}^{design}$ and the functional set of logic gates contains *no* "self-reproductive" logic operations, then the fully functional circuit can be evolved if and only if

$$N_{connect} \geq \frac{N_{cols}}{N_{levels}^{design}}. \tag{7.6.1}$$

*Statement 2.* If the minimal depth of a fully functional design is $N_{levels}^{design}$, the number of logic gates in the $j$-th level is $N_{a.g.}^{j}$ and the functional set of logic gates contains "self-reproductive" logic circuits of $N_{sr.g.}$ logic gates the fully functional circuit can be evolved in *any* case. The minimum number of active logic gates in an evolved circuit design is defined as follows:

$$N_{gates}^{min} = \sum_{j=0}^{N_{levels}^{design}-1} N_{a.g.}^{j} + max(]\frac{\frac{N_{cols}}{N_{levels}^{design}} - N_{connect}}{N_{sr.g.}}[, N_{sr.g.}). \tag{7.6.2}$$

where $]x[$ defines the higher integer value.

Let us consider the one-digit half adder circuit evolved with the following parameters: $N_{connect} = 5$, $N_{cols} = 25$, $N_{levels}^{design} = 1$, $N_{a.g.}^{0} = 2$, $N_{a.g.}^{1} = 2$, $N_{a.g.}^{2} = 1$. The functional set used contains a successor operator that can be "self-reproductive", if the circuit $SUCCESSOR(SUCCESSOR(SUCCESSOR(x)))$ is implemented. Therefore, $N_{sr.g.} = 3$. The minimum number of logic gates in an evolved circuit is calculated as follows: $N_{gates}^{min} = 2 + 2 + 1 + (25 - 5)/3 = 12$. Experimental results show that the mean number of active logic gates in a fully functional circuit evolved is 17.6923. This means that, in most cases, the optimal possible design has not been obtained. A similar analysis of other evolved circuits shows that, in most cases, optimal possible designs have been evolved if the number of generations was large enough.

**The number of columns.**

In this series of experiments we investigate how the number of columns in a rectangular array influences the ES performance. The maximum possible connectivity parameter equal to the number of columns in rectangular array has been employed. Experimental data are shown in Fig. 7.11. Let us define the saturation point as the value after that there is no any improvements in the evolutionary process. Analysing the experimental data we can conclude that there is a saturation point where by increasing the number of columns in the circuit, the number of fully functional designs evolved is not increased. Thus, after this point there is no need to increase the number of columns in the rectangular array. For example, the saturation point for one-digit half adders is 10. Note that the number of active gates in the fully functional evolved circuits is increased with the increase in the number of columns in the rectangular array.

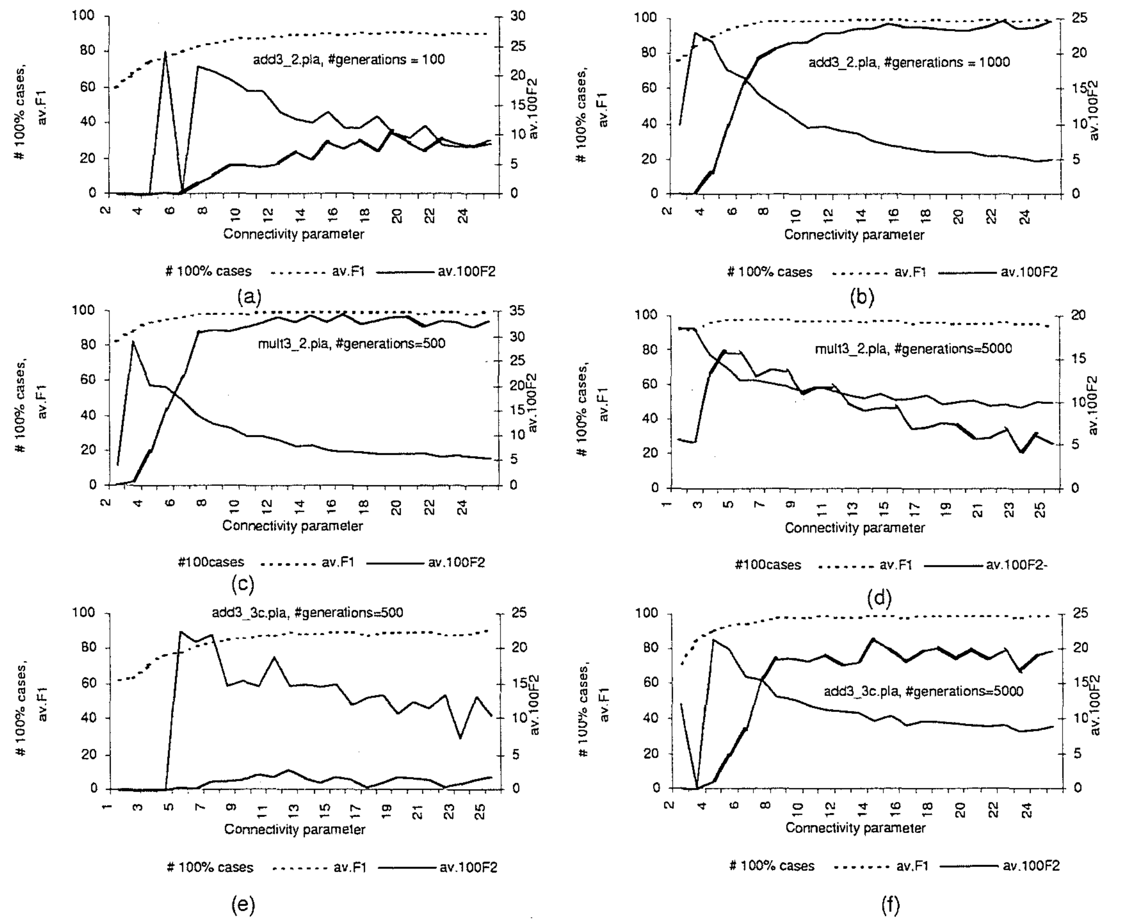


Figure 7.10: Dependence of the algorithm performance on the connectivity parameter. These graphs shows that the algorithm performance depends on the connectivity parameter. There is a range within which the algorithm performs very poorly. The curves are similar to ones, obtained for binary logic design problem (see Fig. 4.1). This confirms that the approach behaves similar to both application tasks.

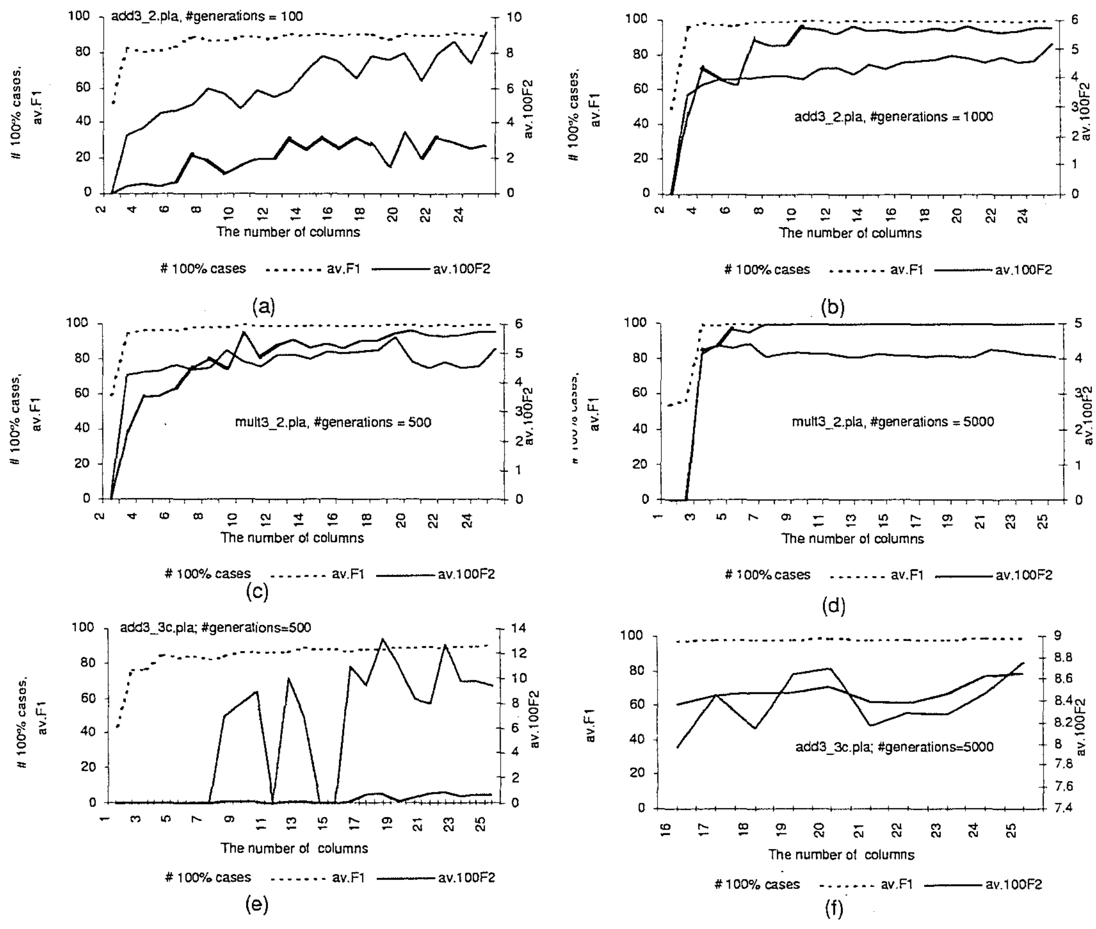


Figure 7.11: Dependence of the algorithm performance on the number of columns. The experiments depicted in these graphs have been performed with connectivity parameter $N_{connect} = N_{cols}$, i.e. there is no connectivity restrictions in the circuit. First, graphs show how with increasing the number of generations the behaviour of algorithm changes. Thus, with increasing the number of generations the algorithm performance has been significantly improved. Comparing the obtained data with ones illustrated in Fig. 4.3 for binary logic design problem, one can notice that the curves illustrate the same behaviour. This proves that the algorithm behaves similar for both problems.

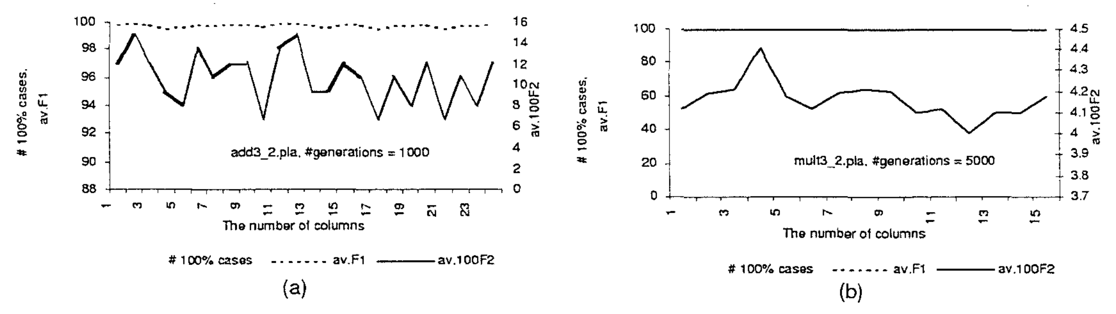


Figure 7.12: Dependence of the algorithm performance on the number of rows. Graphs illustrate that the algorithm performance does not depend on the number of rows in the rectangular array. Similar results have been obtained for binary combinational logic design problem (see Fig. 4.4).

### The number of rows.

In this series of experiments we investigate how the number of rows in a rectangular array influences the algorithm performance. For this purpose the number of columns and the connectivity parameter have been chosen to be constant (Table 7.3). Analysing the experimental data shown in Fig. 7.12 we can conclude that the ES performance does not depend on the number of rows in the rectangular array. Thus, there are no improvements, in the algorithm performance with increasing the number of rows in the rectangular array.

So, finally, we can conclude that the algorithm performance depends on the connectivity parameter and the number of columns in a rectangular array and does not depend on the number of rows. The number of active primitive logic gates in evolved structures strongly depends on the connectivity parameter.

### 7.6.2 Influence the functional set of logic gates used on the algorithm performance

In this section we will discuss how algorithm performance depends on the set of MVL gates chosen for circuit design. This analysis shows us how effective algorithm can be.

The initial data for experimental results, shown in Table 7.5, are reported in Table 7.4. ES performance can be evaluated by looking at the number of 100% functional solutions evolved during 100 runs. In order to make a fair comparison of the results obtained, the circuit layout and connectivity parameter have been fixed for all functional sets of logic gates used. In this case we do not only test the possibility of evolving the circuit, but also the possibility of evolving the circuit with the smallest number of active gates. Note, that if we were unable to evolve a 100% functional solution based on these parameters, it does not mean that it is impossible to evolve it using our method. Increasing the circuit layout usually allows us to evolve the circuit. The duration of evolution (the number of generations) was fixed for each of the tested functions. The functional set of logic gates used in evolution should contain the subset of logic operators which have been proved to be a functionally complete basis. Note, that the logic gates with inverted inputs or outputs have not been used.

Let us consider the evolution of a half adder using a different set of logic gates. It is clear from experimental results that the functional sets {NOT, MIN, MAX, TSUM, TPRODUCT} and {NOT, MIN, MAX, TSUM, TPRODUCT} are not worth using, because, during 100 runs, the 100% functional solution has not been achieved. Note, that the solution has been evolved only when the circuit layout has been increased. But, in the case when the MODSUM and MODPRODUCT are in the functional set,

Table 7.4: Initial data: Functional set of logic gates and algorithm performance.

| **Circuit** | add3_2 | add3_3c | mult3_2 | mult3_3 |
|---|---|---|---|---|
| **EHW parameters** | | | | |
| **Radix of logic,**$r$ | 3 | 3 | 3 | 3 |
| **Circuit layout,** $N_{cols}$x$N_{rows}$ | 10x1 | 10x1 | 10x1 | 10x1 |
| **Connectivity parameter,** $N_{connect}$ | 10 | 10 | 10 | |
| **EA parameters** | | | | |
| **Number of generations,** $N(gen)$ | 1000 | 15000 | 25000 | 25000 |
| **Type of algorithm** | $(1+\lambda)$ ES | $(1+\lambda)$ ES | $(1+\lambda)$ ES | $(1+\lambda)$ ES |
| **Population size,** $\lambda$ | 5 | 5 | 5 | 5 |
| **Number of algorithm runs,** $\mathcal{R}(\mathcal{N})$ | 100 | 100 | 100 | 100 |
| **Mutation type** | Circuit mutation | | | |
| **Mutation rate** | 0.05 | 0.05 | 0.05 | 0.05 |

100% functional solutions have been evolved. It seems that the algorithm actively uses these two operators during evolution of the half adder. An analysis of the circuit evolved proves this. Thus, in all cases the *Sum* digit has been implemented using the MODSUM operator. Analysis of the average number of active gates in the 100% functional circuits evolved shows us that the better solutions ($\overline{F_2^{bf}(\mathcal{N}_f)} = 3.8$) have been evolved using {NOT, TSUM, TPRODUCT, MODSUM, MODPRODUCT}.

Similar results have been obtained for the 3-valued full adder. It has been easy evolved using the {NOT, MIN, MAX, TSUM, TPRODUCT, MODSUM, MODPRODUCT} functional set. Comparing the results obtained for this set of logic gates (72 functional solution out of 100), it is clear that there is a favourable set of logic gates which allows us relatively easily to evolve the target circuit. In terms of the number of active gates, the better solutions have appeared when the set of {NOT, MIN, MAX, MODSUM, MODPRODUCT} is utilized. Similar results have been obtained for the one-digit adder, where the "favourite" functional set of logic gates appears to be {SUCCESSOR, MIN, MAX, TSUM, TPRODUCT, MODSUM,

Table 7.5: Experimental Results: Functional set of logic gates and algorithm performance. Experimental data shows that the algorithm performance strongly depends on the functional set of logic gates chosen. Similar conclusion has been made for binary combinational logic design problem (see Appendix B).

| **Circuit** | $n$ | $m$ | **Functional set** | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}$ | $\overline{F_1^{bf}(\mathcal{N}_f)}$ | $\mathcal{R}(\mathcal{N}_f)$ |
|---|---|---|---|---|---|---|---|
| add3_2 | 2 | 2 | 2-7-8-9-10 | 91.5 | 5.02 | 4.75 | 29 |
| | | | 2-7-8-11-12 | 79.6666 | 4.48 | 0 | 0 |
| | | | 2-9-10-11-12 | 90.8889 | 4.1 | 3.8 | 20 |
| | | | 2-7-8-9-10-11-12 | 92.5555 | 4.21 | 3.59 | 32 |
| | | | 1-7-8-9-10 | 90.6667 | 4.94 | 4.57 | 26 |
| | | | 1-7-8-11-12 | 80.0555 | 4.45 | 0 | 0 |
| | | | 1-9-10-11-12 | 89.55 | 4.19 | 4.0 | 17 |
| | | | 1-7-8-9-10-11-12 | 90.94 | 4.24 | 4.26 | 23 |
| add3_3c | 3 | 2 | 2-7-8-9-10 | 93.6667 | 7.28 | 9 | 1 |
| | | | 2-7-8-11-12 | 63.5185 | 8.44 | 0 | 0 |
| | | | 2-9-10-11-12 | 86.1852 | 5.31 | 8 | 1 |
| | | | 2-7-8-9-10-11-12 | 98.7593 | 6.97 | 7 | 72 |
| | | | 1-7-8-9-10 | 94.1667 | 5.63 | 6.07 | 14 |
| mult3_2 | 2 | 2 | 2-7-8-9-10-11-12 | 94.2222 | 7.35 | 6.07 | 14 |
| | | | 2-9-10-11-12 | 93.5555 | 5.69 | 6.0 | 5 |
| | | | 2-7-8-11-12 | 83.3333 | 3.99 | 0 | 0 |
| | | | 1-7-8-9-10 | 94.1667 | 5.63 | 6.07 | 14 |
| | | | 1-7-8-11-12 | 95.1667 | 5.57 | 6.07 | 13 |
| | | | 1-7-8-9-10-11-12 | 99.8889 | 3.11 | 3.09 | 98 |
| mult3_3 | 3 | 2 | 2-7-8-9-10-11-12 | 76.5556 | 7.67 | 0 | 0 |
| | | | 1-7-8-11-12 | 59.2592 | 4.59 | 0 | 0 |
| | | | 2-9-10-11-12 | 77.9444 | 8.35 | 0 | 0 |

In this Table the functional set of logic gates is encoded in the following way: 1-SUCCESSOR, 2-NOT, 7-MIN, 8-MAX, 9-MODSUM, 10-MODPRODUCT, 11-TSUM, 12-TPRODUCT. $\overline{F_1^{bf}}$ and $\overline{F_2^{bf}}$ are the mean fitnesses $F_1$ and $F_2$ of the best evolved chromosomes respectively; $\overline{F_2(\mathcal{N}_f)}$ is the mean fitness function $F_2$ of fully functional designs evolved during 100 runs; R($\mathcal{N}_f$) is the number of evolved fully functional circuits, $\mathcal{N}_f$

MODPRODUCT}. In this case 98 fully functional solutions out of 100 have been evolved. The attempts to evolve 1.5 digit multipliers using functional sets of logic gates listed in Table 7.5 do not succeed. There are three reasons for this. First, the number of generations defined is not enough to achieve 100% functional solution. Second, the circuit layout is not large enough to evolve the circuit. And finally, we didn't chose the right set of logic gates for evolution.

Thus, we can conclude that the performance of the algorithm as well as the quality of circuit evolved in terms of the number of logic gates used in the circuit, strongly depends on the functional set of logic gates used in evolution.

## 7.7 Comparison of function and gate level EHW

The following experiment allows us to compare the algorithm performance of the function and gate level evolvable hardware approaches. In [7] a two-input one-output chromosome representation has been proposed to design multiple-valued combinational circuits. This representation allow us to employ any primitive MVL functions as well as a T-gate. A T-gate is a prototype multiplexer in MVL design. It uses $r$ inputs which are controlled by $(r+1)$-th input. Thus, the T-gate has $(r+1)$ inputs and one output. Similar experiments were performed for the chromosome representations discussed above and reported in [7]. The initial data for the algorithm and circuit layout are given in Table 7.6. In order to make fair comparison of the algorithm performance we choose the same conditions for both chromosome representations. The difference is that for multi-input multi-output chromosome representation we add some standard logic functions into functional set, such as the half adder and the one-digit multiplier. We evolved logic circuits for the one full digit adder and the 1.5

digit multiplier.

The experimental results obtained are summarised in Table 7.6. It is clear that, in terms of 100% functional solutions evolved, the proposed function-level EHW approach works better. Thus, the number of functional solutions obtained using the proposed method (shown in bold in the last column) has been drastically improved. Adding only the half adder into the functional set allows us to radically improve the algorithm performance in terms of the number of 100% functional solutions obtained. For example, the algorithm with 2-7-8-11-12 logic gates found no positive solutions for a full adder. At the same time, adding a half adder into the functional set allows us to easily evolve the full adder, and we obtained 77 solutions out of 100 possible. Our attempts to evolve the 1.5 digit multiplier with the initial parameters mentioned in Table 7.6 produces no positive results. But when we allow the use of the half adder and an one–digit multiplier in the evolutionary process, some functional circuits have been evolved.

It is interesting to note that, given the larger building blocks to evolve, we improve the ES performance in terms of the number of 100% functional solutions evolved. Based on the experimental results shown above it is clear that the proposed method allows us to extend the set of functional gates used in evolution to multi-input multi-output building blocks. This extension gives us more choice in terms of building blocks used in evolution.

## 7.8 Summary

In this chapter the evolvable hardware approach applied to the multi-valued logic design has been considered. The arithmetical logic functions have been chosen as the

Table 7.6: Initial data: Performance of gate and function level extrinsic evolvable hardware approaches.

| Circuit | add3_3c | mult3_3 |
|---|---|---|
| **EHW parameters** | | |
| **Radix of logic** | 3 | 3 |
| **Circuit layout** | 10x1 | 10x1 |
| **Connectivity parameter** | 10 | 10 |
| **EA parameters** | | |
| **Type of algorithm** | $(1+\lambda)$ ES | $(1+\lambda)$ ES |
| **Population size, $\lambda$** | 5 | 5 |
| **Number of generations, $N_{gen}$** | 15000 | 25000 |
| **Number of algorithm runs, $\mathcal{R}(\mathcal{N})$** | 100 | 100 |
| **Mutation type** | Circuit mutation | |
| **Mutation rate, $p_m$** | 0.05 | 0.05 |

Table 7.7: Experimental Results: Performance of the gate and the function level extrinsic evolvable hardware. Experimental data obtained show that the function-level EHW approach performs better than the gate-level EHW. This confirms the universality of the EHW approach in question, since similar results have been obtained when the binary combinational logic design problem has been considered (see Chapter 5). $\overline{F_1^{bf}}$ and $\overline{F_2^{bf}}$ are the mean fitnesses $F_1$ and $F_2$ of the best evolved chromosomes respectively; $\overline{F_2(\mathcal{N}_f)}$ is the mean fitness function $F_2$ of fully functional designs evolved during 100 runs; $\mathrm{R}(\mathcal{N}_f)$ is the number of evolved fully functional circuits, $\mathcal{N}_f$

| **Circuit** | $n$ | $m$ | **Functional set** | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}$ | $\overline{F_2(\mathcal{N}_f)}$ | $\mathrm{R}(\mathcal{N}_f)$ |
|---|---|---|---|---|---|---|---|
| add3_3c | 3 | 2 | 2-7-8-11-12 | 63.5185 | 8.44 | 0 | 0 |
| | | | **2-7-8-11-12-22** | 98.7778 | 9.12 | 8.86 | **77** |
| | | | 2-7-8-9-10-11-12 | 98.7593 | 6.97 | 7 | 72 |
| | | | **2-7-8-9-10-11-12-22** | 99.7593 | 6.77 | 7 | **96** |
| | | | 2-9-10-11-12 | 86.1852 | 5.31 | 8 | 1 |
| | | | **2-9-10-11-12-16-22** | 99.2222 | 7.87 | 8.069 | **86** |
| mult3_3 | 3 | 2 | 2-7-8-9-10-11-12 | 76.5556 | 6.64 | 0 | 0 |
| | | | **2-7-8-9-10-11-12-22** | 98.3889 | 8.4 | 9.39 | **61** |
| | | | **2-7-8-9-10-11-12-16-22** | 97.5741 | 8.69 | 9.56 | **41** |
| | | | 1-7-8-11-12 | 59.2592 | 8.4 | 0 | 0 |
| | | | **1-7-8-11-12-16-22** | 92.1851 | 13.89 | 12.4 | **10** |
| | | | 2-9-10-11-12 | 77.9444 | 8.3 | 0 | 0 |
| | | | **2-9-10-11-12-22** | 93.9999 | 7.94 | 9.8 | **26** |
| | | | **2-9-10-11-12-16-22** | 96.5741 | 8.13 | 10.11 | **17** |

tested functions. The gate and function-level EHWs are applied to evolved multi-valued logic functions. The evaluation process is performed using dynamic fitness function. A number of experiments has been carried out in order to investigate the behaviour of EHW applied to the multi-valued logic design. The experimental results confirm that there are no differences in the behaviour of EHW applied to binary and multi-valued logic design. Thus, the algorithm performance depends strongly on the choice of the functional set of logic gates, the circuit layout, algorithm parameters. The multi-valued logic functions have been evolved using the gate and function level EHW. The experimental results verify that the function level EHW executed better if suitable multi-input multi-output logic functions are chosen. This applies to binary logic design as well [4]. Therefore, we can conclude that the extrinsic EHW applied to multi-valued logic design behaves similarly to the EHW applied to the binary logic design.

# Chapter 8

# Conclusions

In this dissertation a self-adaptive function-level extrinsic EHW approach with bidirectional incremental evolution is presented. The approach is specifically designed to evolve large circuits implementing logic functions of large number of inputs and outputs. The evaluation and evolutionary processes have been studied in detail in order to define the specific features of the extrinsic EHW approach and applied to overcome a number of problems. From the experimental results presented in this dissertation, some features of extrinsic EHW approach can be identified.

Firstly, the considered extrinsic EHW approach is universal in terms of circuit implementation technology used and ability to evolve various types of circuits. The adaptation of an extrinsic EHW approach to the FPGA and MOS technologies is discussed in Chapter 4. It is shown that varying the optimisation criteria, any target implementation technology can be used to evolve circuits. Further, the behavioural features of the extrinsic EHW approach have been investigated for: binary and multi-valued combinational logic design. The results of this empirical investigation shows that EHW behaves in a similar manner in both cases. Thus, in both cases, the algorithm performance strongly depends on the circuit layout chosen and the functional set of logic gates. Also, the function-level EHW approach produces better results in comparison with the gate-level EHW method for both tasks.

Secondly, the approach discussed in this dissertation evolves fully functional *cost-optimised* logic circuits. The dynamic fitness function employed has the following features:

- improves the quality of evolved circuits producing efficient logic circuits.
- requires less computational efforts (in other words, the evaluation of the number of primitive active logic gates in the circuit is carried out only during the second stage of the evaluation process in contrast to multi-objective fitness functions, where the circuit has to be evaluated in terms of the number of primitive active logic gates and the circuit functionality during every evolutionary step.);
- produces two different evolutionary processes: evolution towards a fully functional circuit and evolution towards an optimised system.

Thirdly, the extrinsic EHW can be self-adaptive in terms of defining the EHW parameters during evolution. In Chapter 4 the possibility of evolving the circuit layout together with circuit functionality is discussed. Empirical study of this case shows that the evolutionary algorithm automatically defines the "favourite" range of circuit layout, where the best results can be produced. The advantage of the self-adaptive EHW approach is that it defines the circuit layout automatically. In other words it adapts to the complexity of the task given and finds the most suitable EHW parameters. One of the disadvantages is the difficulty to evolve logic functions of large numbers of inputs and outputs using the self-adaptive EHW approach. This is due to the fact that the circuit functionality is analysed by the truth table that describes the complex task. Another aspect that has to be mentioned is that the computation time of the proposed method depends on the way the method is implemented. For example, if memory is allocated to each chromosome with a new circuit layout, the performance is significantly slowed down because the memory allocation requires a lot of computational resources.

The idea of function-level extrinsic EHW approach introduced in this dissertation lies in using multi-output logic components as a building blocks. As a result, the logic circuits can be synthesised using higher complexity sub-functions. This approach allows a reduction of the size of chromosome genotype; improve the algorithm performance and adapts to the FPGA-based circuit design. One disadvantage of extrinsic EHW approach remains the same: the approach is not suitable for evolving logic circuits of large number of inputs and outputs, since the evaluation is based on the analysis of the truth table of the complex system.

Finally, the bidirectional incremental evolution is applied to the extrinsic EHW approach. The purpose of this extension is to overcome the "stalling" effect of direct evolution and facilitate to the evolution of complex systems. The bidirectional incremental evolution combines the priori knowledge and the specific features of evolution. The proposed approach has a number of advantages:

1. there are no restrictions on the complexity of evolved circuits in terms of the number of inputs and outputs;

2. there are no limitations on the application task;

3. there is improvement in terms of computational effort, since the strength of this method is to evolve sub-circuits rather than the complex circuit directly;

4. it works mostly with small chromosomes.

One of the difficulties in using this method in another application task can be the definition of metrics. Once the metrics are defined, the method can be easily adapted to the task.

As a result of this research an ECAD software tool called Discovery v.13 to investigate the extrinsic EHW approach has been developed. The created software is designed to support research at different level of extrinsic EHW. The designed software has high level of flexibility in terms of the parameter selection. Besides that, the tools to process the obtained results, are created as well. Thus the progress of evolutionary algorithm can be displayed graphically, evolved circuits can be drawn schematically and the analysis of evolved circuits can be performed. The software can be used for educational and research purposes.

The main contributions of the thesis can be summarised as follows:

- *A self-adaptive function-level EHW approach that is capable of evolving fully functional cost-optimised circuits has been designed and investigated in detail.*
- *An approach has been developed for evolving complex combinational logic circuits in a framework where the functional dependencies of input data are known.*
- *The ECAD software (Discovery v.13), that is suitable to carry out research of extrinsic EHW approach and to learn the basics of EHW approach, has been created.*

## 8.1 Future work

Some issues that become very important in the extrinsic EHW approach, such as evaluation process, self-adaptation features of extrinsic EHW, function-level EHW, etc. are discussed in detail in this dissertation. Further work can be summarised as follows:

- It would be beneficial to extend the extrinsic EHW approach to design sequential circuits. This may be one of directions of future investigations.
- The quality of evolved circuits is estimated in terms of the number of logic gates used in the circuit or the number of transistors. Obviously more precise evaluation of evolved circuits can give a more realistic picture about features of generated solutions, such as circuit delay, wire allocation, etc.. In this case the evaluation can be implemented using a VHDL simulator.
- The self-adaptation of extrinsic EHW is defined by the circuit layout. The importance of a functional set of logic gates in the evolution process is discussed in Appendix B. It is shown that the EA performance significantly depends on the chosen functional set of logic gates. Since in bidirectional incremental evolution the structure and complexity of simpler tasks vary all the time, the choice of the functional logic set can significantly influence the algorithm performance. Therefore, it is very important to design an EHW approach that would be self-adaptive in terms of the circuit layout and the functional set of logic gates.
- The function-level EHW approach is based on the choice of higher complexity logic functions in the functional set of logic gates. This is very convenient when

the basic functional dependencies of considered task are well-known. In case of applying this method to bidirectional incremental evolution some problems can occur, since the specific features of synthesized smaller tasks are unknown. In this case using the chromosome representation introduced in Chapter 5, the structure of complex building blocks can be automatically changed.

- It has been shown that the proposed bidirectional incremental evolution performs very well in comparison with direct evolution in the extrinsic EHW approach. It has been shown that the extrinsic EHW approaches in terms of specific features of evolutionary processes and choice of EHW parameters behave in a similarly way to intrinsic EHW. From another point of view, Xilinx introduced a new reconfigurable device namely Virtex FPGAs, designed to use facilities of the internet. Therefore, a useful future research is to implement EHW using Virtex FPGAs that would implement bidirectional incremental evolution. Future work will be focused on investigation of the behaviour of bidirectional incremental evolution applied to intrinsic EHW. In this case a number of problems have to be overcame. First, new metrics suitable to intrinsic EHW have to be introduced. Next, the problem of evolving complex tasks using several FPGAs has to be solved. A method to map heterogeneous circuit layouts into FPGA has to be developed.

In the future the use of bidirectional incremental evolution applied to EHW is likely to grow, with some movement towards analogue circuit design. This should provide a firm basis for the construction of mixtrinsic EHW systems solving complex problems, which requires a generalization of the technique.

# Appendix A

# Appendix. Digital circuit design

In this appendix we will consider the basics of digital circuit design, including implementation technologies, description of the basic logic gates and combinational digital building blocks.

## A.1 Implementation technologies

In this dissertation we will apply EHW approach for FPGA devices and MOS technology. Each of these technologies has some distinctive features, that will be considered in this section.

### A.1.1 FPGA

Field programmable gate array (FPGA) devices manufactured by Xilinx are primarily for digital design. In many instances, these devices can be erased and re-programmed.

In general, FPGAs are comprised of *logic blocks, I/O cells* and *interconnection lines.* The logic blocks implement the actual logic of the FPGA using primitives such as NAND gates, multiplexers, lookup tables. The I/O cells allow the FPGA's logic blocks to connect to the FPGA's pins. The interconnection lines connect logic blocks

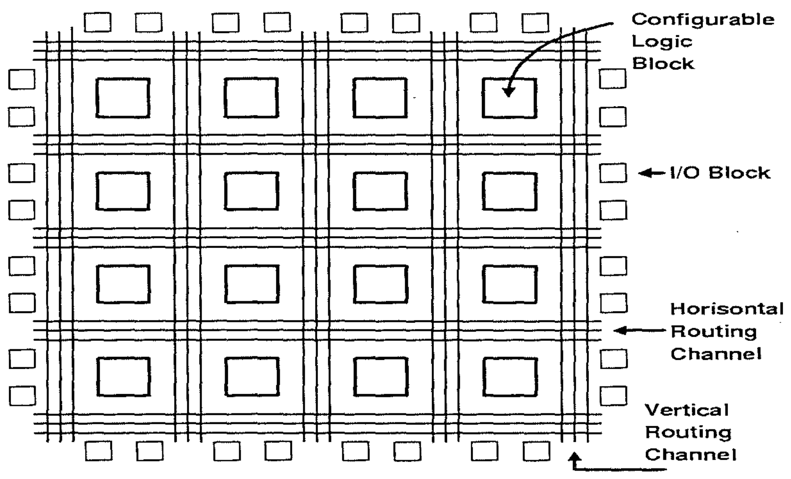


Figure A.1: Overall view of a Xilinx FPGA (courtesy [10]).

to each other and to the I/O cells. The routing done by these lines is implemented with wire segments and a system of programmable switches. The switching technology can be any one of the pass-transistors controlled by static RAM cells, anti-fuses, EPROM transistors or EEPROM transistors.

FPGAs were first created by Xilinx, Incorporated in 1984. Since that time, many other companies have marketed FPGAs, the major companies being Cilinx, Actel and Altera.

### A.1.2 Xilinx FPGA

Xilinx FPGAs use static RAM technology to implement hardware designs. Because of this they are reprogrammable and frequently used in prototyping and other areas where reprogrammability is useful.

A Xilinx FPGA consists of a two-dimensional array of configurable logic blocks (CLBs), a set of surrounding input/output blocks (IOBs) and programmable interconnections between different CLBs and between CLBs and IOBs (Fig. A.1). Each

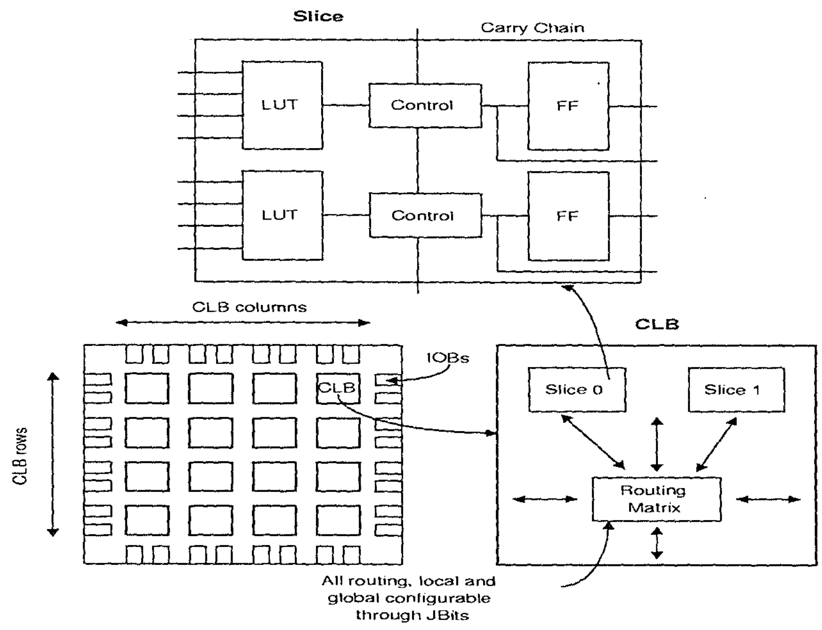


Figure A.2: Virtex Architecture (courtesy [11]).

CLB can implement two arbitrary, independent four-input Boolean functions, $F$ and $G$. The outputs of $F$ and $G$ can be combined with another input in a third Boolean function $H$. Each CLB also has the capacity to implement fast-carry logic. Connections between CLBs and the chip pads are provided by IOBs. The IOBs offer many user-programmable options in I/O control, including tri-state logic for bidirectional I/O, direct connection of lines or connection through, flip-flops, and programmable pull-up or pull-down resistors. The IOBs also provide logic for boundary-scan testing and output slew rate control. The interconnect between different CLBs and between CLBs and IOBs is also programmable. Xilinx FPGAs shown in Fig. A.2 can be applied to evolvable hardware systems and it is simulated in this thesis.

The Virtex family uses a standard FPGA architecture as in Fig. A.2. The logic is divided into an NxM structure of Configurable Logic Blocks (CLB), each block contains a routing matrix and two slices. Each slice contains two Lookup Tables

(LUTs) and two registers (in addition to considerable amounts of logic not used in these examples). The inputs to the slice are controlled through the routing matrix which can connect the CLB to the neighboring CLBs through single lines which terminate six CLBs away. Each of the inputs to the CLB can be configured using the JBits API classes, as well as all the routing. This means that it is possible to route two outputs together creating a bitstream which will damage the device. If it was also possible to program the device from outside the computer, damage could be terminal, something that the Java security was developed to prevent.

### A.1.3 Unipolar logic families

Unipolar logic families are based on the field effect transistor which requires a metal electrode separated from a semiconductor channel by an oxide insulating layer. This is the MOS fabrication technology, and the individual transistors are often referred to as MOSFETs [167]. In our work we will consider the implementation of logic gates using different MOS technologies, because MOS logic devices do not require internal resistors on the chips, and can hence be manufactured at a high packing density. The fabrication process is relatively simple, and power consumption is low.

#### PMOS logic

PMOS logic is a family of MOS integrated circuits. The semiconductor material in p-doped and the majority carries are therefore holes. PMOS is suited to large scale integration and has a greater packing density capability than bipolar transistor logic.

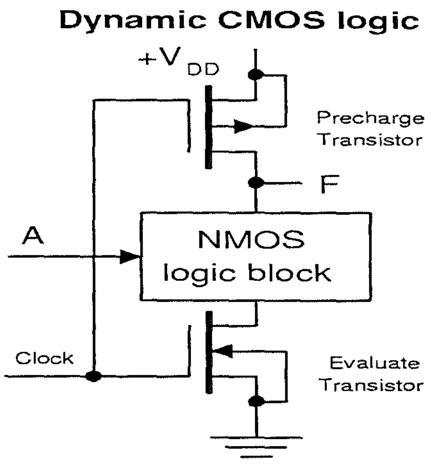


Figure A.3: CMOS dynamic gate.

### NMOS logic

NMOS logic is basically the same structure as PMOS, except that it uses n-doped semiconductor material. The circuit carriers in NMOS are free electrons, which are more mobile that larger positive charges, and this results in faster switching times compared with PMOS devices.

### CMOS complementary logic

Complementary metal oxide semiconductor (CMOS) logic uses both p- and n-type channels in the same circuit. It is faster than PMOS and NMOS and requires considerable less power that the low power TTL series. MOS circuits can operate off a wide range of supply voltages. All complementary gates may be designed as ratioless circuits. That is, if all transistors are the same size the circuit will function correctly.

### Dynamic CMOS logic

CMOS dynamic gates are used when circuit delay is important. Dynamic gates require clocks, and involve circuit and timing design [168]. A basic dynamic CMOS gate is shown in Fig. A.3. It consists of an n-transistor logic structure whose output node is precharged to $V_{DD}$ by a p-transistor (precharge) and conditionally discharged by an n-transistor (evaluate) connected to $V_{SS}$ [129]. Note that the dynamic gates are faster than the static gates under all conditions [127]. The dynamic gate is faster for pulldown than the static gate. This speed advantage is due to reduced capacitance at the output node and reduced overlap current. The description of how dynamic gate operates is given in details in [168].

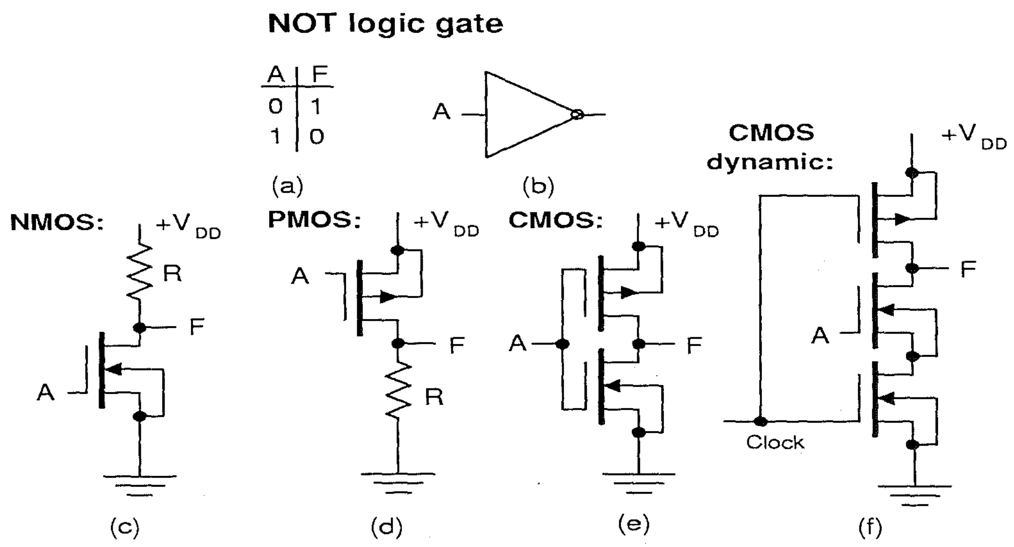
## A.2 Primitive logic gates and their implementation

The *primitive logic gates* are gates that implement primitive logic operations such as AND, OR, NOT, NOR, NAND, etc.. Primitive logic gates are the basic gates used in circuit design. Although a logic gates perform identical logic operations in the different CMOS, PMOS, NMOS versions, the logic levels and other characteristics (speed, power, input current, the number of transistors used, etc.) are quite different. In this dissertation we will consider evolving the logic circuits using such optimisation criteria as the number of primitive logic gates and the number of transistors used in circuit. The explanation of logic circuits discussed below can be found in [128], [127], [169]. The implementations of logic gates using different MOS technologies are illustrated in Fig. A.4 – Fig. A.9.

The number of transistors in MOS circuits can be calculated as follows:

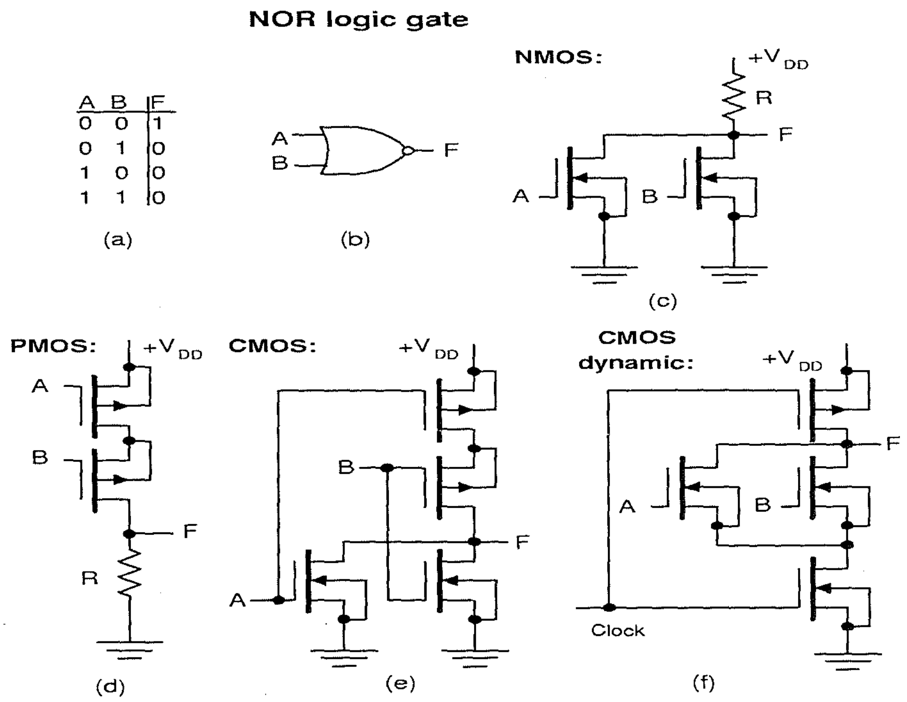
$$N_{FET}^{AND} = N_{FET}^{NAND} + N_{FET}^{NOT}; \quad N_{FET}^{OR} = N_{FET}^{NOR} + N_{FET}^{NOT}; \quad N_{FET}^{EXOR} = N_{FET}^{AND} + N_{FET}^{OR} + N_{FET}^{NAND}. \tag{A.2.1}$$

For example, the CMOS EXOR circuit require 16 transistors, because six transistors



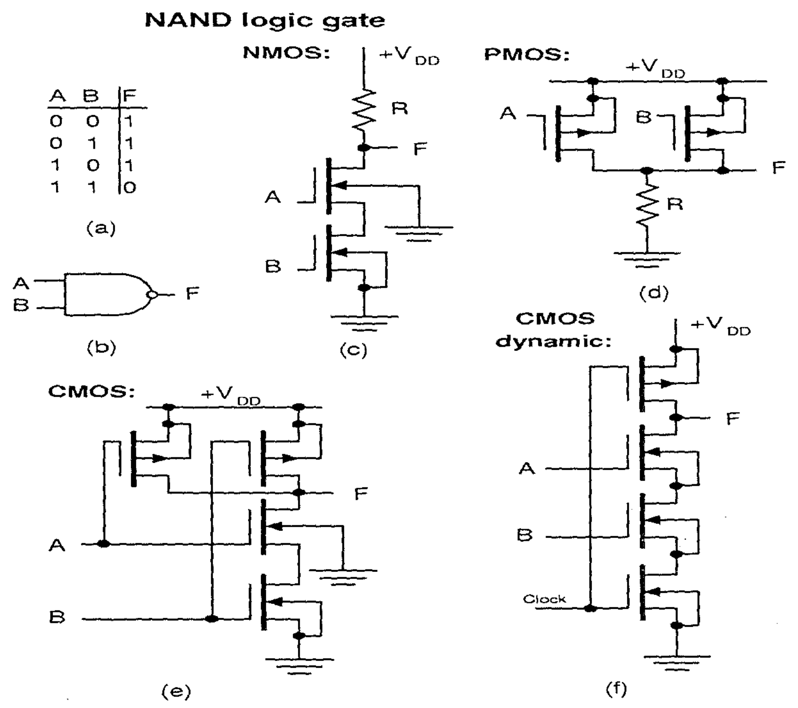
| A | F |
|---|---|
| 0 | 1 |
| 1 | 0 |

Figure A.4: NOT logic gate, $F = \overline{A}$: (a) Truth table, (b) Distinctive-shape symbol, (c) NMOS circuit, (d) PMOS circuit, (e) CMOS circuit, (f) CMOS dynamic circuit.



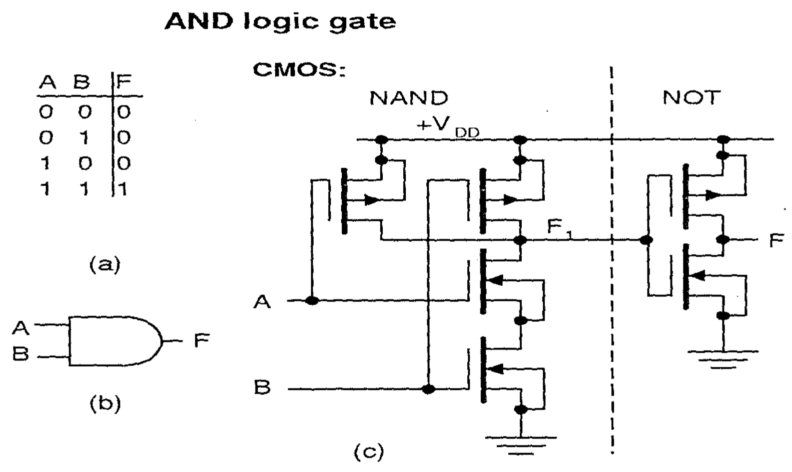
| A | B | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Figure A.5: NOR logic gate, $F = \overline{A \vee B} = \overline{A + B}$: (a) Truth table, (b) Distinctive-shape symbol, (c) NMOS circuit, (d) PMOS circuit, (e) CMOS circuit, (f) CMOS dynamic circuit.



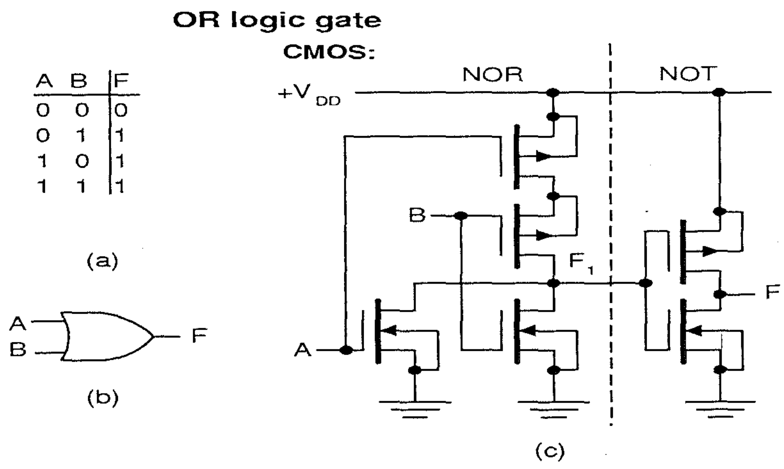
| A | B | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Figure A.6: NAND logic gate, $F = \overline{A \wedge B} = \overline{A \cdot B}$: (a) Truth table, (b) Distinctive-shape symbol, (c) NMOS circuit, (d) PMOS circuit, (e) CMOS circuit, (f) CMOS dynamic circuit.



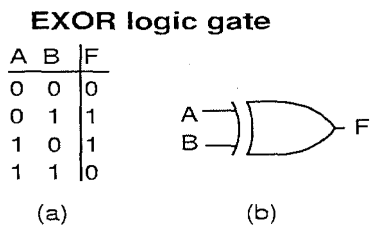
| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Figure A.7: AND logic gate, $F = A \wedge B = A \cdot B$: (a) Truth table, (b) Distinctive-shape symbol, (c) CMOS circuit.



| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Figure A.8: OR logic gate, $F = A \vee B = A + B$: (a) Truth table, (b) Distinctive-shape symbol, (c) CMOS circuit.



| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Figure A.9: EXOR logic gate, $F = A \oplus B$: (a) Truth table, (b) Distinctive-shape symbol.

Table A.1: The number of transistors used in primitive logic circuits.

| **Logic gate** | **Technology** | | | |
|---|---|---|---|---|
| | NMOS | PMOS | CMOS | CMOS dynamic logic |
| NOT | 2 | 2 | 2 | 3 |
| OR | 5 | 5 | 6 | 7 |
| NOR | 3 | 3 | 4 | 4 |
| AND | 5 | 5 | 6 | 7 |
| NAND | 3 | 3 | 4 | 4 |
| EXOR | 13 | 13 | 16 | 18 |

are required to implement any of AND and OR CMOS circuits and four transistors are used to realize CMOS NAND function.

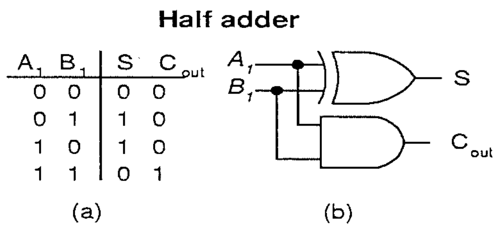
### A.2.1 Comparison of logic gate implementations

There are a lot of criteria by which the performance of circuit can be evaluated. In our work we will estimate the logic gate implementation in terms of the number of transistors used. The number of transistors required to implement primitive logic functions discussed above are given in Table A.1. Note, that the NMOS and PMOS circuits of considered primitive logic functions require the same number of transistors.

## A.3 Combinational building blocks

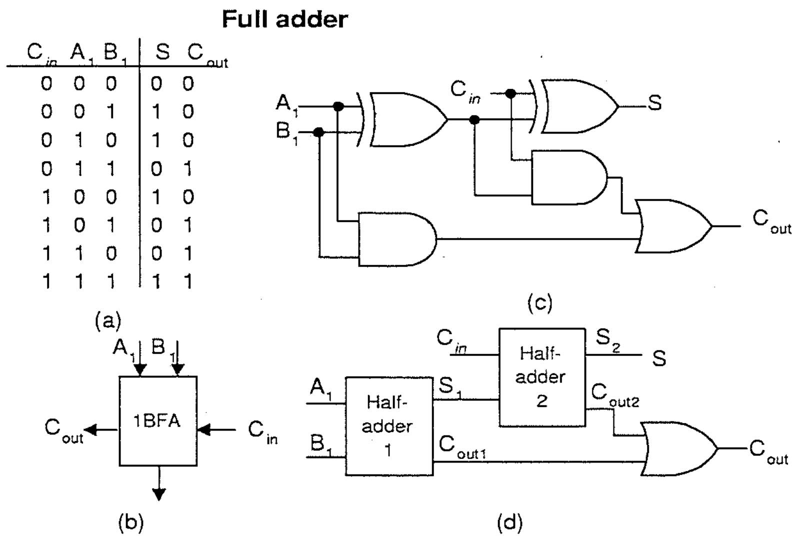
Combinational building blocks are assemblies of gates that implement complex logical operators such as adders, multipliers, encoders, etc.. Complex operators are needed often enough to justify their inclusion in a standard logic family of parts. In this dissertation we will consider the circuit implementation such building blocks as adders and multipliers. Some of adders and multipliers used in this dissertation are shown in Fig. A.10 – Fig. A.12.

A three-bit multiplier (mult3.pla) multiplies the binary numbers $(A_2, A_1, A_0)$ by



| $A_1$ | $B_1$ | S | $C_{out}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Figure A.10: Half-adder: (a) Truth table; (b) Circuit.



| $C_{in}$ | $A_1$ | $B_1$ | S | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Figure A.11: One-bit full adder (1BFA): (a) Truth table; (b) Distinctive-shape symbol; (c) Circuit; (d) Circuit with half-adders.

**Two-bit multiplier**

| $A_1$ | $A_0$ | $B_1$ | $B_0$ | $P_3$ | $P_2$ | $P_1$ | $P_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

(a)

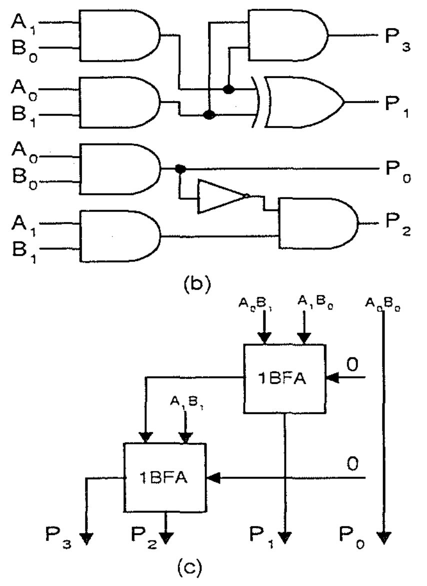


Figure A.12: Two-bit multiplier: (a) Truth table; (b) Conventional circuit; (c) Circuit with the one-bit full-adders (1BFA). In this work the two-bit multiplier is described, by input vector $X = \{x_0, x_1, x_2, x_3\}$ and the output vector $Y = \{y_0, y_1, y_2, y_3\}$, where $x_0 = A_1$, $x_1 = A_0$, $x_2 = B_1$, $x_3 = B_0$, $y_0 = P_3$, $y_1 = P_2$, $y_2 = P_1$ and $y_3 = P_0$.

$(B_2, B_1, B_0)$ to produce the six bit binary number $(y_0, y_1, y_2, y_3, y_4, y_5)$, where $A_2$, $B_2$ and $y_0$ are the most significant bits. In this work the input vector of three-bit multiplier is $X = \{x_0, x_1, x_2, x_3, x_4, x_5\}$, where $x_0 = A_2$, $x_1 = A_1$, $x_2 = A_0$, $x_3 = B_2$, $x_4 = B_1$ and $x_5 = B_0$.

# Appendix B

# Appendix. The functional set of logic gates and EHW performance

There are a number of EHW parameters that can influence the algorithm performance. One of them is the functional set of logic gates involved in evolution. In this section we will consider how the choice of the functional set of logic gates has an effect on EHW performance. The easiest way to define this dependence is to evolve the logic circuits using the gate-level extrinsic EHW with permanent EHW parameters and variable functional set of logic gates. The two-bit adder and the two-bit multiplier are chosen as the tested logic functions. These functions are arithmetic, but the generation principles of functions in question are different. Therefore, the obtained results are valid at least for these two classes of logic functions. A gate-level extrinsic EHW with rudimentary $(1 + \lambda)$ evolutionary strategy and the initial parameters given in Table B.1 is designated to evolve the logic circuits. The circuit layout and connectivity parameters are heterogeneous for both tested functions. The evolutionary algorithm has been executed 100 times for each explored functional set of logic gates. The investigated functional sets of logic gates contain NOT, AND, OR, EXOR logic gates with varieties of inverted and primary inputs and outputs.

Table B.1: Initial data: Investigation the influence the functional set of logic gates used on algorithm performance.

| **Circuit** | add2c.pla | mult2.pla |
|---|---|---|
| **EHW parameters** | | |
| **Circuit layout,** $N_{cols} \times N_{rows}$ | 1x20 | 1x10 |
| **Connectivity parameter,** $N_{connect}$ | 20 | 10 |
| **Gate distribution** | Proportional | |
| **Type of layout** | Heterogeneous | |
| **EA parameters** | | |
| **Type of algorithm** | $(1+\lambda)$ ES | |
| **Population size** | 5 | |
| **Number of generations** | 15000 | 5000 |
| **Number of algorithm runs** | 100 | |
| **Mutation type** | Uniform | |
| **Circuit mutation rate** | 0.05 | |
| **Fitness strategy** | $\mathcal{F}_1 + \mathcal{F}_2$ | |

The encoding of logic gates is given in Table 4.2.

All possible logic gates that can be involved in evolution and their encoding are shown in Table B.2. The functional sets of logic gates are generated as a variation of the one-input logic functions NOT and wire and the two-input logic functions AND, OR, EXOR with primary and inverted inputs. Wire passes the logic value from input to output without changes. This function is labeled as -14. Note that the logic functions encoded -10 and -11 are equal. The same conclusion can be made about the logic functions marked -9 and -12. Nevertheless this fact, these logic functions are encoded separately, since they define the EXOR logic gate with different variations of input variables (primary and inverted). This encoding provides equal opportunities for EXOR logic operator with different types of inputs.

The chosen functional sets of logic gates implicated in the circuit evolution are

Table B.2: The functional characteristics of functional sets. $\mathbb{FS}_i$ is the $i$-th functional set of logic gates; $|\mathbb{FS}_i|$ is cardinality of set $\mathbb{FS}_i$.

| $\mathbb{FS}_i$ | **Functional set** | $\lvert\mathbb{FS}_i\rvert$ | $p_t$ | | | | $p_i$ |
|---|---|---|---|---|---|---|---|
| | | | {**or**} | {**and**} | {**exor**} | {**not**} | {**not**} |
| $\mathbb{FS}_1$ | {-1,-5,-9,-13} | 4 | 0.25 | 0.25 | 0.25 | 0.25 | 0 |
| $\mathbb{FS}_2$ | {-1,-5,-13} | 3 | 0.33 | 0.33 | 0 | 0.33 | 0 |
| $\mathbb{FS}_3$ | {-1,-9,-13} | 3 | 0 | 0.33 | 0.33 | 0.33 | 0 |
| $\mathbb{FS}_4$ | {-5,-9,-13} | 3 | 0.33 | 0 | 0.33 | 0.33 | 0 |
| $\mathbb{FS}_5$ | {-1,-2,-5,-9} | 4 | 0.25 | 0.5 | 0.25 | 0 | 0.125 |
| $\mathbb{FS}_6$ | {-1,-2,-3,-4,-5,-6,-7,-8} | 8 | 0.5 | 0.5 | 0 | 0 | 0.5 |
| $\mathbb{FS}_7$ | {-1,-9,-10,-11,-12} | 5 | 0 | 0.2 | 0.8 | 0 | 0.4 |
| $\mathbb{FS}_8$ | {-1,-9,-10,-11} | 4 | 0 | 0.25 | 0.75 | 0 | 0.25 |
| $\mathbb{FS}_9$ | {-1,-9,-10} | 3 | 0 | 0.33 | 0.66 | 0 | 0.167 |
| $\mathbb{FS}_{10}$ | {-1,-2,-3,-9,-10,-11,-12} | 7 | 0 | 0.42 | 0.56 | 0 | 0.428 |
| $\mathbb{FS}_{11}$ | {-1,-2,-9,-10} | 4 | 0 | 0.5 | 0.5 | 0 | 0.5 |
| $\mathbb{FS}_{12}$ | {-1,-3,-9,-10} | 4 | 0 | 0.5 | 0.5 | 0 | 0.5 |
| $\mathbb{FS}_{13}$ | {-1,-2,-3,-9,-10,-11} | 6 | 0 | 0.5 | 0.5 | 0 | 0.33 |
| $\mathbb{FS}_{14}$ | {-1,-2,-3,-4,-9,-10} | 6 | 0 | 0.66 | 0.33 | 0 | 0.417 |
| $\mathbb{FS}_{15}$ | {-1,-2,-9} | 3 | 0 | 0.66 | 0.33 | 0 | 0.167 |
| $\mathbb{FS}_{16}$ | {-1,-2,-3,-9} | 4 | 0 | 0.75 | 0.25 | 0 | 0.25 |
| $\mathbb{FS}_{17}$ | {-1,-2,-3,-4,-9} | 5 | 0 | 0.8 | 0.2 | 0 | 0.4 |

shown in Table B.2. The functional sets of logic gates have been chosen according to the following characteristics: the percentages of the primitive logic gates AND, OR, EXOR, NOT in the functional set $\mathbb{FS}$, $p_t(\{\textbf{and}\})$, $p_t(\{\textbf{or}\})$, $p_t(\{\textbf{exor}\})$, $p_t(\{\textbf{not}\})$ and the percentage of the inverted inputs in the functional set $\mathbb{FS}$, $p_i(\{\textbf{not}\})$. Let us consider a computational process of the percentages mentioned above using the following example. Let the functional set of logic gates be $\mathbb{FS}_6 = \{x_1 \wedge x_2, x_1 \wedge \overline{x_2}, x_1 \vee x_2, x_1 \oplus x_2\} = \{-1, -2, -5, -9\}$. There are 3 types of primitive logic gates and 4 unique logic gates in considered functional set. -1 and -2 encode the AND logic gate with primary and inverted second input. -5 and -9 interpret OR and EXOR logic gates with primitive logic gates respectively. Therefore, $p_t(\{\textbf{and}\}) = 2/4 = 0.5$,

$p_t(\{\textbf{or}\}) = 1/4 = 0.25$ and $p_t(\{\textbf{exor}\}) = 1/4 = 0.25$. There is no logic gate NOT in the functional set $\mathbb{FS}_6$. Hence, $p_t(\{\textbf{not}\}) = 0$. All logic functions in $\mathbb{FS}_6$ have 2 inputs. There is only one logic function labeled -2 that contains an inverted input. Therefore, $p_i(\{\textbf{not}\}) = 1/8 = 0.125$. $p_t$ and $p_i$ describe the functional characteristics of the functional set $\mathbb{FS}$. These characteristics are calculated for each examined functional set of logic gates $\mathbb{FS}_i$ and are summarised in Table B.2.

The functional sets are generated according to their functional characteristics. Thus, all primitive two-input logic gates with primitive inputs generate $\mathbb{FS}_1$. Therefore, $p_t(\{\textbf{and}\}) = p_t(\{\textbf{or}\}) = p_t(\{\textbf{exor}\}) = p_t(\{\textbf{not}\}) = 0.25$. and $p_i(\{\textbf{not}\}) = 0$. The functional sets $\mathbb{FS}_2$-$\mathbb{FS}_4$ contain no logic functions with inverted inputs. The functional sets $\mathbb{FS}_6$-$\mathbb{FS}_17$ include no logic gates OR and NOT. The percentage of inverted inputs in the functional sets in question is variable. Also, the percentages of AND and EXOR logic gates in $\mathbb{FS}_6$-$\mathbb{FS}_17$ are chameleonic. The process generated the functional sets of logic gates and described above allow us to define the optimal proportion of primitive logic gates in inverted inputs for given tested logic function. In other words the usefulness of using different sub-functions can be derived by analysing the performances of algorithms with different functional sets. At the same time the importance of using the inverted inputs in the functional set of logic gates can be expounded.

The experimental data for experiments described above are summarised in Table B.3. Let us consider how the functional sets of primitive logic gates with equal $p_t$ influence the algorithm performance. It is interesting to note that the fully functional two-bit multipliers have been evolved using the functional sets containing at least AND and EXOR logic gates. This shows that for the specific circuit layout chosen

Table B.3: Experimental results: Investigation the influence of the functional set of logic gates used on $(1+\lambda)$ ES performance. $\mathbb{FS}_i$ is the $i$-th functional set of logic gates; $\overline{F_1^{bf}}$, $\overline{F_2^{bf}}$ and $\overline{F_3^{bf}}$ are the mean fitnesses $F_1$, $F_2$ and $F_3$ of the best evolved chromosomes respectively; $\overline{F_2(\mathcal{N}_f)}$ is the mean fitness function $F_2$ of fully functional designs evolved during 100 runs; R$(\mathcal{N}_f)$ is the number of evolved fully functional circuits, $\mathcal{N}_f$

| $\mathbb{FS}_i$ | add2c.pla | | | | mult2.pla | | | |
|---|---|---|---|---|---|---|---|---|
| | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}$ | $\overline{F_2^{bf}(\mathcal{N}_f)}$ | R$(\mathcal{N}_f)$ | $\overline{F_1^{bf}}$ | $\overline{F_2^{bf}}$ | $\overline{F_2^{bf}(\mathcal{N}_f)}$ | R$(\mathcal{N}_f)$ |
| $\mathbb{FS}_1$ | 96.177 | 11.39 | 11.125 | 32 | 98.9531 | 7.3 | 7.1224 | 49 |
| $\mathbb{FS}_2$ | 84.4375 | 13.29 | 0 | 0 | 96.75 | 7.66 | 0 | 0 |
| $\mathbb{FS}_3$ | 90.2291 | 10.41 | 11.57 | 7 | 99.0937 | 7.38 | 7.0667 | 60 |
| $\mathbb{FS}_4$ | 88.6458 | 10.15 | 0 | 3 | 83.95 | 6.61 | 0 | 0 |
| $\mathbb{FS}_5$ | 97.6667 | 11.92 | 11.21 | 55 | 99.0468 | 7.44 | 7.1818 | 66 |
| $\mathbb{FS}_6$ | 86.2187 | 26.25 | 0 | 0 | 96.5312 | 7.68 | 9.0 | 2 |
| $\mathbb{FS}_7$ | 89.3541 | 20.18 | 17 | 2 | 97.968 | 7.36 | 7.34 | 38 |
| $\mathbb{FS}_8$ | 90.7282 | 13.98 | 14.4 | 5 | 98.0156 | 7.2 | 7.107 | 28 |
| $\mathbb{FS}_9$ | 93.9479 | 20.01 | 15.25 | 20 | 98.5625 | 7.22 | 7.1961 | 51 |
| $\mathbb{FS}_{10}$ | 94.9687 | 20.73 | 16.9333 | 30 | 97.6094 | 7.86 | 7.75 | 36 |
| $\mathbb{FS}_{11}$ | 95.1771 | 14.79 | 13.8485 | 33 | 98.65 | 7.33 | 7.25 | 58 |
| $\mathbb{FS}_{12}$ | 95.2291 | 15.52 | 13.73 | 30 | 98.46875 | 7.4 | 7.19 | 52 |
| $\mathbb{FS}_{13}$ | 96.236 | 15.7 | 14.02 | 37 | 98.14 | 7.74 | 7.39 | 43 |
| $\mathbb{FS}_{14}$ | 96.024 | 14.93 | 13.9 | 36 | 98.15 | 7.72 | 7.38 | 44 |
| $\mathbb{FS}_{15}$ | 97.32 | 13.74 | 13.6 | 45 | 99.1875 | 7.3 | 7.1867 | 75 |
| $\mathbb{FS}_{16}$ | 98.43 | 14.05 | 13.97 | 33 | 98.89 | 7.36 | 7.29 | 68 |
| $\mathbb{FS}_{17}$ | 97.034 | 15.78 | 13.67 | 39 | 98.73 | 7.53 | 7.23 | 59 |

the functional set of logic gates has to contain these two primitive logic gates. At the same time the performance of algorithm with functional set of AND, EXOR and NOT logic gates is better then with additional OR logic gate. These experimental results clearly show that the functional set containing *all* primitive logic gates does not provide the better algorithm performance. Note that the mean number of active logic gates in evolved fully functional two-bit multipliers $\overline{F_2^{bf}(\mathcal{N}_f)}$ is less when $\mathbb{FS}_3$ has been applied and higher for $\mathbb{FS}_1$ Analysing the quality of evolved fully functional circuits we can conclude that the smaller circuits have been evolved when the algorithm with $\mathbb{FS}_3$ has been executed.

Note that the fully functional two-bit multiplier can be evolved using AND and OR logic gates only. In this case the functional set has to contain the primitive logic gates with inverted inputs. This can be demonstrated by experimental results obtained for $\mathbb{FS}_6$. The proportions of AND and OR gates in this functional set are equal. The functional set contains the logic gates with primary and inverted inputs. In this case 2 two-bit multipliers have been evolved. Each of them contains 9 primitive logic gates. Functional sets $\mathbb{FS}_6$ and $\mathbb{FS}_2$ are similar since both of them contain AND and OR logic gates and EXOR logic gate. It can be seen that the mean functionality criteria $\overline{F_1^{bf}}$ obtained after evolution with $\mathbb{FS}_6$ is lower then after evolution with $\mathbb{FS}_2$. The same behaviour implies to the $\overline{F_3^{bf}}$ criteria. This shows that the final circuits evolved using $\mathbb{FS}_6$ contain less correct output bits then the similar circuits evolved using $\mathbb{FS}_2$. This means that the higher correctness of circuits does not always quarantee that the fully functional circuit will be evolved.

Next, let us consider the performance of algorithms executed with $\mathbb{FS}_7$-$\mathbb{FS}_{17}$ functional sets. The ideas of this experiment is to define how using different proportions of AND and EXOR logic gates influence on algorithm performance. Also, analysis of this experimental data shows the importance of inverted inputs in the functional set of logic gates.

Let us take up how the number of fully functional designs evolved changes with varying $p_t(\{\mathbf{and}\})$, $p_t(\{\mathbf{exor}\})$ and $p_i(\{\mathbf{not}\})$. The functional sets $\mathbb{FS}_7$-$\mathbb{FS}_{17}$ are allocated by ascending order of $p_t(\{\mathbf{and}\})$. Analysing the obtained experimental data we can conclude that the algorithm performs better if the number of logic gates dominates in the functional set. The number of fully functional two-bit multipliers evolved magnifies with increasing $p_t(\{\mathbf{and}\})$.

In this section we considered how using different structures of the functional set of logic gates influence the algorithm performance The choice of the functional set of logic gates is a very complicated task and is ultimately linked with the complexity of the logic functions involved in the functional set. Each functional set has been considered in terms of the number of primitive logic gates involved and the number of inverted inputs used with the primitive logic gates. The experimental results shows that there is an *optimal* functional set of logic gates that provides the better algorithm performance then others. Also, it has been shown that the choice of the functional set of logic gates drastically influence the algorithm performance. The wrong choice of the functional set of logic gates can lead to the situation where the fully functional designs would not be evolved even if the others EHW parameters are chosen correctly.

# Appendix C

# Appendix. Evolved optimal two-bit multiplier designs

In this section we will consider a number of evolved optimal two-bit multiplier designs. All these designs require 7 primitive logic gates and have been evolved using different functional set of logic gates. In order to define the optimal design of logic circuits evolved using a function-level EHW, the optimal design of building blocks has to be considered. The two-bit multiplier has been used as a building block to evolve a three-bit multiplier. The Table C.1 shows the number of logic gates required to implement only several outputs of the two-bit multiplier. These data has been used to define the optimal size of evolved three-bit multiplier at function level.

These circuits have been evolved using a gate-level EHW proposed by Miller [25].

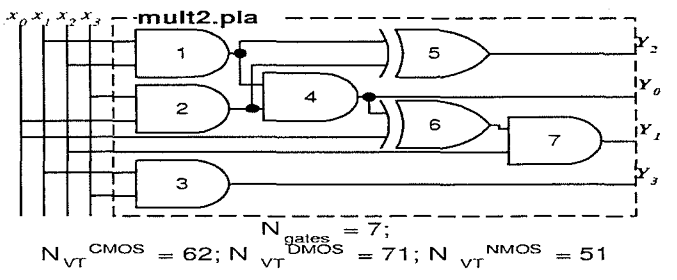


Figure C.1: Evolved two-bit multiplier design (A)

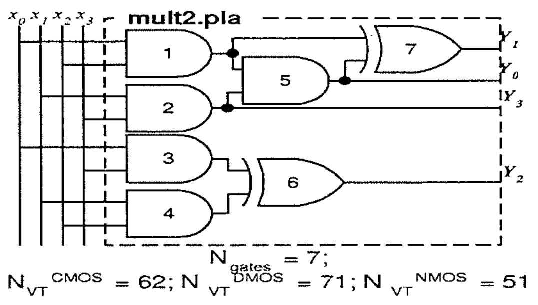


Figure C.2: Evolved two-bit multiplier design (B)

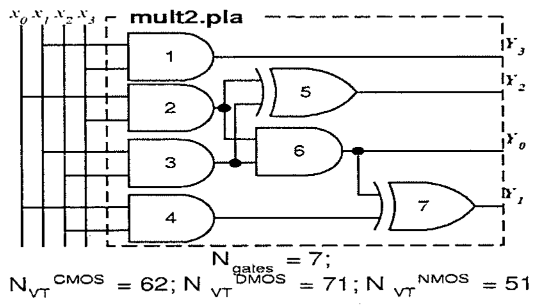


Figure C.3: Evolved two-bit multiplier design (C)

Table C.1: The number of primitive logic gates required to implement different combinations of the two-bit multiplier outputs.

| Output | mult2.pla | | |
|---|---|---|---|
| combinations | A | B | C |
| $Y_0$ | 3 | 3 | 3 |
| $Y_1$ | 5 | 4 | 4 |
| $Y_2$ | 3 | 3 | 3 |
| $Y_3$ | 1 | 1 | 1 |
| $Y_0$, $Y_1$ | 5 | 4 | 5 |
| $Y_0$, $Y_2$ | 4 | 6 | 4 |
| $Y_0$, $Y_3$ | 4 | 3 | 4 |
| $Y_1$, $Y_2$ | 6 | 7 | 6 |
| $Y_1$, $Y_3$ | 6 | 4 | 6 |
| $Y_2$, $Y_3$ | 4 | 4 | 4 |
| $Y_0$, $Y_1$, $Y_2$ | 6 | 7 | 6 |
| $Y_0$, $Y_1$, $Y_3$ | 6 | 4 | 6 |
| $Y_0$, $Y_2$, $Y_3$ | 5 | 6 | 5 |
| $Y_1$, $Y_2$, $Y_3$ | 7 | 7 | 7 |

Three of the evolved circuits with 7 2-input logic gates are shown in Fig. C.1 – C.3. It is necessary to note that the best designed two-bit multiplier requires at least 8 2-input logic gates. Hence, the designs shown in Fig. C.1 – C.3 are optimal in terms of the number of 2-input logic gates used. Examining these circuits instantly reveals their strangeness. Note that in conventional model of multiplication only output $Y_3$ is re-used (Fig. A.12). In all evolved designs the outputs $Y_0$ and $Y_1$ are used to produce $Y_1$. The circuit for $Y_3$ in all four evolved circuits is effectively the same. In circuits A and C the output $Y_3$ is not re-used and in circuit B the output $Y_2$ is implemented independently. In conventional circuits the output $Y_2$ has never been implemented separately. The most evolved two-bit multipliers with 7 2-input logic gates have one of the circuit structures discussed above.

# Appendix D

# Computational effort of EHW approach

The experimental results reported in this dissertation have been performed using Pentium Pro II with 400 MHz frequency. The computational effort required to evolve a circuit depends on the circuit layout used, type of fitness function implemented (parallel or consistent), the memory allocation (if the circuit layout evolution is used), the size of truth table analysed. So, the empirical study has been carried out to define the computational cost required to evolve logic function by various implementations of EHW. Table D.1 summarises these results.

Parallel fitness function is calculated according to specific features of C++ programming language. The details of this can be found in [115], [170].

Analysing experimental results one can notice that the evolutionary algorithm with parallel fitness function performs faster than with consistent fitness function.

Larger function requires more computational time to be evolved. For example, in order to evolve a three-bit multiplier using 1x50 circuit layout, parallel dynamic fitness function and evolutionary strategy that performs during 200 000 generations requires a 1 hour, 20 min, 03 sec, 420 msec.

Table D.1: Computational effort of EHW approach during one run to evolve the two-bit multiplier. GGM, BGM and LGM are the global, boundary and local geometry mutations respectively.

**EHW parameters**

| | | | |
|---|---|---|---|
| **Circuit layout,** $N_{cols} \times N_{rows}$ | 1x10 | 1x10 | 1x10 |
| **Connectivity parameter,** $N_{connect}$ | 10 | 10 | 10 |
| **Gate distribution** | Proportional | Proportional | Proportional |
| **Type of layout** | Homogeneous | Homogeneous | Heterogeneous |

**EA parameters**

| | | | | | | |
|---|---|---|---|---|---|---|
| **Type of algorithm** | $(1+\lambda)$ ES | $(1+\lambda)$ ES | | GA | | |
| **Population size** | 4 | 5 | | 5 | | |
| **Number of generations** | 5000 | 5000 | | 5000 | | |
| **Crossover type** | - | - | | Cell Uniform | | |
| **Mutation type** | Uniform | Uniform | | Uniform | | |
| **Geometry Mutation type** | - | - | | GGM | BGM | LGM |
| **Circuit mutation rate** | 0.05 | 0.05 | | 0.05 | | |
| **Fitness strategy** | $\mathcal{F}_1 + \mathcal{F}_2$ | $\mathcal{F}$ | $\mathcal{F}_1 + \mathcal{F}_2$ | $\mathcal{F}_1 + \mathcal{F}_2$ | | |
| **Fitness implementation** | Consistent | Parallel | | Parallel | | |
| **Computational effort** min : sec : msec | 2:20:720 | 26:150 | 34:660 | 10:660 | 23:900 | 22:360 |

# Appendix E

# Appendix. Distinctive features of the TPRODUCT operator

Let $a$, $b$ and $c$ be the $r$-valued logic variables. Let "x", "$\diamondsuit$", "$\vee$" and "$\cdot$" be the TPRODUCT, TSUM, MAX and MIN $r$-valued logic operations respectively. The following operations hold for multi-valued algebra:

**Proposition E.0.1.** *Associative law:*

$$(a \times b) \times c = a \times (b \times c) \tag{E.0.1}$$

*Proof.*

$$\begin{aligned}(a \times b) \times c = \mathrm{MAX}(a + b - (r-1), 0) \times c = \\ \mathrm{MAX}(a + b + c - 2(r-1), 0 + c - (r-1), 0) = \\ \mathrm{MAX}(a + b + c - 2(r-1), 0),\end{aligned} \tag{E.0.2}$$

because $c - (r-1) \leq 0$ for all values of $c$.

$$\begin{aligned}a \times (b \times c) = a \times \mathrm{MAX}(b + c - (r-1), 0) = \\ \mathrm{MAX}(a + b + c - 2(r-1), 0 + a - (r-1), 0)) = \\ \mathrm{MAX}(a + b + c - 2(r-1), 0),\end{aligned} \tag{E.0.3}$$

because $a - (r-1) \leq 0$ for all values of $a$.

Since expressions Eq. E.0.2 and Eq. E.0.3 are equal, then the TPRODUCT operator is associative. □

**Proposition E.0.2.** *Commutative law:*

$$a \times b = b \times a \tag{E.0.4}$$

*Proof.* This statement is proved using the definition of TPRODUCT operator. □

**Proposition E.0.3.** *Identity:*

$$(r-1) \times a = a \tag{E.0.5}$$

*Proof.*

$$(r-1) \times a = \text{MAX}(r-1+a-(r-1), 0) = \text{MAX}(a, 0) = a, \tag{E.0.6}$$

because $a \in \{0, 1, \cdots, r-1\}$. □

**Proposition E.0.4.** *4. Identity:*

$$0 \times a = 0$$

*Proof.*

$$0 \times a = \text{MAX}(0+a-(r-1), 0) = 0, \tag{E.0.7}$$

because $a \in \{0, 1, \cdots, r-1\}$, we have $a-(r-1) \leq 0$ □

**Proposition E.0.5.** *5. Identity:*

$$a_0 \times a_1 \times \cdots \times a_{r-1} = 0 \tag{E.0.8}$$

*Proof.* This expression describes all input combinations of $a$, therefore by definition of literal function Eq. 4 takes value $(r-1)$. By definition of TPRODUCT operator we have:

$$a_0 \times a_1 \times \cdots \times a_{r-1} = \text{MAX}(r-1-(r-1), 0) = 0.$$

□

**Proposition E.0.6.** *6. Identity:*

$$1 \times a_1 \vee 2 \times a_2 \vee \cdots \vee (r-1) \times a_{r-1} = a \tag{E.0.9}$$

*Proof.* If $a = 1$, then $1 \times x^1 \vee 2 \times x^2 \vee \cdots \vee (r-1) \times x^(r-1) = (MAX)(1+r-1-(r-1), 0) \vee 0 \vee \cdots \vee 0 = 1$;

If $a = 2$, then $1 \times x^1 \vee 2 \times x^2 \vee \cdots \vee (r-1) \times x^(r-1) = (MAX)(2+r-1-(r-1), 0) \vee 0 \vee \cdots \vee 0 = 2$;

...

If $a = r-1$, then $1 \times x^1 \vee 2 \times x^2 \vee \cdots \vee (r-1) \times x^(r-1) = (MAX)(r-1+r-1-(r-1), 0) \vee 0 \vee \cdots \vee 0 = r-1$;

According to Eq. E.0.4: $a \times x^0 = 0$. □

**Proposition E.0.7.** *7. Identity:*

$$(1 \times a_1)\Diamond(2 \times a_2)\Diamond \cdots \Diamond((r-1) \times a_{r-1}) = a \tag{E.0.10}$$

*Proof.* If $a = 1$, then $(1 \times a_1)\Diamond(2 \times a_2)\Diamond \cdots \Diamond((r-1) \times a_{r-1}) = (MAX)(1 + r - 1 - (r-1), 0)\Diamond 0 \Diamond \cdots \Diamond 0 = \text{MIN}(1, r-1) = 1$;

If $a = 2$, then $(1 \times a_1)\Diamond(2 \times a_2)\Diamond \cdots \Diamond((r-1) \times a_{r-1}) = (MAX)(2 + r - 1 - (r-1), 0)\Diamond 0 \Diamond \cdots \Diamond 0 = \text{MIN}(2, r-1) = 2$;

...

If $a = r-1$, then $(1 \times a_1)\Diamond(2 \times a_2)\Diamond \cdots \Diamond((r-1) \times a_{r-1}) = (MAX)(r - 1 + r - 1 - (r-1), 0)\Diamond 0 \Diamond \cdots \Diamond 0 = \text{MIN}(r-1, r-1) = r-1$;

According to Eq. E.0.4: $a \times x^0 = 0$. □

**Proposition E.0.8.** *8. Distributive law:*

$$(a \times b) \vee c =\neq (a \vee c) \times (b \vee c) \tag{E.0.11}$$

*Proof.*

$$\begin{aligned}(a \times b) \vee c &= \text{MAX}(a + b - (r-1), 0) \vee c \\ &= \text{MAX}(\text{MAX}(a + b - (r-1), 0), c) \\ = \text{MAX}(a + b - (r-1), c, 0) &= \text{MAX}(a + b - (r-1), c),\end{aligned} \tag{E.0.12}$$

because $c \in \{0, 1, \cdots, r-1\}$;

$$\begin{aligned}(a \vee c) \times (b \vee c) &= \text{MAX}(a, c) \times \text{MAX}(b, c) \\ &= \text{MAX}(\text{MAX}(a, z) + \text{MAX}(b, c) - (r-1), 0) \\ = \text{MAX}(a + b - (r-1), a + c - (r-1), &b + c - (r-1), 2c - (r-1), 0)\end{aligned} \tag{E.0.13}$$

Since expressions Eq. E.0.12 and Eq. E.0.13 are not equal, then the TPRODUCT operator does not obey the distributive law. □

**Proposition E.0.9.** *Distributive law:*

$$(a \times b)\Diamond c \neq (a\Diamond c) \times (b\Diamond c) \tag{E.0.14}$$

*Proof.*

$$\begin{aligned}(a \times b)\Diamond c &= \text{MAX}(a + b - (r-1), 0)\Diamond c \\ = \text{MIN}(\text{MAX}(a + b &- (r-1), 0) + c, r-1);\end{aligned} \tag{E.0.15}$$

$$\tag{E.0.16}$$

if $a+c \geq r-1$, then $(a \times b)\diamondsuit c = \text{MIN}(a+b+c-(r-1), r-1)$;
if $a+c < r-1$, then $(a \times b)\diamondsuit c = \text{MIN}(z, r-1)$;

$$(a\diamondsuit c) \times (b\diamondsuit c) = \text{MAX}(a+c, r-1) \times \text{MAX}(b+c, r-1) = \text{MIN}(\text{MAX}(a+c, r-1) + \text{MAX}(b+c, r-1), r-1); \quad \text{(E.0.17)}$$

if $a+c \geq r-1; b+c \geq r-1$, then $((a\diamondsuit c) \times (b\diamondsuit c) = \text{MIN}(a+c+b+c, r-1) = \text{MIN}(a+2c+b, r-1)$;
if $a+c \geq r-1; b+c < r-1$, then $((a\diamondsuit c) \times (b\diamondsuit c) = \text{MIN}(a+c+r-1, r-1) = r-1$;
if $a+c < r-1; b+c \geq r-1$, then $((a\diamondsuit c) \times (b\diamondsuit c) = \text{MIN}(r-1+b+c, r-1) = r-1$;
if $a+c < r-1; b+c < r-1$, then $((a\diamondsuit c) \times (b\diamondsuit c) = \text{MIN}(r-1+r-1, r-1) = r-1$;
Because the expressions Eq. E.0.15 and Eq. E.0.17 are not equal, TPRODUCT operator does not obey the distributive law. □

**Proposition E.0.10.** *Distributive law:*

$$(a \times b) \vee c^i = (a \vee c^i) \times (b \vee c^i), \quad \text{(E.0.18)}$$

*where* $c^i \in \{0, r-1\}$.

*Proof.*

$$\begin{aligned}(a \times b) \vee c^i &= \text{MAX}(a+b-(r-1), 0) \vee c^i \\ &= \text{MAX}(\text{MAX}(a+b-(r-1), 0), c^i) \\ &= \text{MAX}(a+b-(r-1), c^i, 0)\end{aligned} \quad \text{(E.0.19)}$$

$$\begin{aligned}(a \vee c^i) \times (b \vee c^i) &= \text{MAX}(a, c^i) \times \text{MAX}(b, c^i) \\ &= \text{MAX}(\text{MAX}(a, c^i) + \text{MAX}(b, c^i) - (r-1), 0) \\ &= \text{MAX}(a+c^i-(r-1), a+b-(r-1), \\ &\quad c^i+b-(r-1), 2c^i-(r-1), 0).\end{aligned} \quad \text{(E.0.20)}$$

Let us consider the following conditions, taking into account that $c^i \in \{0, r-1\}$.
If $c^i = 0$, then $a+c^i-(r-1) = a-(r-1)$.
If $c^i = r-1$, then $a+c^i-(r-1) = a$.
Note that if $c^i = r-1$, then $a-(r-1) \leq 0$. Hence, in any case Eq. E.0.20 is equal (r-1), because $2c^i-(r-1) = 2(r-1)-(r-1)-r-1$. Therefore, term $a+c^i-(r-1)$ is not essential.

The expression $b+c^i-(r-1)$ is analysed by analogy with the previous expression. Hence, we have:

$$\begin{aligned}&\text{MAX}(a+c^i-(r-1), a+b-(r-1), c^i+b-(r-1), 0) \\ &\quad = \text{MAX}(a+b-(r-1), 2c^i-(r-1), 0).\end{aligned} \quad \text{(E.0.21)}$$

The TPRODUCT operator obeys the distributive law, since the expressions Eq. E.0.19 and Eq. E.0.21 are equal. □

**Proposition E.0.11.** *Distributive law*

$$(a \vee b) \times c = (a \times c) \vee (b \times c). \tag{E.0.22}$$

*Proof.*

$$\begin{aligned}(a \vee b) \times c = \mathrm{MAX}(a, b) \times c &= \mathrm{MAX}(a, b) + c - (r - 1), 0)) \\ &= \mathrm{MAX}(a + c - (r - 1), b + c - (r - 1), 0).\end{aligned} \tag{E.0.23}$$

$$\begin{aligned}(a \times c) \vee (b \times c) = \mathrm{MAX}(a + c(r - 1), 0) \vee \mathrm{MAX}(b + c - (r - 1), 0) = \\ \mathrm{MAX}(\mathrm{MAX}(a + c - (r - 1), 0), \mathrm{MAX}(b + c - (r - 1), 0)) \\ = \mathrm{MAX}(a + c - (r - 1), b + c - (r - 1), 0).\end{aligned} \tag{E.0.24}$$

The TPRODUCT operator obeys distributive law, since expressions Eq. E.0.23 and Eq. E.0.24defining the right and left parts of identity are equal. □

**Proposition E.0.12.** *Distributive law*

$$(a \Diamond b) \times c \neq (a \times c) \Diamond (b \times c). \tag{E.0.25}$$

*Proof.*

$$\begin{aligned}(a \Diamond b) \times c = \mathrm{MIN}(a + b, r - 1) \times c \\ = \mathrm{MAX}(\mathrm{MIN}((a + b, r - 1_{+} c - (r - 1), 0);\end{aligned} \tag{E.0.26}$$

(E.0.27)

if $a + b \leq r - 1$, then $(a \Diamond b) \times c = \mathrm{MAX}(a + b + c - (r - 1), 0)$;
if $a + b > r - 1$, then $(a \Diamond b) \times c = \mathrm{MAX}(r - 1 + c - (r - 1), 0) = c$;

$$\begin{aligned}(a \times c) \Diamond (b \times c) = \mathrm{MAX}(a + c - (r - 1), 0) \Diamond \mathrm{MAX}(b + c - (r - 1), 0) \\ = \mathrm{MIN}(\mathrm{MAX}(a + c - (r - 1), 0) + \mathrm{MAX}(b + c - (r - 1), 0), r - 1).\end{aligned} \tag{E.0.28}$$

Because expressions Eq. E.0.26 and Eq. E.0.28 defining the right and left parts of the identity are not equal, the TPRODUCT operator does not obey distributive law. □

**Proposition E.0.13.** *Distributive law:*

$$(a \vee b) \times c^i = (a \times c^i) \vee (b \times c^i). \tag{E.0.29}$$

*Proof.*

$$\begin{aligned}(a \vee b) \times c^i = \mathrm{MAX}(a, b) \times c^i =\\ \mathrm{MAX}(\mathrm{MAX}(a, b) + c^i - (r-1), 0) =\\ \mathrm{MAX}(a + c^i - (r-1), b + c^i - (r-1), 0).\end{aligned} \tag{E.0.30}$$

$$\begin{aligned}(a \times c^i) \vee (b \times c^i)\\ = \mathrm{MAX}(a + c^i - (r-1), 0) \vee \mathrm{MAX}((b + c^i - (r-1), 0)\\ = \mathrm{MAX}(\mathrm{MAX}(a + c^i - (r-1), 0), \mathrm{MAX}((b + c^i - (r-1), 0))\\ = \mathrm{MAX}(a + c^i - (r-1), b + c^i - (r-1), 0).\end{aligned} \tag{E.0.31}$$

The TPRODUCT operator obeys the distributive law, since the expressions Eq. E.0.30 and E.0.31 are equivalent. □

**Proposition E.0.14.** *Distributive law:*

$$(a \Diamond b) \times c^i = (a \times c^i) \Diamond (b \times c^i). \tag{E.0.32}$$

*Proof.*

$$\begin{aligned}(a \Diamond b) \times c^i = \mathrm{MIN}(a + b, r-1) \times c^i\\ = \mathrm{MAX}(\mathrm{MIN}(a + b, r-1) + c^i - (r-1), 0);\end{aligned} \tag{E.0.33}$$

if $a + b > r - 1$, then $(a \Diamond b) \times c^i = \mathrm{MAX}(\mathrm{MIN}(a + b, r-1) + r - 1 - (r-1), 0) = \mathrm{MAX}(\mathrm{MIN}(a + b, r-1), 0) = \mathrm{MIN}(a + b, r-1)$;

if $c^i = 0$, and $a + b \leq r - 1$, then $(a \Diamond b) \times c^i = \mathrm{MAX}(\mathrm{MIN}(a+b, r-1) - (r-1), 0) = \mathrm{MAX}(\mathrm{MIN}(a + b - (r-1), 0) = 0$;

if $c^i = 0$, and $a + b > r - 1$, then $(a \Diamond b) \times c^i = \mathrm{MAX}(\mathrm{MIN}(a+b, r-1) - (r-1), 0) = \mathrm{MAX}(r - 1 - (r-1), 0) = 0$;

$$\begin{aligned}(a \times c^i) \Diamond (b \times c^i) = \mathrm{MAX}(a + c^i - (r-1), 0) \Diamond \mathrm{MAX}(b + c^i - (r-1), 0)\\ = \mathrm{MIN}(\mathrm{MAX}(a + c^i - (r-1), 0) + \mathrm{MAX}(b + c^i - (r-1), 0), r-1);\end{aligned} \tag{E.0.34}$$

if $c^i = r - 1$, then $(a \times c^i) \Diamond (b \times c^i) = \mathrm{MIN}(a + b, r-1)$;

if $c^i = 0$, then $(a \times c^i) \Diamond (b \times c^i) = \mathrm{MIN}(0, r-1) = 0$;

Note that expression $\mathrm{MAX}(a + c^i - (r-1), 0)$ is equal to $am$ if $c^i = r - 1$ and is equal to 0, if $c^i = 0$.

The TPRODUCT operator obeys the de Morgan law, since the expressions Eq. E.0.33 and Eq. E.0.34 are equal. □

**Proposition E.0.15.** *De Morgan law:*

$$\bar{a}\Diamond\bar{b} = \overline{a \times b}. \tag{E.0.35}$$

*Proof.*

$$\bar{a}\Diamond\bar{b} = \mathrm{MIN}(2r - 2a - b, r - 1); \tag{E.0.36}$$

$$\overline{a \times b} = \overline{\mathrm{MIN}(a + y - (r-1), 0)}$$
$$= \mathrm{MIN}(r - 1 - a - b + r - 1, r - 1 - 0) = \mathrm{MIN}(2r - 1 - a - b, r - 1). \tag{E.0.37}$$

Because the equations Eq. E.0.36 and Eq. E.0.37 defining the left and right side of expression Eq. E.0.35 are equal, the statement is correct. □

**Proposition E.0.16.** *16. De Morgan law:*

$$\bar{a} \times \bar{b} = \overline{a\Diamond b}. \tag{E.0.38}$$

*Proof.*

$$\bar{a} \times \bar{b} = \mathrm{MAX}(r - 1 - a + r - 1 - b - r + 1, 0) = \mathrm{MAX}(r - 1 - a - b, 0). \tag{E.0.39}$$

$$\overline{a\Diamond b} = \overline{\mathrm{MIN}(a + b, 0)} = \mathrm{MAX}(r - 1 - a + r - 1 - b - r + 1, 0) = \mathrm{MIN}(r - 1 - a - b, 0). \tag{E.0.40}$$

The statement is correct, because the expressions Eq. E.0.39 and Eq. E.0.40 defining the right and left side of statement are equal. □

# Appendix F

# Appendix. Distinctive features of the TSUM operator

Let $a$, $b$ and $c$ be the $r$-valued logic variables. Let "$\diamondsuit$" and "$\cdot$" be the TSUM and MIN $r$-valued logic operations respectively. The following operations hold for multi-valued algebra:

1. Associative law: $(a\diamondsuit b)\diamondsuit c = a\diamondsuit(b\diamondsuit c)$

2. Commutative law: $a\diamondsuit b = b\diamondsuit a$

3. Identities:

   3A. $(r-1)\diamondsuit a = r-1$

   3B. $0\diamondsuit a = a$

   3C. $x^0\diamondsuit x^1\diamondsuit\cdots\diamondsuit x^{r-1} = r-1$

   3D. $1\cdot a^1\diamondsuit 2\cdot a^2\diamondsuit\cdots\diamondsuit(r-1)\cdot a^{r-1} = a$

4. Distributive laws:

   4A. $(a\cdot b)\diamondsuit c = (a\diamondsuit c)\cdot(b\diamondsuit c)$

4B. $(a \diamondsuit b) \cdot c \neq (a \cdot c) \diamondsuit (b \cdot c)$

4C. $(a \diamondsuit b) \cdot c^i = (a \cdot c^i) \diamondsuit (b \cdot c^i)$

# Bibliography

[1] Kalganova T. and Miller J. Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 54–63. IEEE Computer Society, July 1999.

[2] Kalganova T. and Miller J. Circuit layout evolution: An evolvable hardware approach. In *Coloquium on Evolutionary hardware systems. IEE Colloquium Digest.*, London, UK, 1999.

[3] Kalganova T., Miller J., and Fogarty T.C. Evolution of the digital circuits with variable layouts. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 2 of *ISBN 1-55860-611-4*, page 1235, Orlando, USA, July 1999. Morgan Kaufmann, San Francisco, CA.

[4] Kalganova T. An extrinsic function-level evolvable hardware approach. In Poli R., Banzhaf W., Langdon W.B., Miller J., Nordin P., and Fogarty T.C., editors, *Proc. of the Third European Conference on Genetic Programming, EuroGP2000*, volume 1802 of *Lecture Notes in Computer Science*, pages 60–75, Edinburgh, UK, 2000. Springer-Verlag.

[5] Kalganova T. A new evolutionary hardware approach for logic design. In Annie S. Wu, editor, *Proc. of the Genetic and Evolutionary Computation Conference (GECCO'99) Student Workshop*, pages 360–361, Orlando, USA, 1999.

[6] Kalganova T. Bidirectional incremental evolution in ehw. In *Proc. of the Second NASA/DoD Workshop on Evolvable Hardware*, July 2000.

[7] Kalganova T., Miller J., and Fogarty T. Some aspects of an evolvable hardware approach for multiple-valued combinational circuit design. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 78–89, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[8] Kalganova T., Miller J., and Lipnitskaya N. Multiple-valued combinational circuits synthesized using evolvable hardware approach. In *Proc. of the 7th Workshop on Post-Binary Ultra Large Scale Integration Systems (ULSI'98) in association with ISMVL'98*, Fukuoka, Japan, May 1998. IEEE Press.

[9] Miller J.F., Kalganova T., Lipnitskaya N., and Job D. The genetic algorithm as a discovery engine: Strange circuits and new principles. In *Proc. of the AISB'99 Symposium on Creative Evolutionary Systems, CES'99*, ISBN 1-902956-03-6, pages 65–74. Edinburgh, UK, The Society for the Study of Arificial Intelligence and Simulation of Behaviour, April 1999.

[10] Xilinx Inc. *The Programmable Logic Data Book.* San Jose, California, USA, 1994.

[11] Hollingworth G., Smith A., and Tyrrell A. The intrinsic evolution of virtex devices through internet reconfigurable logic. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 72–79, Edinburgh, UK, 2000. Springer.

[12] Zebulum R., Vellasco M., and Pacheco M. Evolvable hardware systems: Taxonomy, survey and applications. In *Proc. Of the 1st Int. Conf. on Evolvable*

*Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 344–358, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[13] Almaini A.E.A. and Zhuang N. Variable ordering of bdds for multioutput boolean functions using evolutionary techniques. In *Proc. of the 4th IEEE Int. Conference on Electronics, Circuits and Systems, ICECS'97*, pages 1239–1244, 1997.

[14] Almaini A.E.A., Zhuang N., and Bourset F. Minimisation of multioutput binary decision diagrams using hybrid generic algorithm. *IEE Electronic Letters*, 31(20):1722–1723, 1995.

[15] Almaini A.E.A. and Zhuang N. Using genetic algorithms for the variable ordering of reed-muller binary decision diagrams. *Microelectronic Journal*, 26(4):471–480, 1995.

[16] Bystrov A. and Almaini A.E.A. Testability and test compaction for decision diagram circuits. *IEE Proceedings on Circuits, Devices and Systems.*, 146(4):153–158, 1999.

[17] Murakawa M., Yoshizawa S., Kajitani I., Furuya T., Iwata M., and Higuchi T. Hardware evolution at function level. In *Proc. of the Fifth International Conference on Parallel Problem Solving from Nature (PPSNIV)*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 1996.

[18] Higuchi T., Murakawa M., Iwata M., Kajitani I., Liu W., and Salami M. Evolvable hardware at function level. In *Proc. of IEEE 4th Int. Conference on Evolutionary Computation, CEC'97*. IEEE Press, NJ, 1997.

[19] Sanchez E., Mange D., Sipper M., Tomassini M., Perez-Uribe A., and Stauffer A. Phylogeny, ontogeny and epigenesis: Three sourses of biological inspiration

for softening hardware. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 35–54, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[20] Kitano H. Morphogenesis for evolvable systems. In Sanchez E. and Tomassini M., editors, *Towards Evolvable Hardware. The Evolutionary Engineering Approach.*, volume 1062 of *Lecture Notes in Computer Science*, pages 99–117. Springer-Verlag, 1996.

[21] Zebulum R.S., Pacheco M.A., and Vellasco M. Analog circuits evolution in extrinsic and intrinsic modes. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 154–165, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[22] Thompson A. Silicon evolution. In Koza J., editor, *Proc. of the Conference on Genetic Programming, GP'96*, Lecture Notes in Computer Science, pages 444–452, Edinburgh, UK, 1996. MIT Press.

[23] Higuchi T., Iwata M., Kajitani J., Iba N., Hirao J., Furuya T., and Manderick B. Evolvable hardware and its applications to pattern recognition and fault toulerant systems. In Sanchez E. and Tomassini M., editors, *Towards Evolvable Hardware. The Evolutionary Engineering Approach.*, volume 1062 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, 1996.

[24] Stoica A., Zebulum R., and Keymeulen D. Mixtrinsic evolution. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 208–217, Edinburgh, UK, 2000. Springer.

[25] Miller J. F., Thomson P., and Fogarty T. C. Genetic algorithms and evolution strategies. In Quagliarella D., Periaux J., Poloni C., and Winter G., editors, *Engineering and Computer Science: Recent Advancements and Industrial Applications*. Wiley, 1997.

[26] Coello C. A., Christiansen A. D., and Hernndez A. A. Use of evolutionary techniques to automate the design of combinational circuits. *International Journal of Smart Engineering System Design*, 2(4), 2000.

[27] Coello C. A., Christiansen A. D., and Hernndez A. A. Towards automated evolutionary design of combinational circuits. *Computers and Electrical Engineering*, 2000.

[28] Miller J. Evolution of digital filters using a gate array model. In Poli R., Voigt H-M., Cagnoni S., Corne D., Smith G.D., and Fogarty T.C., editors, *Proc. of the First EvoIASP'99 Workshop on Image Analysis and Signal Processing*, volume 1596 of *Lecture Notes in Computer Science*, pages 17–30, Goteborg, Sweden, 1999. Springer-Verlag.

[29] Miller J. Digital filter design at gate-level using evolutionary algorithms. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 1 of *ISBN 1-55860-611-4*, pages 1127–1143, Orlando, USA, July 1999. Morgan Kaufmann, San Francisco, CA.

[30] Hamilton A., Thomson P., and Tamplin M. Experiments in evolvable filter design using pulse based programmable analogue vlsi models. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 61–71, Edinburgh, UK, 2000. Springer.

[31] Koza J.R., Benett III F.H., Andre D., and Keane M.A. Four problems for which a computer program evolved by genetic programming is competitive with human performance. In *Proc. of 1996 IEEE Int. Conference on Evolutionary Computation.*, pages 1–10. IEEE Press, 1996.

[32] Koza J.R., Andre D., Benett III F.H., and Keane M.A. Design of a high-gain operational amplifier and other circuits by means of genetic programming. In Angelint P.J., Reynolds R.G., McDonnell J.R., and Eberhart R., editors, *Proc. of the 5th International Conference on Evolutionary Programming VI, EP97.*, volume 1213 of *Lecture Notes in Computer Science*, pages 125–136, Indianapolis, Indiana, USA, 1997. Berlin: Springer-Verlag.

[33] Bennett III F.H., Koza J.R., Yu J., and Mydlowec W. Automatic synthesis, placement and routing of an amplifier circuit by means of genetic programming. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 1–10, Edinburgh, UK, 2000. Springer.

[34] Bennett III F.H., Koza J.R., Andre D., and Keane M.A. Evolution of a 60 decibel op amp using genetic programming. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 455–469, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[35] Thompson A., Harvey J., and Hasbands P. Unconstraned evolution and hard consequences. In Sanchez E. and Tomassini M., editors, *Towards Evolvable Hardware. The Evolutionary Engineering Approach.*, volume 1062 of *Lecture Notes in Computer Science*, pages 136–165. Springer-Verlag, 1996.

[36] Thompson A. An evolved circuit, intrisic in silicon, entwined with physics. In Higuchi T., Iwata M., and Liu W, editors, *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 390–405, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[37] Thomspon A. On the automatic design of robust electronics through artificial evolution. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 13–24, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[38] Huelsbergen L., Rietman E., and Slous R. Evolution of astable multivibrators in silico. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 66–77, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[39] Lohn J.D. and Colombano S.P. Automated analog circuit synthesis using a linear representation. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 125–134, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[40] Zebulum R.C., Pacheco M.A., and Vellasco M. Comparison of different evolutionary methodologies applied to electronic filter design. In *Proc. of 1998 IEEE Int. Conference on Evolutionary Computation.*, Anchorage, Alaska, USA, 1998. IEEE Press.

[41] Zebulum R.S., Pacheco M.A., and Vellsco M. Artificial evolution of active filters: A case study. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc.*

*of the First NASA/DoD Workshop on Evolvable Hardware*, pages 66–75. IEEE Computer Society, July 1999.

[42] Murakawa M., Yoshizawa S., Adachi T., Suzuki S., Takasuka K., Iwata M., and Higuchi T. Analogue ehw chip for intermediate frequency filters. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 134–143, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[43] Perkins S., Porte R., and Harvey N. Everything on the chip: a hardware-based self-cintained spatially-strctured genetic algorithm for signal processing. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 165–174, Edinburgh, UK, 2000. Springer.

[44] Zebulum R.S., Stoica A., and Keymeulen D. A flexible model of a cmos field programmable transistor array targeted for hardware evolution. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 274–283, Edinburgh, UK, 2000. Springer.

[45] Tufte G. and Haddow P.C. Prototyping a ga pipeline for complete hardware evolution. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 18–25. IEEE Computer Society, July 1999.

[46] Kajitani I., Hoshino T., Nishikawa D., Yokoi H., Nakaya S., Yamauchi T. oand Inuo T., Kajihara N., Iwata M., Keymeulen D., and Higuchi T. A gate-level ehw

chip: Implementing ga operations and reconfigurable hardware on a single lsi. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 1–12, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[47] Keymeulen D., Durantez M., Konaka K., Kuniyoshi J., and Higuchi T. An evolutionary robot navigation system using a gate-level evolvable hardware. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 195–209, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[48] Higuchi T., Iwata M., Kajitani I., Murakawa M., Yoshizawa S., and Furuya T. Hardware evolution at gate and function level. In *Proc. of the Int. Conf. on Biologically Inspired Autonomous Systems: Computation, Cognition and Action.* Durham, NC, USA, 1996.

[49] Murakawa M., Yoshizawa S., and Higuchi T. Adaptive equalization of digital communication channels using evolvable hardware. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 379–389, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[50] Liu W., Murakawa M., and Higuchi T. Atm cell scheduling by function level evolvable hardware. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 180–192, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[51] Koza J. R. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, 1994.

[52] Salami M., Iwata M., and Higuchi T. Lossless image compression by evolvable hardware. In Husbands P. and Harvey I., editors, *Proc. of the Fourth European Conference on Artificial Life (ECAL97)*, pages 407–416. A Bradford book, MIT Press, 1997.

[53] Miller J. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 1 of *ISBN 1-55860-611-4*, pages 1135–1142, Orlando, USA, July 1999. Morgan Kaufmann, San Francisco, CA.

[54] Torresen J. A divide-and-conquer approach to evolvable hardware. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 57–65, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[55] Torresen J. Increased complexity evolution applied to evovable hardware. In *Smart Engineering System Design, ANNIE'99*. St. Louis, USA, 1999.

[56] Harvey I. Artificial evolution for real problems. In Gomi T., editor, *Proc. of the 5th Intl. Symposium on Evolutionary Robotics, Evolutionary Robotics: From Intelligent Robots to Artificial Life (ER'97)*, Tokyo, Japan, 1997. AAI Books.

[57] Floreano D. and Mondada F. Hardware solutions for evolutionary robotics. In Husbands P. and Meyer J-A., editors, *Proc. of the First European Workshop on Evolutionary Robotics*. Berlin: Springer-Verlag, 1998.

[58] Fukunaga A.S. and Kahng A.B. Improving the performance of evolutionary optimization by dynamically scaling the evolution function. In *Proc. of the 1995 IEEE Conference on Evolutionary Computation*, volume 1, pages 182–187, Perth, Australia, 29-1 1995. IEEE Press.

[59] Gomez F. and Miikkulainen R. Solving non-markovian control tasks with neurevolution. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI'99)*, Stockholm, Sweden, 1999. Denver: Morgan Kaufmann.

[60] Dorigo M. and Gambardella L. M. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[61] Dorigo M., Maniezzo V., and Colorni A. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics–Part B*, 26(1):29–41, 1996.

[62] Dorigo M. and Di Garo G. The ant colony optimisation meta-heuristic. In Corne D., Dorigo M., and Glover F., editors, *New Ideas in Optimisation.* McGraw-Hill, 1999.

[63] Lohn J.D., Haith G.L., Colombano S.P., and Stassinopoulos D. A comparison of dynamic fitness schedules for evolutionary design of amplifiers. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 87–92. IEEE Computer Society, July 1999.

[64] Kasai Y., Sakanashi H., Murakawa M., Kiryu S., Marston N., and Higuchi T. Initial evaluation of an evolvable microwave circuit. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 103–112, Edinburgh, UK, 2000. Springer.

[65] Layzell P. A new research tool for intrinsic hardware evolution. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 47–56, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[66] Manovit C., Aporntewan C., and Chongstivatana P. Synthesis of synchronous sequencial logic circuits from partial input/output sequences. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 98–105, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[67] Levi D. and Guccione S.A. Genetic fpga: Evolving stable circuits on mainstream fpga devices. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 12–17. IEEE Computer Society, July 1999.

[68] Damiani E., Tettamanzi A.G.B., and Liberali V. On-line evolution of fpga-based circuits: A case study on hash functions. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 26–33. IEEE Computer Society, July 1999.

[69] Pollack J.B., Lipson H., Ficici S., Funes P., Hornby G., and Watson R. Evolutionary techniques in physical robotics. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 175–186, Edinburgh, UK, 2000. Springer.

[70] Mondada F. and Floreano D. Evolution and mobile autonomus robotics. In Sanchez E. and Tomassini M., editors, *Towards Evolvable Hardware. The Evolutionary Engineering Approach.*, volume 1062 of *Lecture Notes in Computer Science*, pages 221–249. Springer-Verlag, 1996.

[71] Naito T., Odagiri R., Matsunaga J., Tanifuji M., and Murase K. Genetic evolution of a logic circuit which controls an autonomous mobile robot. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware*

*(ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 210–219, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[72] Silva A., Neves A., and Costa E. Evolving controllers for autonomous agents using genetically programmed networks. In Poli R., Nordin P., Langdon W.B., and Fogarty T.C., editors, *Proc. of the 2nd European Workshop on Genetic Programming, EuroGP'99*, volume 1598 of *Lecture Notes in Computer Science*, pages 255–269, Goteburgh, Sweden, 1999. Springer-Verlag.

[73] Yamamoto J. and Anzai J. Autonomous robot with evolving algorithm based on biological systems. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 220–233, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[74] Ito H. and Furuya T. Memory-based neural network and its application to a mobile robot with evolutionary adn experience learning. In Higuchi T., Iwata M., and Liu W., editors, *Proc. Of the 1st Int. Conference on evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 234–246, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[75] Thompson A. Evolving electronic robot controllers that exploit hardware resources. In *Proc. of the Third European Conference on Artificial Life (ECAL95)*, pages 640–656. Springer-Verlag, 1995.

[76] Ebner M. Evolution of a control architecture for a mobile robot. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 303–310, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[77] Koza J.R., Yu J., Keane M.A., and Mydlowex W. Evolution of a controller with a free variable using genetic programming. In Poli R., Banzhaf W., Langdon W.B., Miller J., Nordin P., and Fogarty T.C., editors, *Proc. of the Third European Conference on Genetic Programming, EuroGP2000*, volume 1802 of *Lecture Notes in Computer Science*, pages 91–106, Edinburgh, UK, 2000. Springer-Verlag.

[78] Andersson B., Svensson P., Nordahl M., and Nordin P. On-line evolution of control for a four-legend robot using genetic programming. In Cagnoni S. et al, editor, *Proc. of EvoWorkshops 2000: EvoIASP, EvoSCONDI, EvoTel, SvoSTim, EvoRob and EvoFlight*, volume 1803 of *Lecture Notes in Computer Science*, pages 319–326, Edinburgh, UK, 2000. Springer-Verlag.

[79] Hornby G.S., Takamura S., Hanagata O., Fujita M., and Pollack J. Evolution of controllers from a high-level simulator to a high dof robot. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 80–89, Edinburgh, UK, 2000. Springer.

[80] Hollingworth G., Tyrrel A., and S. Smith. Simulation of evolvable hardware to solve low level image processing tasks. In Poli R., Voigt H-M., Cagnoni S., Corne D., Smith G.D., and Fogarty T.C., editors, *Proc. of the First EvoIASP'99 Workshop on Image Analysis and Signal Processing*, volume 1596 of *Lecture Notes in Computer Science*, pages 46–58, Goteborg, Sweden, 1999. Springer-Verlag.

[81] Dumoulin J., Foster J.A., Frenzel J.F., and McGrew S. Special purpose image convolution with evolvable hardware. In Cagnoni S. et al, editor, *Proc. of EvoWorkshops 2000: EvoIASP, EvoSCONDI, EvoTel, SvoSTim, EvoRob and*

*EvoFlight*, volume 1803 of *Lecture Notes in Computer Science*, pages 1–11, Edinburgh, UK, 2000. Springer-Verlag.

[82] Yasunaga M., Nakamura T., Yoshihara I., and Kim J.H. Genetic algorithm-based design methodology for pattern recoginition hardware. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 264–273, Edinburgh, UK, 2000. Springer.

[83] Takanaka M., Sakahashi H., Salami H., Iwata M., Kurita T., and Higuchi T. Data compression for digital color electrophotographic printer with evolvable hardware. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 106–114, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[84] Stoica A., Fukunaga A., Hayworth L., and Salazar-Lazaro C. Evolvable hardware for space applications. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 166–173, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[85] Manderick B. and Higuchi T. Evolvable hardware: An outlook. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 305–311, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[86] Salami M., Murakawa M., and Higuchi T. Data compression based on evolvable hardware. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology*

*to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 169–179, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[87] Hamilton A., Papathanasiou K., Tamplin M., and Brandtner T. Palmo: Field programmable analogue and mixed-signal vlsi for evolvable hardware. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 335–344, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[88] Langeheine J., Folling S., Meier K., and Schemmel J. Initial evaluation of an evolvable microwave circuit. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 123–132, Edinburgh, UK, 2000. Springer.

[89] Stoica A., Keymeulen D., Tawel R., Salazar-Lazaro C., and Li W. Evolutionary experiments with a fine-grained reconfigurable architecture for analog and digital cmos circuits. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 76–84. IEEE Computer Society, July 1999.

[90] Thompson A. and Layzell P. Evolution of robustness in an electronics design. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 218–228, Edinburgh, UK, 2000. Springer.

[91] Layzell P. and Thompson A. Understanding inherent qualities of evolved circuits: Evolutionary history as a predictor of fault tolerance. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int.*

*Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 133–144, Edinburgh, UK, 2000. Springer.

[92] Millet P. and Heudin J.-C. Fault tolerance of a large-scale mimd architecture using a genetic algorithm. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 356–363, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[93] Layzell P. Inherent qualities of circuits designed by artificial evolution: A preliminary study of populational fault tolerance. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 85–86. IEEE Computer Society, July 1999.

[94] Bradley D.W. and Tyrrell A.M. Automatic synthesis, placement and routing of an amplifier circuit by means of genetic programming. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 11–20, Edinburgh, UK, 2000. Springer.

[95] Moreno J.M., Madrenas J., Faura J., Canto E., Cabestany J., and Insenser J.M. Feasible evolutionary and self-repairing hardware by means of the dynamic reconfiguration capabilities of the fipsoc devices. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 345–355, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[96] Moreno J., Madrenas J., CAbestany J., Canto E., Kiebik R., Faura J., and Insenser J. Realization of self-repairing and evolvable hardware structures by

means of implicit self-configuration. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 182–187. IEEE Computer Society, July 1999.

[97] Iba I., Iwata M., and Higuchi T. Machine learning approach to gate-level evolvable hardware. In Higuchi T., Iwata M., and Liu W., editors, *Proc. Of the 1st Int. Conference on evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 327–344, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[98] Masher J., Cavalieri J., Frenzel J., and Foster J.A. Representation and robustness for evolved sorting networks. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 255–261. IEEE Computer Society, July 1999.

[99] Xilinx Inc. *XC6200 field programmable gate arrays databook.* http://www.xilinx.com/partinfo/6200.pdf, 1995.

[100] Xilinx Inc. *Virtex field prograsmmable gate arrays databook.* http://www.xilinx.com/partinfo/ds003.pdf, 1999.

[101] Xilinx Inc. *JBits documentation.* JBits 2.0.1 documentation., 1999.

[102] Miller J. and Thomson P. Evolving digital electronic circuits for real-valued function generation using a genetic algorithm. In Koza J.R., Banzhaf W., Chellapilla K., Deb K., Dorigo M., Fogel D.B., Garzon M.H., Goldberg D.E., Iba H., and Riolo R.L., editors, *Genetic Programming 1998: Proc. of the Third Annual Conference (GP'98)*, pages 863–868, Madison, Wisconsin. San Francisco, July 1998. CA: Morgan Kaufmann.

[103] Hernandez-Aguirre A.H., Coello Coello C.A., and Buckles B.P. A genetic programming approach to logic function synthesis by means of multiplexer. In

Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 46–53. IEEE Computer Society, July 1999.

[104] Miller J. On the filtering properties of evolved gate arrays. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 2–11. IEEE Computer Society, July 1999.

[105] Coello Coell C.A., Zavala R.L., Garcia B.M., and Aguirre A.H. Ant colony system for the design of combinational logic circuits. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, UK, 2000. Springer.

[106] Louis S.J. *Genetic Algorithms as a Computational Took for Design.* Phd thesis, Department of Computer Science, Indiana University, August 1993.

[107] Coello C.M. A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information systems*, 1(3):269–308, 1999.

[108] Coello C. A., Christiansen A. D., and Aguirre A.H. Automated design of combinational logic circuits using genetic algorithms. In Smith D.G., Steele N.C., and Albrecht R.F., editors, *Proc. Of the Int. Conference on Artificial Neural Nets and Genetic Algorithms, ICANNGA'97*, pages 335–338. University of East Anglia, Norwich, England, Springer-Verlag, 1997.

[109] Hernandez A.A., Buckles B.P., and Coello C.C.A. Gate-level synthesis of boolean functions using binary multiplexers and genetic programming. In *Proc. of the Congress on Evolutionary Computation, CEC'00.* San-Diego, California, July 2000.

[110] Miller J. and Thomson P. Aspects of digital evolution: Geometry and learning. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 25–35, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[111] Fogarty T. C., Miller J.F., and Thomson P. Evolving digital logic circuits on xilinx 6000 family fpgas. In P. K. Chawdhry, R. Roy, and R.K. Pant, editors, *The 2nd Online Conf. on Soft Computing in Engineering Design and Manufacturing (1997)*, pages 299–305. Springer-Verlag, London, 1998.

[112] Miller J. and Thomson P. Aspects of digital evolution: Evolvability and architecture. In Eiben A. et al., editor, *Proc. of the Fifth International Conference on Parallel Problem Solving from Nature (PPSNV)*, volume 1498 of *Lecture Notes in Computer Science*, pages 927–936. Springer-Verlag, Heidelberg, September 1999.

[113] Miller J. and Thomson P. Cartesian genetic programming. In Poli R., Banzhaf W., Langdon W.B., Miller J., Nordin P., and Fogarty T.C., editors, *Proc. of the Third European Conference on Genetic Programming, EuroGP2000*, volume 1802 of *Lecture Notes in Computer Science*, pages 121–133, Edinburgh, UK, 2000. Springer-Verlag.

[114] Iwata M., Kajitani I., Yamada H., Iba H., and Higuchi T. A pattern recognition system using evolvable hardware. In *Proc. of the Fifth International Conference on Parallel Problem Solving from Nature (PPSNIV)*, volume LNCS 1141 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 1996.

[115] Poli R. Sub-machine-code gp: New resultss and extensions. Technical Report Technical report CSRT-98-19, University of Birmingham, School of Computer Sciencem, Birmingham, UK, 1998.

[116] Vassilev V. and Miller J. The advantages of landscape neutrality in digital circuit evolution. In Miller J., Thompson A., Thomson P., and Fogarty T.C., editors, *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 252–263, Edinburgh, UK, 2000. Springer.

[117] Gomez F. and Miikkulainen R. Incremental evolution of complex general behaviour. *Adaptive Behaviour.*, 5:317–342, 1997.

[118] Filliat D., Kodjabachian J., and Meyer J.A. Incremental evolution of neural controllers for navigation in a 6-legged robot. In Sugisaka and Tanaka, editors, *Proc. of the Fourth International Symposium on Artificial Life and Robotics.* Oita Univ. Press, 1999.

[119] Harvey J., Husbands P., and Cliff D. Seeing the light: Artificial evolution, real vision. In *From Animals to Animats 3: Proc. of the 3rd Int. Conf. on Simulation of Adaptive Behaviour*, pages 392–401, 1994.

[120] Harvey J., Husbands P., and Cliff D. Issues in evolutionary robotics. In *From Animals to Animats 2: Proc. of the 2nd Int. Conf. on Simulation of Adaptive Behaviour*, pages 364–374, 1992.

[121] Nolfi S., Floreano D., Miglino O., and Mondada F. How to evolve autonomous robots: Different approaches in evolutionary robots. In *Proc. of the Artificial Life IV*, pages 190–197, 1994.

[122] Winkeler J.F. and Manjunath B.S. Incremental evolution of neural controllers for navigation in a 6-legged robot. In Koza J.R., Banzhaf W., Chellapilla K., Deb K., Dorigo M., Fogel D.B., Garzon M.H., Goldberg D.E., Iba H., and Riolo R.L., editors, *Genetic Programming 1998: Proc. of the Third Annual Conference (GP'98)*, pages 403–411, Madison, Wisconsin. San Francisco, July 1998. CA: Morgan Kaufmann.

[123] Chavas J., Corne C., Horvai P., Kodjabachian J., and J.A. Meyer. Incremental evolution of neural controllers for robust obstacle-avoidance in khepera. In Husbands P. and Meyer J-A., editors, *Proc. of the First European Workshop on Evolutionary Robotics.* Berlin: Springer-Verlag, 1998.

[124] King R. and Noval M. Sybil: A system for the incremental evolution of distributed, heterogeneous database layers. In *Proc. of the 2nd Annual Americas Conference on Information Systems Minitrack on Heterogeneous Interoperability*, Phoenix, Arizona, USA, 1996.

[125] Ramakrishnan S. An object-oriented design for modelling business rules in resource allocation jobs. In *Proc. of the Object-Oriented Information Systems (OOIS 94)*, pages 105–113, London, UK, 1994.

[126] Kitano H. Building complex systems using developmental process: An engineering approach. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 218–229, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.

[127] Geiger R.L., Allen P.E., and Strader N.R. *VLSI Design Techniwues for Analog and Digital Circuits.* McGraw-Hill Publishing Company, 1990.

[128] Horowitz P. and Hill W. *The Art of Electronics.* Cambridge University Press, second edition edition, 1990.

[129] Weste N. and Eshraghian K. *Principles of CMOS VLSI Design. A Systems Perspective.* Addison-Wesley, 1985.

[130] Brzozowski J.A. and Seger C.-J. *Asynhronous Circuits.* Springler Verlag, New York, NY, 1995.

[131] Srinivas N. and Deb K. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.

[132] Coello C.A. An updated survey of evolutionary multiobjective optimization techniques: State of the art and future trends. In *Proc. of the Congress on Evolutionary Computation, CEC'99*, volume 2 of *ISBN 0-7803-5536-9*, pages 3–13. Washington DC, USA, IEEE Press, Piscataway, NJ, July 1999.

[133] Fonseca C.M. and Fleming P.J. An overview of evolutionary algorithms in multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 3(1):1–16, 1995.

[134] Hans A.E. Multicriteria optimization for highly accurate systems. *Multicriteria Optimization in Engineering and Sciences, Mathematical concepts and methods in science and engineering*, 19:309–352, 1988.

[135] Miller J., Job D., and Vassilev V. Principles in the evolutionary design of digital circuits - part 1. *Genetic Programming and Evolvable Machines*, 1(1/2):7–37, 2000.

[136] Cantu-Paz E. A survey of parallel genetic algorithms. *Calculateurs Parallels*, 10(2), 1998.

[137] Koza J. R. *Genetic Programming.* MIT Press, Cambridge, Massachusetts, 1992.

[138] Koza J. R., Andre D., Bennet III F.H., and Keane M.A. Design of a high-gain operational amplifier and others circuits by means of genetic programming. In *Proc. of the 6th Int. Conference on Evolutionary Programming.*, volume 1213 of *Lecture Notes in Computer Science*, pages 125–135. Springer-Verlag, 1997.

[139] Goeke M., Sipper M., Mange D., Stauffer S., Sanchez E., and Tomassini M. Online autonomous evolware. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in*

*Computer Science*, pages 96–106, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.

[140] Clark G.R. A novel function-level ehw architecture within modern fpgas. In *Proc. of the Congress on Evolutionary Computation, CEC'99*, volume 2 of *ISBN 0-7803-5536-9*, pages 830–833. Washington DC, USA, IEEE Press, Piscataway, NJ, July 1999.

[141] Moraga C. and W. Wang. Evolutionary methods in the design of quaternary digital circuits. In *Proc. of the 28th Int. Symposium on Multiple-Valued Logic (ISMVL'98)*, Fujioka, Japan, 1998. IEEE Press.

[142] Wang W. and C. Moraga. Design of multiple-valued circuits using genetic algorithms. In *Proc. of the 26th Int. Symposium on Multiple-Valued Logic (ISMVL'96)*, pages 216–221, Santiago de Compostela, Spain, 1996. IEEE-CS-Press.

[143] Wang W. and C. Moraga. Evolutionary synthesis of current-mode cmos 4-valued circuits. In *Proc. Of the 3rd Int. Conf. on Signal Processing*, Beeijing, P.R. China, 1996. IEEE-CS-Press.

[144] Allen C.M. and Givone D.D. A minimization technique for multiple-valued logic systems. *IEEE Transactions on Computers*, 17:182–184, 1968.

[145] Bernstein B.A. Modular representation of finite algebras. In *Proc. of 7th Int. Congress of Mathematics*, volume 2, pages 207–216, 1924.

[146] Onneweer S., Kerfhoff H., and Butler J. Structured computer-aided design of current mode circuits. In *Proc. of the 18th Int. Symposium on Multiple-Valued Logic (ISMVL'88)*, pages 21–30. IEEE Press, 1988.

[147] Utsumi T., Kamiura N., Nata Y., and Yamato K. Multiple-valued programmable logic arrays with universal literals. In *Proc. of the 27th Int. Symposium on Multiple-Valued Logic (ISMVL'97)*, pages 169–174, Nova Scotia, Canada, May 1997. IEEE-CS-Press.

[148] Hata Y. and K. Yamato. Multiple-valued logic functions represented by tsum, tproduct, not and variables. In *Proc. of the 23th Int. Symposium on Multiple-Valued Logic (ISMVL'93)*, pages 222–227. IEEE-CS-Press, May 1993.

[149] Webb D.L. Generation of any n-valued logic by one binary operator. *Proc. National Academy of Science*, 21:252–254, 1935.

[150] Hurst S.L. Multiple-valued logic: Its status and its future. *IEEE Transactions on Computers*, C-33:1160–1179, 1984.

[151] Davio M. and Deschamps J.P. Synthesis of discete functions using i$^2$l technology. *IEEE Transactions on Computers*, C-30:653–661, 1981.

[152] Jain A.K., Abd-El-Barr M.H., and Bolton R.J. A new structure of cmos realization of mvl functions. *Int. J. Electronics*, 74:251–163, 1993.

[153] Vranesic Z., Lee S., and Smith L. A many-valued algebra for switching systems. *IEEE Transactions on Computers*, C-19(10):964–971, 1970.

[154] Post E.I. Introduction to a general theory of elementary propositions. *Amer. J. Math.*, 43:163–185, 1921.

[155] Dueck G.W. and Miller D.M. A 4-valued pla using the modsum. In *Proc. of the 16th Int. Symposium on Multiple-Valued Logic (ISMVL'86)*, pages 232–240. IEEE Press, 1986.

[156] Pelayo F.J., Prieto A., Lloris A., and Ortega J. Cmos current - mode multiple-valued pla's. *IEEE Transactions on Circuits and Systems.*, 38:434–441, 1991.

[157] Vranesic Z. and Smith K.C. Electronic circuits for multiple-valued digital systems. *Computer Science and Multiple-Valued Logic: Theory and Applications*, 1977.

[158] Pugsley J.H. and Silio C.B. Some $i^2l$ circuits for multiple-valued logic. In *Proc. of the 8th Int. Symposium on Multiple-Valued Logic (ISMVL'78)*, pages 23–31. IEEE Press, 1978.

[159] Abd-El-Barr M.H. and Vranesic Z. Charge-coupled devices implementation of multiple-valued logic systems. *IEEE Transactions on Computers*, 136:306–314, 1986.

[160] Sasao T. On the optimal design of multiple-valued pla's. *IEEE Transactions on Computers*, C-38(5):582–592, 1989.

[161] Jain A.K., Bolton R.J., and Abd-ElBarr M.H. Cmos multiple-valued logic design - part1: Circuit implementation. *IEEE Transactions on Circuits and Systems - I. Fundamental theory and applications.*, 40(8):503–514, 1993.

[162] Jain A.K., Bolton R.J., and Abd-ElBarr M.H. Cmos multiple-valued logic design - part2: Function realization. *IEEE Transactions on Circuits and Systems - I. Fundamental theory and applications.*, 40(8):515–522, 1993.

[163] Zilic Z. and Vranesic Z. Current-mode cmos galois field circuits. In *Proc. of the 23th Int. Symposium on Multiple-Valued Logic (ISMVL'93)*, pages 245–250. IEEE-CS-Press, May 1993.

[164] Piezchala E., Perkowski M., and Grydel S. A field programming analog array for continuous, fuzzy and multi-valued logic applications. In *Proc. of the 24th Int. Symposium on Multiple-Valued Logic (ISMVL'94)*, pages 148–155, Santiago de Compostela, Spain, 1994. IEEE-CS-Press.

[165] Deng X., Hanyu T., and Kameyama M. Design and evaluation of a current-mode multiple-valued pla based on a resonant tunneling transistor model. *IEEE Proc. - Circuits Devices System*, 141(6):445–450, 1994.

[166] Abd-El-Barr M.H. and Hasan M.N. New mvl-pla structures based on current-mode cmos technology. In *Proc. of the 26th Int. Symposium on Multiple-Valued Logic (ISMVL'96)*, pages 98–103, Santiago de Compostela, Spain, 1996. IEEE-CS-Press.

[167] Stonham T.J. *Digital Logic Techniques. Principles and Practice.* Chapman-Hall, third edition edition, 1996.

[168] Masakazu Shoji. *CMOS Digital Circuit Technology.* Prentice-Hall International, Inc., 1988.

[169] Almaini A.E.A. *Electronic Logic Systems, 3rd ed.* Prentice Hall, 1994.

[170] Poli R. Evolution of graph-like programs with parallel distributed genetic programming. In Bäck T., editor, *Genetic Algorithms: Proc. of the Seventh International Conference.*, pages 346–353. Morgan Kaufmann, San Francisco, CA, 1997.