Lukáš Sekanina

Evolutionary Approach to the Implementation Problem

Habilitation Thesis

Brno University of Technology
Faculty of Information Technology
Department of Computer Systems

2005

Abstract

Evolutionary circuit design and evolvable hardware traditionally belong to the area of electrical engineering. In this habilitation thesis, we interpret the evolutionary design of computational systems from the perspective of computer science. In particular, the evolution is able to produce a computing system satisfying the given specification; however, in general, we do not understand how and why the system performs a computation. It means that we cannot classify the evolved systems as computing mechanisms. On the basis of experimental results, we have shown the following: (1) There is no significant relation among the complexity of the evolved circuits, chromosome size and novelty obtained. Human-competitive results can be evolved at any level of interest if the problem is formulated in a clever way. (2) It is possible to speed up the digital circuit evolution using a common FPGA for a reasonable class of applications. (3) It is possible to evolve sequential circuits at reconfigurable platforms operating at sub-gate levels. In particular, the RS circuit was (re)discovered at the transistor level. (4) Evolution is able to form computational elements in extreme low temperatures. (5) The embodiment can be utilized to introduce additional functions to digital modules in a manageable way; in particular, polymorphic gate-level modules were discovered. All the problems investigated should contribute to the explanation what it means for a computational system to be designed evolutionarily and what we can expect from the evolutionary approach that works on the position of human designer.

Acknowledgements

This work was carried out in the period 2002–2005 at the Faculty of Information Technology of the Brno University of Technology, Czech Republic. Chapter 6 is based on the results obtained during my Fulbright stay with the evolvable hardware group at NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena CA in 2004.

I would like to thank my colleagues and students (co-authors of many papers) at the Faculty of Information Technology, namely Zdeněk Kotásek, Otto Fučík, Richard Růžička, Josef Strnadel, Vladimír Drábek, Michal Bidlo, Zdeněk Vašíček, Tomáš Pečenka, Tomáš Martínek, Jan Kořenek and Štěpán Friedl. I would like to acknowledge my colleagues at the Jet Propulsion Laboratory for their support, namely Ricardo S. Zebulum, Didier Keymeulen, Adrian Stoica, Ian Ferguson and Vu Duong.

Last, but not least, I am very grateful to my wife for her love, patience and tolerance.

The research presented in this thesis was partly supported by the following projects: *Evolvable Hardware-Based Application Design Methods* and *Modern Methods of Digital Systems Design* (financed by the Grant Agency of the Czech Republic under numbers 102/03/P004 and 102/04/0737), *Using Evolutionary Algorithms for Implementation of Adaptive Image Filter in FPGA* and *Evolutionary Design of Sorting and Median Networks Using FPGA* (financed by the Ministry of Education, Youth and Sports under numbers FR2987/2005/G1 and FR3042/2005/G1) and *Functional Recovery of Damaged Electronic Systems for Space Applications Using Evolvable Hardware* sponsored by the Fulbright organization. I also greatly appreciated the financial support that I received from Siemens (the Siemens Award for outstanding PhD thesis in 2003), Josef Hlávka Foundation (Josef Hlávka Award 2003) and Brno University of Technology (Rector's Award for outstanding research and pedagogical results 2005).

Brno, November 2005 *Lukáš Sekanina*

Contents

Symbols and Abbreviations

$\hbar$	Planck's reduced constant ($1.05457168 \times 10^{-34}$J.s)
k_B	Boltzmann's constant ($1.3806505 \times 10^{-23}$J.K^{-1})
m_e	the electron mass ($9.1093826 \times 10^{-31}$ kg)
ADC	Analog-to-Digital Converter
ASIC	Application Specific Integrated Circuit
CFB	Configurable Functional Block
CGP	Cartesian Genetic Programming
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide-Semiconductor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DSP	Digital Signal Processor
EA	Evolutionary Algorithm
EHW	Evolvable Hardware
EORA	Evolution-Oriented Reconfigurable Architecture
ES	Evolutionary Strategy
FIFO	First-in-first-out
FPAA	Field Programmable Analog Array
FPGA	Field Programmable Gate Array
FPTA	Field Programmable Transistor Array
FSM	Finite State Machine
GU	Genetic Unit
HDL	Hardware Description Language
HW	Hardware
IC	Integrated Circuit
IP	Intellectual Property
LUT	Look-Up Table
mdpp	Mean Difference Per Pixel
MEMS	Microelectromechanical System
MOSFET	Metal-Oxide-Semiconductor Field Effect Transistor
NFL	No Free Lunch (theorem)
PLA	Programmable Logic Array
RAM	Random Access Memory
RTL	Register Transfer Level
SRAM	Static Random Access Memory
SW	Software
TA	Testability Analysis

TID	Total Ionizing Dose
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
VLSI	Very Large-Scale Integration

1. Introduction

1.1 The Scope of the Thesis

This habilitation thesis deals with the problem of *implementation* of computational devices. The process of implementation is traditionally understood as the top-down process converting an abstract specification of the computational system to a realization in a suitable physical medium, in a way that best satisfies the problem-specific constraints on performance, cost, power dissipation, testability and so on. The computational devices are realized either as *general-purpose* processors or *application-specific* devices. We will be interested in the second case in this thesis.

Our brains are physical devices, products of nature. Their behavior is often interpreted as computing [84]. In comparison with the ordinary computers, they differ in many features. One of them, especially important for this thesis, is their origin. Our brains were evolved; their increasing complexity has emerged as a consequence of selection pressure [20]. They are constructed in the process of development, in the course of cellular divisions and cellular differentiations starting from the zygote. In contrast, the ordinary computers are built according to abstract computational models introduced to satisfy the a priori given specification. The conventional process of implementation is well-developed and has achieved an amazing success. In the process of top-down approach, engineers are able to construct a chip consisting of 10^8 transistors; every of them perfectly controlled. However, it seems that traditional top-down design methods do not allow engineers to create systems above a certain complexity. Furthermore, it is very difficult to achieve self-adaptation and self-recovery using conventional techniques. This contrasts very strongly with the mechanisms which have produced the extraordinary diversity and sophistication of living creatures.

Getting inspiration from nature, *evolutionary algorithms* have been utilized to design hardware in the recent decade [42, 141]. *Evolvable hardware* is an emerging field that applies evolutionary algorithms to automate design and adaptation of physical, reconfigurable and morphable structures such as electronic systems, antennas, MEMS and robots. Figure 1.1 demonstrates the difference between the traditional implementation scheme and evolutionary design approach. In contrast to the conventional implementation, the evolutionary approach is based on the generate&test approach that modifies properties of the target physical system in order to obtain the required behavior. In particular, the most promising outcomes of the approach are: (1) artificial evolution can produce intrinsic designs for electronic circuits which lie outside the scope of conventional methods [122] and (2) the challenge of conventional design is replaced by that of designing an evolutionary process that automatically performs the design in our place. This may be harder than doing the design directly, but makes autonomy possible [115].

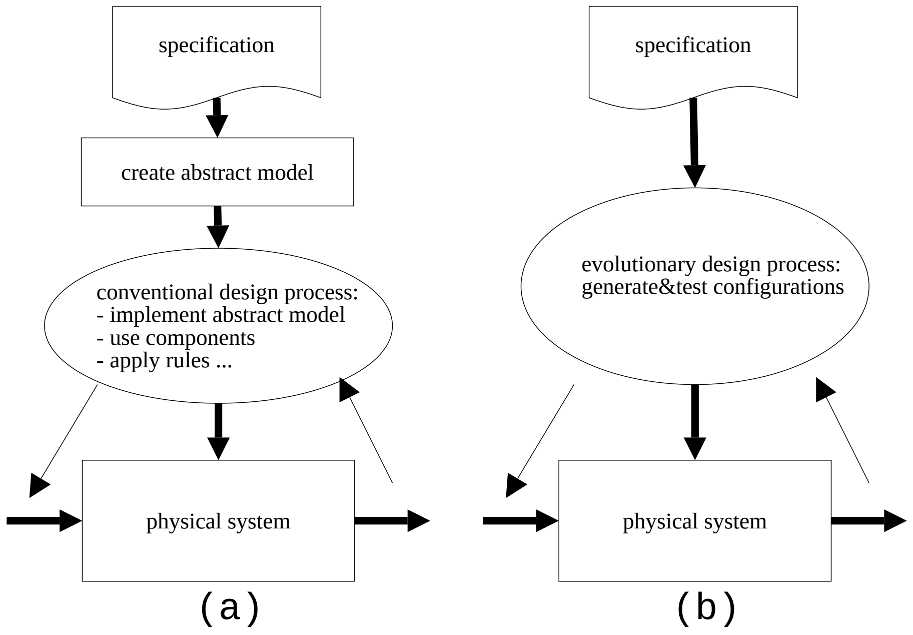


Fig. 1.1. Conventional design (a) vs evolutionary design (b)

1.2 Hypotheses and Goals

In this habilitation thesis, the evolutionary algorithms are utilized to design physical computational devices. All the problems investigated here should contribute to the explanation what it means for a computational system to be designed evolutionarily and what we can expect from the evolutionary approach that works on the position of human designer. We will present and investigate a class of computational devices of various complexity which were designated and implemented evolutionarily. Similarly to other classes of computational devices (e.g., abstract computing machines or processors), we should address fundamental questions such as what computational power these devices possess, whether they can be universal and what is limiting for their complexity. However, because of the specific nature of the process utilized to create them, we have to ask more basic questions: Are the evolved devices the computers in sense we usually understand the computers? Can we really obtain a computing device solving any computable problem we can imagine by means of evolutionary approach? In particular, we have formulated the following hypotheses and goals for this habilitation thesis:

1.2.1 Hypothesis H1: There is no significant relation between the complexity of evolved circuits, chromosome size and novelty obtained

Evolutionary design techniques have been applied at different levels of computational systems design (for example, at transistor level, gate level, functional level). There is no analysis indicating what complexity of resulting circuits can be obtained at different levels using accessible computers, what is the most limiting factor to evolve complex systems and whether innovative and human-competitive results[1] can be obtained at all levels. The objective is to perform an analysis of the relation between the size of chromosome, complexity of resulting

[1] Human-competitiveness will be defined in Chapter 3.1.2.

circuits and the level of innovation we can obtain. In order to answer these questions, we will perform experiments with the evolutionary design working at various levels in various problem domains. The evolutionary design will be performed using circuit simulators as well as digital and analog reconfigurable devices.

1.2.2 Hypothesis H2: It is possible to speed up digital circuit evolution for a reasonable class of applications using a common FPGA

It is important to minimize the time of evolution for almost all applications of evolvable hardware. Because it is practically impossible to quickly[2] reconfigure current FPGAs, it is difficult to use them for the evolutionary circuit design. The objective is to design, build and evaluate an FPGA-based evolvable hardware platform that can be utilized to speed up the evolutionary design process for a reasonably wide class of digital circuits.

1.2.3 Hypothesis H3: It is possible to evolve sequential circuits at reconfigurable platforms operating at sub-gate levels

Should the evolutionary approach be useful in the design of complex computational systems, it shall be able to produce sequential circuits. As many experiments are nowadays conducted at sub-gate levels (i.e. at transistor level, molecular level etc.), it is important to evolve sequential circuits at those levels. It is unknown whether the evolutionary approach can (re)discover the concept of the discrete state in a physical hardware which can be reconfigured at a sub-gate level. The objective is to show that it is possible using an FPTA.

1.2.4 Hypothesis H4: Evolution is able to form computational elements at extreme low temperatures

It is unknown whether the evolutionary approach can implement computational elements in extreme low temperatures. The objective is to show that it is possible.

1.2.5 Hypothesis H5: The embodiment can be utilized to introduce additional functions to digital modules in a non-traditional but manageable way

In the traditional paradigm of computing, the implementation does exactly the same as a corresponding abstract model. However, it was shown that some additional functionality can be introduced into a system if the system is composed of polymorphic gates. However, no polymorphic circuits more complicated than a single gate have been reported in literature yet. The objective is to show that evolutionary approach is able to find polymorphic modules that benefit from the "embodiment phenomenon[3]".

1.2.6 Hypothesis H6: Every evolved computational device is a computing mechanism

As the classical computation theory is philosophically and terminologically closely bounded to the theory of Turing machines, it is not clear whether the evolved computational devices

[2] The requirements on reconfiguration time will be defined in Chapter 5.
[3] an operation in a physical environment

(i.e. the results of the evolutionary design process) are *computing mechanisms* (in Turing's sense) and whether continual adaptation of an evolvable device can be simulated using standard models of computation such as Turing machines. We will analyze properties of evolved computational devices and attempt to fit them to the standard view of theoretical computer science.

1.3 Structure of the Thesis

The experimental results presented, further developed and analyzed in this thesis are based on my papers published in the recent four years[4]. My colleagues and students coauthored many of those papers. This subchapter describes the content of each chapter, indicates the papers utilized to write each chapter and credits to all the people that have contributed to the particular research.

Chapter 2 surveys what computing means traditionally and how this meaning has been shifted with the recent development in artificial intelligence, especially in the area of natural computing. It provides views of prominent theorists on the implementation problem and discusses the physical limits of computing.

In Chapter 3, the principles and basic terminology of evolutionary design and evolvable hardware are summarized.

Chapter 4 is devoted to the evolutionary design of digital circuits using simulators. Firstly, it explains why the evolutionary approach can discover novel implementations. Then it demonstrates that the evolutionary approach is able to produce human-competitive results at various levels of description of digital systems, including gate-level evolution, functional-level evolution, developmental techniques and some domain-specific representations. The gate-level evolution of seven-input circuits was performed by Zdeněk Vašíček [133]. The comparison of different encoding strategies for the median problem was adopted from [95]. We introduced the idea of evolutionary design of arbitrarily large sorting networks using development in [98]. Together with Michal Bidlo, we have improved this method and published the results in Genetic Programming and Evolvable Machines journal [99]. Furthermore, this work won the Honorable Mention Award at the Human-Competitive Results competition organized at the Genetic and Evolutionary Computation Conference GECCO 2005. In cooperation with our FIT group dealing with diagnostics of digital circuits, we have developed a method for the evolutionary functional-level design of benchmark circuits with predefined testability properties. The results presented here are adopted from Pečenka's et al. paper [83]. It is interesting that the evolved circuits are probably the largest circuits designed by means of an evolutionary technique ever.

In Chapter 5, the three applications of digital evolvable hardware are presented that are completely implemented on a single chip using the approach proposed in [92, 94]. An evolvable combinational unit was implemented and evaluated by Štěpán Friedl [96], evolvable image filter by Tomáš Martínek [75] and evolvable sorting networks by Jan Kořenek [57]. In particular, these implementations show that evolvable hardware can be implemented in a common FPGA and that the obtained speed up and cost are reasonable.

Chapter 6 is based on experiments performed at the NASA Jet Propulsion Laboratory (JPL) during my Fulbright project in 2004. It shows the approach utilized to discover elementary sequential circuits in a reconfigurable transistor array [101]. In order to demonstrate that

[4] Most papers are available from my website: http://www.fit.vutbr.cz/~sekanina

the evolutionary techniques are able to explore properties of a given platform and environment to find a suitable solution, this chapter presents the results obtained from experiments performed in extreme low temperatures (at –196.5°C). The preliminary results are adopted from [142].

Chapter 7 is devoted to polymorphic electronics—a technology discovered during experiments with evolvable hardware in extreme environments at JPL. The chapter presents multifunctional modules composed of polymorphic gates that were evolved and published in [102, 103]. The work [102] won the best paper award at the 2nd European Workshop on Evolutionary Computation in Hardware Optimisation. Michal Bidlo has implemented and evaluated a developmental approach to the polymorphic circuit design [8].

In Chapter 8, the experimental results presented in previous chapters are analyzed and evaluated. The answers to hypotheses H1 and H6 are proposed. These results are utilized to characterize the class of computational devices designed by means of evolutionary algorithms.

Chapter 9 concludes this habilitation thesis and indicates open problems for future research.

2. Abstract Computational Machines and Physical Computational Devices

This chapter surveys what computing means traditionally and how this meaning has been shifted with the recent development in artificial intelligence, especially in the area of natural computing. We will also survey physical limits of computing, the limits of current technology utilized to implement computational devices and some alternative computational paradigms and devices.

2.1 Classical Computational Scenarios

The classical computational theory has its roots in Turing's work on the Entscheidungsproblem [130] (the decision problem for first order logic), one of the most important problems in mathematics at the time, proposed by David Hilbert. Turing introduced a novel abstract computational device, called *automatic machine* (a-machine). We now know a-machines as Turing machines. The Turing machine consists of a one-dimesional tape divided into an infinite number of cells, with one distinctive starting cell. The cells of the tape can contain any symbol from a finite *tape alphabet* Σ, or a symbol $\sqcup$, representing an empty cell. A *read–write head* is positioned at any moment of the discrete time on a cell. The machine is controlled using a *finite control unit* that is always in one of the following states: either one of a finite *set Q of non-terminating states* (containing the initial state q_0) or one of the *terminal states* from the set $H = \{\text{HALT, ACCEPT, REJECT}\}$. The machine implements a (partial) *transition function*:

$$\delta : Q \times \Sigma \to (Q \cup H) \times \Sigma \times \{\leftarrow, \downarrow, \to\}.$$

A single transition $\delta(q, x) = (q', x', d)$ means that if the machine is in state q and the head reads x, then the machine enters the state q', stores x' in the cell in which the head is currently on, and the head moves in the direction d, to the right if $d = \to$, to the left if $d = \leftarrow$, and does not move at all if $d = \downarrow$.

Informally, a *computation* of the Turing machine goes as follows. At the beginning of the computation the entire (finite) input data string is available on the input tape and the control unit is set to q_0. The computational steps are performed according to δ. If the machine terminates in an accepting state, the result of the computation is given by the content of the tape at this time. If the machine yields a terminating configuration with the state ACCEPT (REJECT or HALT, respectively), then the machine is said to *accept* the input string (*reject* the input string, or *terminate*, respectively). If a computation of the machine does not terminate, then we say that the machine *diverges* on the input string. Note that if the length of the tape were finite, the computation power of Turing machine is reduced to the power of a simple finite-state automaton.

The central role in computing plays the notion of *algorithm*—"a systematic procedure that produces, in a finite number of steps, the answer to a question or the solution to a problem"

[12]. We can observe that the control unit, in fact, encodes the *algorithm* which uses the tape as an input/output device and temporal memory. For a given machine, the same fixed algorithm is used *uniformly* for all inputs. If a new data input is processed, then all previous information is erased; no information can be carried over from the past to the present run of the Turing machine.

The existence of the so-called *universal Turing machine*—a specific Turing machine that can simulate any other Turing machine (if properly encoded)—is another important result by Turing [130]. This idea represents crucial theoretical foundation for current mainstream approach to the design of computational systems. Much of the credit for the spread of Turing's notion of universality into the engineering community belongs to von Neumann [84]. Designer's task is just to program a universal computer (e.g. processor) to solve a given problem; no particular knowledge about the processor hardware is needed.

Turing proved that there exist problems that are *undecidable* [130]. A problem is called undecidable if there is no algorithm (Turing machine) for deciding it. For instance, the *halting problem* is to decide, given a Turing machine M and input w, whether M halts on w. The halting problem is undecidable for a Turing machine [130]. In other words, the function which decides the halting problem is *uncomputable.*

It is possible to prove that (so-called) μ-recursive *functions* are computable using a Turing machine [36]. Any such the function can be represented in the form of algorithm. The input data on the tape represents the argument(s) of the function. The result is available on the tape at the end of the computation. So a link to Church's work on theory of computable functions has been established via Turing machines.

Initially unformal notion of algorithm was formalized using Turing machines. Both Turing and Church were interested in effective ways of computing functions, where "effectiveness" was a mathematical notion synonymous with "mechanical" and lacking formal definition. Later, it was shown that the computational power of Turing machines, recursive functions and λ-definable functions is equivalent. This result was taken as confirmation that the notion of effective function computation had finally been formally capcured. After its proponents, Kleene (1952) dubbed this the Church-Turing thesis:

"A function is effectively calculable if and only if it is Turing-computable."

While the Church-Turing thesis deals with effectiveness of computing functions over integers, it can easily be extended to strings; however, not to real-valued inputs, interactive computing or computing which requires infinite time or resouces. In order to capture another type of computing, Turing introduced an interactive machine, a choice machines (c-machines) "whose motion is only partially determined by the configuration. When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator" [130]. However, the concept remained unformalized, as probably unimportant for solving the Entscheidungsproblem. Turing machines were adopted in the 1960 as a complete model for algorithmics and computational problem solving and the two very important braches of theoretical computer science—computability and complexity—were built on them. Turing machines has served as a standard computational model to these days.

2.2 From Abstract Computing to Physical Computing

Theoretical computer science considers digital (i.e. Boolean) circuits as one of many models of computation. While the Turing machine is an example of uniform and infinite computer model (in the sense that each particular computer can process inputs of an arbitrary size), a single Boolean circuit computes only a single Boolean function. In order to make a universal computer model of the same power as Turing machines out of Boolean circuits, uniformly designed families of Boolean circuits have to be utilized [36]. Boolean circuits design belongs to the main techniques in contemporaty digital computers design. This technique, developed by von Neumann, has its origin in constructing nets of neurons proposed by McCulloch and Pitts [84]. Furtheromore, the connection drawn by McCulloch and Pitts, via their nets, between mental phenomena and computation had an effect on the way the Church-Turing thesis was interpreted. Under McCulloch and Pitts's theory, every net could be described as computing a function. So in the sense in which McCulloch and Pitts computed, and to the extent that McCulloch-Pitts nets were a good model of the brain, every neural activity was a computation. So computing was transformed from a specific human activity into all that brains did and Church-Turing thesis was turned from a thesis about what functions can effectively calculated into a thesis about the intrinsic limitations of brain (Piccinini has traced this connection in [84]).

Turing has offered his model for the study of the computations whose steps can be carried out by "a human being working mechanically with pencil and paper, unaided by machinery, and idealized to the extent of having available unlimited amounts of time, internal memory, and so forth" [18, 130]. Similarly to other models, its fundamental property is being independent from the physical. However, the Turing machine was not (only) a mathematical representation of a computing human, but literally an idealized mechanical device. Furthermore, according to Piccinini, Turing thought his machine could compute any function computable by machines [84].

The first who incorporated physically motivated mathematical constraints into a formal model of computation was Gandy [49] who attempted to capture in a precise mathematical framework the notion of computation by a *discrete deterministic mechanism* or device. Gandy imposes some conditions on the device: a bound to the speed of its signal propagation, a bound to its size and discreteness of its state space. Gandy proposed the Thesis M:

"Any function computable by a discrete deterministic device is effectively computable (and vice versa)."

Gandy have shown that the machine-computable functions, even in this broader sense, are simply the Turing-computable functions, thus adding a striking bit of evidence for the adequacy and stability of Turing's analysis. Other researchers have analyzed the Church-Turing thesis with respect to the physical implementation and in terms of *computing mechanism.* A computing mechanism is a mechanism whose proper function is to obtain certain output strings of tokens from certain input strings (and internal states), according to a general rule that applies to all inputs and outputs (for example, ordinary analog computers are not computing mechanisms because their inputs and outputs are real-valued quantities and not finite strings). For purposes of this habilitation thesis, we will mention two other variants of the Church-Turing thesis, Modest Physical and Bold Physical Church-Turing thesis, reflecting two different opinions on computational properties of physical computing mechanisms. According to Piccinini [84], Modest Physical Church-Turing thesis states that

"A function is computable by a mechanism if and only if it is Turing-computable."

Modest Physical Church-Turing thesis does not entail that everything is a computing mechanism. It only says that if something is a computing mechanism then the functions it computes are Turing computable. Modest Physical Church-Turing thesis is believed by most computability theorists and computer scientists [84]. On the other hand the Bold Physical Church-Turing thesis states:

"Any function whose values are generated by a physical system is Turing-computable."

Modest Physical Church-Turing thesis does not apply to all physical systems, but only to mechanisms that perform computations. It says that if a physical system computes, then it computes a Turing-computable function. Bold physical Church-Turing thesis applies to all physical systems. It is out of scope of this work to review all the views on the Church-Turing thesis and its variants; for a detailed analysis see Piccinini [84].

2.3 Hypercomputing

Many times the Turing thesis was re-formulated in this way: "Anything computable is Turing-machine computable" [18]. However, this interpretation is wrong. It is interesting and somewhat overlooked that Turing was the first person who considered computing beyond his model. The *Turing machine with oracle* (o-machine) supposes that it is possible to enter (potentially uncomputable) information into a computing Turing machine. While the computational mechanism of the Turing machine can operate with this information at any time, uncomputable problems can be solved in principle by the machine. Let A be a language. Technically speaking, a Turing machine with oracle A is a single-tape Turing machine with three special states $q_?$, q_y and q_n. The state $q_?$ is used to ask whether a string is in the set A. When the Turing machine enters state $q_?$, it requests an answer to the question: "Is the string to the right of the tape head in A?" The answer is supplied by having the state of the Turing machine change on the next move to one of two states q_y or q_n, depending on whether the answer is positive or negative. If A is a recursive set, then the oracle can be simulated by another Turing machine, and the set accepted by the Turing machine with oracle A is recursively enumerable. Otherwise, if A is not a recursive set and an oracle is available to supply the correct answer, the Turing machine with oracle A may accept a set that is not recursively enumerable [46]. Therefore, the halting problem is decidable by a Turing machine with oracle.

There are many other computational *models* provably more powerful than a standard Turing machine, for example, real-number computation, coupled Turing machines, accelerating Turing machines, various neural networks and site machine (see [17, 18, 66, 107, 137]). The research field dealing with computational models and systems more powerfull than Turing machine is called *hypercomputation* or *super-Turing computation*. Eberbach et al. summarizes three principles that allow us to derive computational models more expressive than Turing machines: interaction with the world, infinity of resources or evolution of system [24].

- Interactive computation involves interaction with an external world, or the environment of the computation, during the computation. In contrast to algorithmic problems (computed off-line), interactive problems have to concern with carrying out a computational task or service (computed on-line).

- Introducing an infinite initial configuration, infinite architecture, infinite time of computing (we do not assume that some computational processes, such as traffic light controllers, should stop) or infinite alphabet (i.e. infinite precision) yields computational models more powerful than Turing machines.
- By evolution of a system, we mean that hardware or software is upgraded by self-adaptive, learning or evolvable procedure during the operational time.

It is easy to extend an abstract computational device to obtain a super-Turing computational device. However, the question is whether *physical* computational devices can exhibit super-Turing computational power.

A *spurious hypercomputer* is a physical system, like a random process, whose output cannot be generated by a Turing machine. It is called spurious because an observer can not use it to compute arbitrary values of a previously specified function starting at an arbitrary time and place. Since spurious hypercomputers are not classified as computing mechanisms, they are irrelevenat to both Church-Turing and Modest Physical Church-Turing thesis. However, they are relevant to Bold Physical Church-Turing thesis.

A *genuine hypercomputer*, instead, is a physical system that (giving enough time and space) can be used by an observer to compute arbitrary values of least one previously specified function f that is not Turing-computable. A well known machine of this type should be Hogarth machine that exploits the properties of a special kind of spacetime, called Malament-Hogarth spacetime, which is physically possible in the sense of constituing a solution to Einstein's field equations for General Relativity. From a computation point of view, the idea is to use the infinity of certain discrete processes in finite physical time [136]. However, the validity of the model has not been experimentally verified.

According to Piccinini, Hogarth machines do not satisfy the definition of genuine hypercomputer, because they cannot be started at an arbitrary place and time [84]. Other proposed hypercomputers (such as accelerating Turing machines, real-valued artificial neural nets etc.) are similar to genuine hypercomputers, but their hypercomputational power derives from assuming infinite resources, which violates the intuitive notion of calculability by effective procedure. However, genuine hypercomputers are in conflict with Modest Physical Church-Turing thesis. To summarize, Church-Turing thesis does not rule out the possibility we construct a genuine hypercomputer. It merely rules out the possibility that a hypercomputer computes a function that is not Turing-computable by following an effective procedure.

There is another group of computational devices that is often counted among super-Turing computers. Those machines do not aspirate to calculate a priory defined non-computable functions; rather, they perform computations that do not follow the computational scenario of Turing machines. Hence, Turing machines cannot simulate them. Typically, non-uniform, interactive and infinite computations play a crucial role in that machines.

For instance, consider the *Driving Home from Work* problem presented in [24] that is uncomputable on a standard Turing machine, but computable in reality (with a driving agent): An automatic car has a task to drive across town from work to home. The output for this problem should be a time-series plot of signals to the car's control that enables it to perform this task autonomously. In the algorithmic scenario, all inputs (a map of the city etc.) must be available a priori. However, it is impossible to specify all inputs in reality; a behavior of pedestrians, weather etc. is unpredictable in general. Therefore, the task is not solvable algorithmically. Nevertheless, the problem is computable—interactively, with an online driving agent which is equipped with a camera, which analyzes situation by situation and drive the car. Mobile robotics and Internet are similar examples [66].

Van Leeuwen and Wiedermann have shown that such computations may be realized by an *interactive Turing machines with advice.* They have proposed the following extension of the Church–Turing thesis [66]:

"Any (non-uniform interactive) computation can be described in terms of interactive Turing machines with advice."

Interactive Turing machine with advice is a classical Turing machine endowed with three important features: advice function, interaction and infinity of operation. The advice function is to change the machine's control unit when it is necessary. The change may depend only on a given time moment and the change cannot be too large. If this should not be the case, one could in principle enter ready-made answers to any pending problems that are being solved into the machine. Hence, the size of the description of the new control unit depends only on the time t of the change, and the size of the description is at most polynomial in t. This machine interacts with an external world and it is not assumed that its computation will be treminated [66].

Van Leeuwen and Wiedermann discuss their thesis in [66]: "Do the results of the previous theorems mean that now we are able to solve some concrete, a priori given undecidable problems with them? The answer is negative. What we have shown is that some computational evolutionary processes, which by their very definition are of an interactive and unpredictable nature, can be modeled a posteriori by interactive Turing machines with advice. Yet none of these systems is seen as violating the Church–Turing thesis. This is because none of them fits the concept of a *finitely describable algorithm* that can be mechanically applied to data of arbitrary and potentially unbounded size."

The standard classical computation theory (including complexity and computability analysis) is philosophically and terminologically closely bounded to the theory of Turing machines. The recent development in the areas of artificial intelligence, molecular biology, computer engineering and some others together with considering concepts such us interaction, evolution over time, infinite computation and embodiment has led to a different understanding and explanation of computation. We will deal with these new views in Chapter 2.5.

2.4 The Implementation Problem

The Church-Turing thesis explains what it means to compute from a formal-mathematical point of view. But what exactly is the relationship between the formal structure and the physical system? What is for a physical system to be a computing mechanism? The process of building a physical computational device according to a formal model is called *implementation.* This section deals with *the implementation problem.* This problem is closely related to another fundamental issue: What is a computing mechanism and, more generally, computation? In the following subsections, viewpoints of prominent scientists are presented. Unfortunately, standard textbooks on theoretical computer science do not usually address these issues at all.

2.4.1 H. Putnam

According to Putnam, a system is a computing mechanism if and only if there is a mapping between a computational description and a physical description of the system [84, 88]. By computational description, Putnam means a formal description of the kind used in computability theory, such as a Turing Machine or a finite state machine. Putnam puts no constraints on how to find the mapping between the computational and the physical description, allowing any computationally identified state to map onto any physically identified state. Therefore, every ordinary open system is the realization of every abstract finite automaton. The following constructions, explaining Putnam position, are due Scheutz [88]:

A physical system S (described in a theory P) realizes n computational steps of a FSM (without input and output) within a given interval I of real-time if there exists a 1-1 mapping f from automata state types onto physical state types of S and a division into n subintervals such that for all automata states q, p the following holds: if $q \to p$ is a transition in the automaton from the k-th to the $(k+1)$-th computational step $(1 \leq k < n)$ and S is in state $f(q)$ during the k-th subinterval, then this will "cause" S to change into state $f(p)$ in the $(k+1)$-th subinterval.

Given this definition, Scheutz has interpreted Putnam's result in this way: There exists a theory P such that for every ordinary open system S, for every finite state machine M (without input and output), for every number n of computational steps of the automaton M, and for every real-time interval I (divisible into n subintervals) S (described in P) realizes n computational steps of M within I .

Scheutz argues: "There are basically two critical points if a direct mapping between abstract and physical states is to be established: either the mapping itself or the formation of physical states/state types might not be constrained enough (or both). A theory of implementation should provide necessary and sufficient criteria to determine whether a class of computations is implemented by a class of physical systems (described at a given level), otherwise the term 'implementation' is not appropriate" [88].

2.4.2 J. Copeland

For Copeland, to compute is to execute an algorithm [16]. More precisely, to say that a device or organ computes is to say that there exists a modeling relationship of a certain kind between it and a formal specification of an algorithm and supporting architecture. Then (physical) entity e is computing function f if and only if there exist a labeling scheme L and a formal specification SPEC (of an architecture and an algorithm specific to the architecture that takes arguments of f as inputs and delivers values of f as outputs) such that $< e, L >$ is an "honest" model of SPEC. To describe a physical entity as a computing machine of a certain kind is to envisage being able to predict aspects of its physical behavior on the basis of its architecture-algorithmic specification and its labeling scheme. He suggested two necessary conditions for honesty: (1) the labeling scheme must not be *ex post facto* (from a thing done afterward) and (2) the interpretation associated with the model must secure the truth of appropriate counterfactuals concerning the machine's behaviors.

2.4.3 M. Scheutz

Scheutz criticizes Putnam results and argues that it is better to start in the concrete, in the physical, and not in the abstract: take a physical theory P and consider its (simple and complex) objects together with their properties. He shows how input, output, and behavior of a physical system as described by a physical theory can be used to distill a (matter-independent) mathematical mapping f—the function realized by S—between inputs and outputs of S by abstracting over every physical dimension. This abstraction is not arbitrary, but determined by P. In other words, it means to obey the laws of physics that are described by that function. His construction is as follows:

A function f with domain D and range R is realized by a physical system S (describable in a theory P) if and only if the following conditions hold: (1) there exists a (syntactic) isomorphic mapping I from the 'input domain' of S to D; (2) there exists a (syntactic) isomorphic mapping O from the 'output domain' of S to R; (3) there exists a function F that describes the physical property (i.e. behavior) of S for the given input-output properties (i.e., F is a mapping from the 'input domain' of S to its 'output domain' described in the language and by the laws of P) such that for all $x \in D$ the following holds: $O(F(I^{-1}(x))) = f(x)$.

The approach developed by Scheutz differs significantly from the "state-to-state correspondence view": it does not require a notion of physical state, but determines directly the function which is realized by a physical system. Furthermore, it does not have to assume a particular computational formalism (to which the physical system exhibits a state-to-state correspondence), but can be related to computations described by any computational formalism via the function that these computations give rise.

2.4.4 R. Cummins

According to Cummins[1], a system is a computing mechanism if and only if it has inputs and outputs that can be given a systematic semantic interpretation and the system's input-output behavior is the "execution" of a program that yields the interpretation of the system's outputs given an interpretation of the system's inputs. For Cummins, a process is the "execution" of a program if and only if it is made up of sub-processes each of which instantiates one of the steps described by the program. A possible objection (see [84]) is that many connectionist computing mechanisms are not analyzable into sub-processes corresponding to steps defined by a program in any obvious way. In fact, connectionist computing mechanisms have been used as examples that not all computation is algorithmic.

2.4.5 J. Fodor

According to Fodor[2], a system is a computing mechanism if and only if (1) it operates on symbols by responding to internal representations of rules and (2) the system's operations "respect" the semantic content of the symbols. For Fodor, a symbol is a semantically individuated token that can be manipulated by virtue of its formal (non-semantic) properties. At least in the case of natural systems like brains, for Fodor the content of symbols is a natural property that can be discovered in a way that is *observer-independent*.

2.4.6 G. Piccinini

Piccinini offers an account of what it is for a physical system to be a computing mechanism—a mechanism that performs computations [84]. A computing mechanism is any mechanism whose functional analysis ascribes it the function of generating output strings from input strings in accordance with a general rule that applies to all strings. The approach is based on existence of primitive components that must be identified; the functional explanation of

[1] See Piccinini's survey [84].
[2] Ibid.

specific physical realizations of primitive computing components is irrelevant. An optimal account of computing mechanisms should satisfy the following six desiderata:

1. Paradigmatic computing mechanisms compute. All paradigmatic examples of computing mechanisms, such as digital computers, calculators, both universal and non-universal Turing machines, finite state automata, and humans who perform mathematical calculations, compute.
2. Paradigmatic non-computing systems don't compute. All paradigmatic examples of non-computing mechanisms and systems, such as planetary systems, hurricanes, and stomachs, don't compute.
3. Computation is observer-independent. The computations performed by a mechanism can be identified independently of which observer is studying the mechanism. This is a desideratum for two reasons. First, it underwrites ordinary computer science and computer engineering, where different observers agree on which computations are performed by which mechanisms. Second, it underwrites a genuinely scientific computational theory of the brain (or mind), according to which the computations performed by the brain (or mind) can be empirically discovered.
4. Computations can go wrong. A good account of computing mechanisms should explain what it means for a computing mechanism to miscompute.
5. Some computing mechanisms are not computers. A good account of computing mechanisms should explain why the term is used in this restrictive way and what is special about computers.
6. Program execution is explanatory. A good account of computing mechanisms should say how appeals to program execution, and more generally to computation, explain the behavior of computing mechanisms. It should also say how program execution relates to the general notion of computation: whether they are the same or whether program execution is a species of computation, and if so, what is distinctive about it.

Piccinini argues that the proposed account of computing mechanisms "...allows us to formulate the question of whether a mechanism computes as an empirical hypothesis, to be decided by looking at the functional organization of the mechanism. It allows us to formulate a clear and useful taxonomy of computing mechanisms and compare their computing power".

2.4.7 T. Toffoli

In [125], Toffoli argues that in ordinary circumstances most computing machinery—mechanical, electronic, or biological—can quite confidently be recognized by its structural features. A conventional definition is: "Computation is the exercise of function composition in a context where the building blocks—the data storage, transport, and interaction primitives—(a) are finite objects and (b) exist in a finite number of types specified once and for all." Computation is a discipline in which to achieve novel functions one is not allowed to introduce novel types of components, but only to appropriately combine together more instances of components drawn from the same, fixed and finite, repertoire. Toffoli recognizes that the previous definition encompasses just about any kind of activity that displays a certain loose regularity or repetitiveness.

He sympathizes with the idea of viewing the entire physical world, governed as it is by uniform laws, as "one big computer." Thus he requires *intention* for the purpose of predicting or manipulating—in other words, for knowing or doing something. "This is what shall distinguish bona-fide computation from other intriguing function-composition phenomena such as weather patterns or stock-exchange cycles. The concept of computation must emerge as a natural, well-characterized, objective construct, recognizable by and useful to humans, Martians, and robots alike."

2.4.8 D. Deutsch

Deutsch's approach is very liberal, practically everything is a computing system. More precisely: "A computing machine is any physical system whose dynamical evolution takes it from one of a set of 'input' states to one of a set of 'output' states. The states are labelled in some canonical way, the machine is prepared in a state with a given input label and then, following some motion, the output state is measured. For a classical deterministic system, the measured output label is a definite function of the prepared input label; moreover the value of that label can in principle be measured by an outside observer and the machine is said to 'compute' the function f" [21].

Copeland pointed out that there is nothing about algorithm in the previous description, which is a problem for a serious account of computation [16].

2.4.9 J. Searle

Searle [89] starts explaining his point by recalling that on the standard textbook definition, computation is defined syntactically in terms of symbol manipulation. He argues that "...syntax and symbols are not defined in terms of physics. Syntax, in short, is not intrinsic to physics. This has the consequence that computation is not discovered in the physics, it is assigned to it. Certain physical phenomena are assigned or used or programmed or interpreted syntactically. Syntax and symbols are observer relative." On the question "Is the brain a digital computer?" he answers that this question is ill defined and does not a clear sense because "...you could not discover that the brain or anything else was intrinsically a digital computer, although you could assign a computational interpretation to it as you could to anything else." In case of ordinary computers it is reasonable to interpret the physics in both syntactical and semantic terms.

2.4.10 C. Johnson

According to Johnson, an important characteristic of computing is "...that symbols within the system have a consistent interpretation throughout the computation, or at least if they do not there is a component of the system which explains how the interpretation of the symbols changes as the computation progresses. That is, any external system which observes and/or initiates a computation must declare in advance how it is going to interpret those symbols. This seems to be a key characteristic of computing which can be applied to natural systems. If there is not a consistent allocation of symbols then transformations are meaningless. In particular, if we are completely free to assign any symbol to any meaning at any point in the computation then we can say that any transformation is doing any computing" [51].

2.4.11 Summary

In this subchapter we learned, surprisingly, that such the fundamental terms as computing and implementation of computing do not have a uniform interpretation. It seems that solution to the implementation problem is a question of belief.

2.5 Non-classical and Natural Computing

Nowadays, we are looking for inspiration in reality (biology, physics etc.). We are attempting to exploit features of reality to perform useful computation, not-necessary a super-Turing computation. Often, nature does it better, or at the very least differently and interestingly. Examining how the real world solves its computational problems provides inspirations for novel algorithms, for novel views of what constitutes a computation, and for novel computational paradigms [108]. Non-classical computing (in particular, natural computing) is a research area concerned with computing taking place in nature and with human-designed computing inspired by nature [72, 25].

It is important to distinguish between the computational model and the computational device that should perform computing. When we perform a real computation, we choose to interpret certain observations of the executing physical device as corresponding to aspects of the mathematical model. But the concrete physical device includes features not included in the abstract computational model [108]. We can choose to observe other features, such as the power drain, which might correspond to different computations than we traditionally observe. Focussing only on the abstract model leads to oversights. Intelligence via embodiment and bottom-up layering of functionality stressed in Chapters 4, 5 and 6 is considered as an alternative to top-down symbolic processing. Brooks' subsumption architecture, physical embodiment, has made great strides in intelligent robotics [13].

The following survey of key features of natural computing is based on MacLennan's paper [72]. Natural computation shifts the focus from the abstract deductive processes of the traditional theory of computation to the computational processes of embodied intelligence. Therefore, in natural computation we are mostly concerned with nongeneral algorithms, that is, algorithms designed to handle inputs of a specific, fixed size. One of the principal issues of natural computation is real-time response; furthermore, computation is not usually terminated. Natural computation must exhibit tolerance to noise, errors, faults, and damage, both internal to the system and external, in the environment. Most natural computation is continuously adaptive since the environment is continually changing. In natural computation, we are generally interested in 'good enough' answers rather than optimal solutions, which are usually a luxury that cannot be afforded in a demanding real-time environment. Indeed, broad (robust) suboptimal solutions are often preferable to better, tightly defined optima, since the latter are more brittle in the presence of noise and other sources of uncertainty. Finally, we must note that natural computations need not be interpretable. Pragmatics is primary; the computation is fulfilling some purpose for the agent. Semantics (interpretation) and syntax (well-formedness) are secondary. Universal computation—the ability to have a programmable universal Turing machine—is not central to natural computation. Natural computation systems are typically constructed from the interconnection of large numbers of special-purpose modules.

One important issue is a natural computation system's generalization ability and flexibility in response to novelty. Therefore, such systems must be able to discover pragmatically useful structure that can be a basis for reliable extrapolation. Given the hardware, how do we get the best results for the widest variety of inputs most quickly? The generality of natural computation algorithms derives from the procedures for fitting the process to the hardware and real-time constraints. We observe that the 'power' of natural computing is not defined in terms of the class of functions it can compute, nor in terms of numerical 'capacity' (number of memories, associations, etc. that can be stored). Rather, power is defined in terms of such factors as real-time response, flexibility, adaptability, and robustness. Some of these factors may be difficult to quantify or define formally (e.g., flexibility).

The Grand Challenge for computer science was formulated by group of scientists [108] as "to journey through the gateway event[3] obtained by breaking our current classical assumptions, and thereby develop a mature science of non-classical computation". The assumptions of classical computation were formulated and alternative paradigms were highlighted.

Many unconventional models of computing as well as physical unconventional computers were built and investigated [108, 124]. Toffoli summarizes in the first issue of Journal of Unconventionla Computing [125]: "The fundamental role of unconventional computing is to generate and support through their initial stages those variants—those experiments and explorations—that will then be evaluated by evolution's differential survival engine[4]".

2.6 Computational Devices and Physical Limits of Computing

A computation, whether it is performed by electronic machinery, a biological system such as the brain or the Universe, is a physical process. It is subject to the same questions that apply to other physical processes: How much energy must be expended to perform a particular computation? How long must it take? In other words, what are the physical limits of computing? We are looking for general laws that must govern all information processing, no matter how it is accomplished [30]. In addition to these fundamental limits, we have to take into account practical limits that are determined by the state-of-the-art of technology.

As most computational devices are implemented as electronic circuits, we will survey basic limits of current technology. Some alternative paradigms will be mentioned.

2.6.1 Fundamental Limits

Lloyd explored the ultimate physical limits of computation as determined by the speed of light c, the quantum scale $\hbar$ and the gravitational constant G. As an example, quantitative bounds are put to the computational power of an *ultimate laptop* with a mass of one kilogram confined to a volume of one liter. The speed at which a physical system can process information is limited by its energy and the amount of information that it can process is limited by the number of degrees of freedom it possesses. Lloyd showed that the ultimate laptop could store and process 10^{31} bits of information and capable of performing 10^{51} operations per second [69].

In context of current electronic devices, the main interest is put on device scaling and speed limitations on irreversible computing that are derived from the requirement of *least energy computation.* A computational system whose material realizations utilize electrons and energy barriers to represent and manipulate binary representations of state will serve as a computational model for these investigations. In order to identify fundamental limits, we will fellow Zhirnov et al. [144] in this chapter.

Two important parameters in the realization of digital systems are: 1) *device switching time* t and 2) *integration density* (the number of binary switches per cm^2) n. The *information throughput* (e.g., the maximum number of binary transitions per unit time $B = n/t$) has grown exponentially with time because of scaling. Since each binary transition requires energy E_{bit},

[3] i.e. a change to a system that leads to the possibility of huge increases in kinds and levels of complexity
[4] selection of the fittest

the total power dissipation growth is in proportion to the information throughput: $P = BE_{bit}$. In order to reduce power dissipation and concurrently to keep information throughput, E_{bit} must be decreased. The well-known minimum limit for E_{bit} is given by the Shannon-von Neumann-Landauer (SNL) expression for smallest energy to process a bit

$$E_{bit} \geq E_{SNL} = k_B T \ln 2 = 0.017\text{eV} \ (T = 300\text{K}) \tag{2.1}$$

where k_B is Boltzmann's constant and T is the temperature.

To construct a model for a computer operating at SNL limit at 300 K, the minimum size and switching time of binary switches can be estimated from the Heisenberg uncertainty relations

$$\Delta x \Delta p \geq \hbar \tag{2.2}$$

$$\Delta E \Delta t \geq \hbar. \tag{2.3}$$

From these equations, the minimum size x_{min} of a scaled computational element or switch, which operates at E_{SNL} is

$$x_{min} = \frac{\hbar}{\Delta p} = \frac{\hbar}{\sqrt{2m_e E_{bit}}} = 1.5 \text{ nm (T = 300K)} \tag{2.4}$$

where m_e is the electron mass.

This minimum size corresponds to a maximum integration density of switches

$$n_{max} = \frac{1}{x_{min}^2} = 4.4 \times 10^{13} \text{ devices/cm}^2. \tag{2.5}$$

The minimum switching time of the "least energy" switch is estimated as

$$t_{min} = \frac{\hbar}{\Delta E} = \frac{\hbar}{k_B T \ln 2} = 0.04 \text{ ps} \tag{2.6}$$

The power dissipation per unit area of this limit technology is given by

$$P = \frac{n_{max} E_{bit}}{t_{min}} = 3.7 \times 10^6 \text{ W/cm}^2 \tag{2.7}$$

If we let $E_{bit} = K * E_{SNL}, K > 1$, then the feature size computed from (Eq. 2.4) will be smaller and the switching time will be faster, but unfortunately, the power per unit area at maximum packing density increases as K^3.

Short-term predictions call for scaling of CMOS technology to the 22-nm node [50]. This technology node specifies NMOS transistors with a physical gate length of 9 nm and a CMOS IC technology with a real gate density of 1.5×10^9 gates/cm^2 and power dissipation of 93 W/cm^2 . The minimum-scaled computational switch discussed above has a minimum size only a factor of six less than the minimum size of the 22-nm node NMOS transistor (physical gate length of 9 nm), but has a device density 3×10^4 larger and a power density 5×10^4 larger than the short-term prediction.

Classical information processing technology always requires a physical carrier, usually electrons and holes. The first requirement to physical realization of any arbitrary switch is the creation of distinguishable states within a system of such material particles. The second requirement is the capability for a conditional change of state. The properties of distinguishability and conditional change of state are two fundamental and essential properties of a material subsystem that represents binary information. These properties are usually obtained by creating *energy barriers* in a material system.

The location of the electron is said to be "distinguishable" if there is a very low probability of spontaneous transition to the alternate *well.* If, on the other hand, for the election in a given state, the probability of spontaneous transition to the alternate state is equal to 0.5, we say that distinguishability is lost. Using a more complicated model considering the barrier width a [144], the energy per switching of operation is estimated corresponding to the barrier height

$$E_b^{min} \approx k_B \mathrm{T} \ln 2 + \frac{\hbar^2 (\ln 2)^2}{8ma^2}. \tag{2.8}$$

It is seen that for $a > 5$ nm, the expression is a valid representation of minimum energy per switch operation, while for $a < 5$ nm, the minimum switching energy can be considerably larger. Suppose that there are n devices in a unit area A operating at a switching frequency f. Let P_{max} denote maximum thermal power that can be removed from the chip per unit area. Then, according to [144],

$$P_{max} \approx nE_b f \geq \frac{nf}{A}(k_B T \ln 2 + \frac{\hbar^2 (\ln 2)^2}{8m_e a^2}) \tag{2.9}$$

and tradeoffs between a and f are implied by the physical limit on *heat removal capability.*

According to the analysis, Zhirnov et al. has drawn two conclusions: (1) Any binary switch, operating at lower bound of energy, and in a maximum density configuration, fundamentally is limited in size to a critical dimension of ≈ 1 nm. This critical dimension is less than 10 times smaller than the critical dimension (7 nm) of an end-of-the-roadmap MOSFET. (2) It is clear from the above analysis that scaling for binary switches, packed to maximum density, is ultimately limited by the system capability to remove heat.

The 30-year-long trend in microelectronics has been to increase both speed and density by scaling of device components (e.g., CMOS switch). However, this trend will end as we approach the energy barrier due to limits of heat removal capacity. For nanoelectronics, this result implies that an increase in device density will require a sacrifice, due to power consideration, in operational speed, and vice versa. Thus, it appears that we are entering a regime where tradeoffs are required between speed and density, quite in contrast to the traditional simultaneous benefits in speed and density from conventional scaling.

In addition to fundamental limits, Meindl [76] considered several kinds of other limits on VLSI:

- Material limits (e.g. carrier mobility, ...)
- Device limits (e.g. minimum channel length, ...)
- Circuit limits (e.g. minimum allowable supply voltage, standby power drain, ...)
- System limits (e.g. the architecture of a chip, heat removal, ...)

In all previous categories, Meindl distinguished theoretical and practical limits. Theoretical limits are derived from the basic physical principles of thermodynamics, quantum mechanics and electromagnetism. Practical limits are summarized by Moore's law; they must take account of manufacturing costs and markets. Moore's law, the trend of exponential improvement in computer speed and cost-efficiency, has been mainatained over the last half-century. Moore's law continues to hold heading toward 10^9 transistors on a chip. However, the semiconductor industry is well aware of a variety of potential obstacles to further improvements of its technology over the next 10 years. Table 2.1 summarizes basic predictions for IC technologies (adopted from International Technology Roadmap for Semiconductors 2004 update [50]):

Table 2.1. Prediction of VLSI chip features for 2010 and 2018

Future VLSI chip	2010	2018
Physical gate length	18 nm	7 nm
On-chip local clock	15 GHz	53 GHz
Chip size	280 mm^2	280 mm^2
Transistors /cm^2	309M	1960M
Number of wiring levels	16	18

2.6.2 Conventional Approaches to Implementation

Contemporary computers are usually constructed as *universal computers* consisting of a controller, an arithmetic logic unit (ALU), a memory and an input/output unit interconnected by a bus system. Physical circuits of the computer are not usually modified during computer lifetime. The behavior of computer is determined by instructions and data stored in the memory. All the complicated circuits (note that Pentium IV processor consists of 42.10^6 transistors) are controlled by a relatively tiny set of instructions. The required behavior is obtained by composing the right sequence of instructions for a given problem. Problem solving using computers has been transformed to programming. The main problem of the approach is performance, i.e. how much time a single instruction requires to be executed and how many instructions we need to execute in order to solve a given problem. A number of approaches have been discovered to improve performance, including caching, pipelining, parallel processing, vector processing, multithreading, specialized arithmetic, etc. [41]. The obtained performance is sufficient for some problems. On the other hand, many problems cannot be solved on the universal computers effectively.

Creating an application specific hardware, i.e. a dedicated computer, can sometimes solve the problem of performance. The dedicated computer contains application-specific units and interconnecting systems that are not available in common processors. Considering the same microelectronic technology and operational frequency, the dedicated computer can be faster in several orders of magnitude than the universal computer for certain tasks [45]. The dedicated computers are produced as *application-specific integrated circuits* (ASIC). The realization cost is the main disadvantage of ASICs. Many pieces of these mostly “non-universal” chips have to be manufactured to make the production commercially attractive.

General-purpose *reconfigurable devices* like field programmable gate arrays (FPGAs) offer us other implementation options. The reconfigurable devices operate according to a configuration bitstream that is stored in the configuration memory. The cells of the configuration memory control a set of transistor switches. The switches define the way in which the programmable elements available on the device operate and the way in which are interconnected. By writing to this configuration memory, the user can physically create new (digital) electronic circuits. The advantage of the approach is that new hardware functionality is obtained through a simple reprogramming of the chip, similarly to reprogramming of the universal computer.

Contemporary FPGAs contain thousands of programmable elements. Furthermore, various additional circuits are integrated on the FPGAs, including RAMs, efficient multipliers and interfaces. For instance, the Xilinx FPGA Virtex II Pro contains four PowerPC processors on the chip [139]. Figure 2.1 shows the classic architecture of the FPGA, which is a two-dimensional array of reconfigurable resources that include reconfigurable cells (e.g. 16-bit look-up tables), reconfigurable interconnection network and reconfigurable I/O blocks. The FPGA vendors provide collections of tools that are utilized for designing circuits for FPGAs.

The designer describes the target-dedicated computer in a hardware description language (HDL), such as VHDL, Verilog, or HandelC. After simulations, a specialized program, a synthesizer, generates a netlist, i.e. an optimized gate-level implementation of the circuit. The next steps include mapping, placement, routing and creating the configuration bitstream for a given FPGA. The configuration bitstream is then uploaded into the configuration memory of the reconfigurable device. Although an FPGA-based solution usually requires around 100 times more silicon space than an ASIC solution for the implementation of the same behavior, and furthermore, the FPGA-based solution is usually ten times slower than the ASIC, FPGA can offer 10–100 times higher performance than conventional universal processors in a number of problems. In case that the configuration of FPGA is modified dynamically, during the run, we speak about *reconfigurable computing*. Higher performance is then achieved by (dynamically) building custom computational operators, pathways, and pipelines suited to specific properties of the task at hand [15, 62]. Novel hybrid high-performance architectures combine general-purpose processor(s) and reconfigurable device(s) and reconfigurable interconnection network on a single chip.

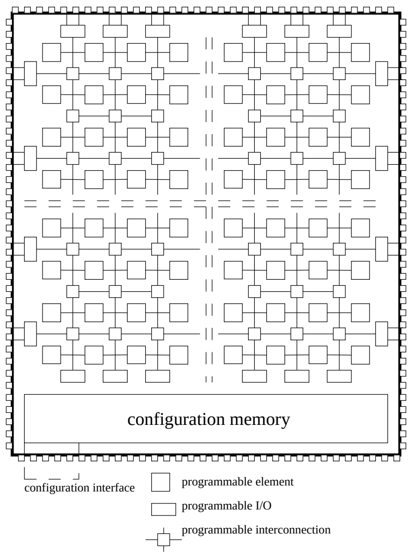


Fig. 2.1. Two-dimensional FPGA composed of programmable units. Its function is defined by uploading configuration bitstream into the configuration memory.

2.6.3 Reversible Circuits

To date, most microchips have been designed to dissipate the entire amount of electrical energy transferred during a binary switching transition. However, new approaches based on the second law of thermodynamics point the way to recycle switching energy by avoiding the erasure of information and switching under quasi-equilibrium conditions [30].

The first connection between computation and fundamental thermodynamics was made by John von Neumann. He reportedly performed a calculation of the thermodynamical minimum energy that is dissipated “per elementary act of information, that is, per elementary decision of a two-way alternative and per elementary transmittal of one unit of information”. He quantified this energy as $k_B T$ ln 2. Rolf Landauer was apparently the first person to explicitly state the argument establishing that the irreversible erasure of a bit of computational information inevitably requires the generation of a corresponding amount of physical entropy [63]. He recognized that reversible operations do not cause such dissipation. Fredkin and Toffoli had developed theory of reversible computing [124].

Logical reversibility means that logical states of the computation just prior to the operation is uniquely determined by its state after the operation. Logical reversibility does not bring so many advantages; however, benefits to be gained from using logical reversibility to enable another important kind of reversibility, *physical reversibility.* A physically reversible process is a process that dissipates no energy to heat, and produces no energy. It seems that absolutely prefect physical reversibility is technically unattainable in practice in a complex system [30].

Frank investigated ultimate models of computation that take full advantage of the computational power of the known laws of physics, while recognizing the limitations imposed by these physical laws, including the three-dimensionality of space, the finiteness of the speed of light and the second law of thermodynamics [30]. In this analysis, a reversible three-dimensional mesh of processing elements is evaluated as the best ultimate physically-based model of computation. Each such machine consists of a uniform three-dimensional array of processing elements, each of which has the capability of fully logically reversible and arbitrarily thermodynamically reversible operation. Each processor is connected locally to its nearest neighbors. This model also explicitly accounts with the non-zero entropy coefficient of each processing element and other physical limits on information processing. Frank conjecture is that this model scales better physically than any irreversible model of computation. He shows that only reversible computations are capable of realizing the maximum level of computational scalability that is afforded by the laws of physics.

2.6.4 Non-conventional Computers

Current paradigm to the design of computational devices is based on top-down approach. Tour claims that we need 15-20k electrons to store one bit of information. It is realistic to expect that the bottom-up design in molecular electronics will allow us to utilize 10^{14} computing elements per a squared centimeter. Potentially, 1–100 electrons will be needed to store one bit of information [128]. A number of non-conventional computing devices have been built. They represent challenges in many areas: speed of operation, density of data storage, parallel execution, low-power etc. Perhaps the most significant feature is that they provide totally different approach to problem solving than the classical processing of instructions offers. On the other hand, in some cases, their current variants exhibit a lot of undesirable features, such as low speed of operation, low density of data storage, noise and temperature sensitivity, difficult fabrication, cost etc. It is impossible to create a comprehensive list here; rather we will mention some typical examples[5]. Note that novel variants of these devices appear every day. However, they are not commercially attractive yet (as an exception we can mention quantum random number generators[6]).

- DNA computers [1];
- Molecular electronics (to replace VLSI by novel concepts such as carbon nanotubes, silicon nanowires, single-electron transistors, magnetic switches, helicene-based or fullerene-based devices [128, 11]);

[5] Theoretical models are irrelevant for this survey.
[6] http://www.idquantique.com/products/quantis.htm

- Synthetic biology (to modify genes of some organisms to perform computation, e.g. [26]);
- Quantum computing (to exploit quantum properties of reality to perform computation using e.g. NMR devices, trapped ions, optical lattices, superconductors ect. [11]);
- Liquid crystals [39];
- NanoCell [128];
- Reversible computers [30];
- Cellular automata-based machines (e.g. Cell Matrix [23], Embryonics [73] and POEtic [120]);
- and others as surveyed in [108] (reaction-difusion chemical systems, optical computing, analog computing, artificial neural network-based machines . . .).

3. Evolutionary Design and Evolvable Hardware

The aim of this chapter is to summarize the basic ideas and terminology related to evolutionary algorithms and evolvable hardware that will be needed in the following chapters. This brief introduction is based on [94].

3.1 Evolutionary Algorithms

Evolutionary algorithms (EAs) are stochastic search algorithms inspired by Darwin's theory of evolution. The *search space* is a space that contains all possible considered solutions to the problem, and a point in that space defines a solution. Instead of working with one solution at a time (as random search, hill climbing and other search techniques do [77]), these algorithms operate with the *population* of *candidate solutions* (individuals). Every new population is formed using genetically inspired operators (like *crossover* and *mutation*) and through a selection pressure, which guides the evolution towards better areas of the search space. The evolutionary algorithms receive this guidance by evaluating every candidate solution to define its *fitness value*. The fitness value, calculated by the *fitness function* Φ (i.e. objective function), indicates how well the solution fulfills the problem objective (specification).

The search is conducted on the *fitness landscape* of "hills and valleys", which is also called a response surface in that it indicates the response of the evaluation function to every possible trial solution. A higher fitness value implies a greater chance that an individual will live for a longer while and generate offspring, which inherit parental genetic information. This leads to the production of novel genetic information and so to novel solutions to the problem.

The fundamental structure of an evolutionary algorithm is captured in the following pseudocode:

Algorithm 3.1: Fundamental structure of an evolutionary algorithm

```
set time t = 0
create initial population P(t)
evaluate P(t)
WHILE (not termination condition) DO
  t = t + 1
  P(t) = create new population using P(t-1)
  evaluate P(t)
END WHILE
```

Because the objects in the search space can be arbitrary structures (e.g. real-valued vectors, circuits, buildings, etc.), it is often helpful to distinguish between the *search space* (i.e. solution or phenotype space) and the *representation space* (i.e. chromosome or genotype space). The encoded solutions (*genotypes*) must be mapped onto actual solutions (*phenotypes*). In general, a *representation* of a problem consists of a representation space and the *encoding* (*growth*) function. Genotypes (i.e. chromosomes[1]) consist of *genes.*

The fitness function is applied to evaluate phenotypes. While the fitness function operates with phenotypes, genetic operators are defined over genotypes. This concept of *genotype–phenotype* mapping has not been considered as crucial in the history of this field. It is probably due to the use of evolutionary algorithms as tools for optimization. However, recent studies (especially in the field of evolutionary design) assume this concept. There is one important rule about genotype–phenotype mappings: a small change in the genotype should produce a small change in the phenotype. If it does not hold, the algorithm is not efficient.

Any search strategy (and so any evolutionary algorithm) can be viewed as utilizing one or more *move* (i.e. genetic) operators, which produce new candidate solutions from those previously visited in the search space. Effective search strategies allow balancing two apparently conflicting goals: *exploiting* the best solution found so far and *exploring* the search space [77]. For instance, *hill climbing* techniques exploit the best available solution for possible improvement but neglect exploring a large portion of the search space. In contrast, a *random search*—where points are sampled with equal probabilities—explores the search space thoroughly but foregoes exploiting promising regions of the space. Each search space is different and even identical spaces can appear very different under different representations and evaluation functions. So there is no way to choose a single search method that can serve well in all cases [77].

This can be proved mathematically. The so-called *no free lunch theorem* (NFL theorem) states that if a sufficiently inclusive class of problems is considered, no search method (never resampling points of the search space) can outperform an enumeration [138]. The theorem says that over the ensemble of all representations of one space with another, all algorithms (in a rather broad class) perform identically on any reasonable measure of performance. This is a fundamental limitation of any search method. However, there are many possible improvements from a practical viewpoint. Primarily, it is *domain knowledge* that has to be employed in some way in the process of effective problem-specific algorithm design.

Traditionally, four main variants of the evolutionary algorithm are defined: genetic algorithm (GA), genetic programming (GP), evolutionary strategies (ES) and evolutionary programming (EP). These variants mainly differ in problem representation and genetic operators used. While GAs traditionally operate with a binary, integer, character or a real-valued vector stored in the chromosome of fixed length, genetic programming operates over executable structures (such as trees and instructions). For an overview of other features, see [4, 5, 33, 58].

3.1.1 Evolutionary Design and Optimization

In order to show how EA can be used we can utilize Bentley's classification [7], which defines four main categories of applications of evolutionary algorithms: evolutionary design optimization, creative evolutionary design, evolutionary art and evolutionary artificial life forms.

Evolutionary design optimization: In general, an optimization problem requires finding a set of free parameters of the system under consideration such that a certain quality criterion (fitness function here) is maximized (or minimized). Designers usually begin the process

[1] In biology, the terms *genotype* and *chromosome* have different meanings. However, the users of evolutionary algorithms do not usually distinguish between them.

with an existing design, and parameterize those parts of the design that they feel need improvement. The parameters are then encoded into chromosomes and an evolutionary algorithm tries to find the global or a sufficient local optimum. Evolutionary optimization places great emphasis upon finding a solution as close to the global optimum as possible. Genotype–phenotype mapping is not practically important. Optimization of the coefficient values of a digital filter or component placement on the chip are typical examples from the electrical engineering industry.

Creative evolutionary design: Bentley specifies what "creativity" means in the computer context [7]: "...the main feature that all creative evolutionary design systems have in common is the ability to generate entirely new designs starting from little or nothing and be guided purely by functional performance criteria. The emphasis is on the generation of novelty and originality, and not the production of globally optimal solutions. A computer is designing creatively when it explores the set of possible design state spaces in addition to exploring parameters within individual design spaces. Typically such systems are free to evolve any form capable of being represented." In terms of evolutionary algorithms, creative design is able to introduce new variables in the chromosome and so to define new search spaces. Among others, evolvable hardware falls into this category.

Evolutionary art: Evolutionary art is focused on creating new forms of images and art. The output from evolutionary art systems is usually attractive, but non-functional. A human evaluator determines the fitness, which is normally based on aesthetic appeal.

Evolutionary Artificial Life forms: This means the usage of evolutionary algorithms in the domain of Artificial Life.

3.1.2 Human-Competitive Results

John Koza, the inventor of genetic programming, has made an effort to achieve routine human-competitive machine intelligence using evolutionary techniques, and genetic programming especially. According to his classification, an automatically created result is "human-competitive" if it satisfies at least one of the eight following criteria [59, 60]:[2]

- (A) The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.
- (B) The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed scientific journal.
- (C) The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts.
- (D) The result is publishable in its own right as a new scientific result—independent of the fact that the result was mechanically created.
- (E) The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions.
- (F) The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.
- (G) The result solves a problem of indisputable difficulty in its field.

[2] Starting in 2004, the results generated by evolutionary techniques are annually evaluated by an international committee of experts at the "Humies" competition organized at the Genetic and Evolutionary Computation Conference. Every entry must be published in an open literature before it is submitted to the competition.

- (H) The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

Appendix provides the list of all the results that were considered as human-competitive. The results 1–36 are adopted from [60], the results 37–42 were presented at the Humies 2004 and the results 43–60 were presented at the Humies 2005. Table 3.1 summarizes how many times a certain criterion was claimed in the competition. We can observe that more than 23% of results satisfy the criterion A; on the other hand, only less than 3% of results satisfy the criterion C. We have identified nine main application areas of the human-competitive results (see Table 3.2). Most results were evaluated as human-competitive in the area of analog circuit design (25) and quantum circuit design (8). The reason is that we do not have such developed techniques for the design of analog circuits as, for example, for the design of digital circuits. The design of quantum circuits is not intuitive; hence EAs work well in these areas.

Table 3.1. The occurrence of criteria claimed in the entries to the Humies competitions

Criteria	A	B	C	D	E	F	G	H
[%]	23.31	10.53	2.26	15.04	11.28	18.05	16.52	3.01

Table 3.2. The number of human-competitive results awarded in a given area

Domain	# of human-competitive results
Analog circuit design	25
Quantum circuit design	8
Physics	7
Digital circuits/programs	5
Classical optimization	5
Game strategies	4
Chemistry	3
Antenna design	2
Mathematics	1

3.2 Evolvable Hardware

Previous section has shown that evolvable hardware is the field in which many human-competitive results were awarded. The idea of evolutionary hardware design was introduced at the beginning of nineties in papers of Higuchi and de Garis [42, 31]. Evolvable hardware is usually defined as an approach in which the evolutionary algorithm is utilized to search for a suitable configuration of a reconfigurable device in order to achieve the behavior required by specification [42]. Let us briefly introduce main components of evolvable hardware systems and basic terminology.

3.2.1 Reconfigurable Devices

The idea of *reconfigurable hardware*, initially introduced in programmable digital devices (PALs, FPGAs), has attracted attention in various other domains in the recent years. Field programmable analog arrays (FPAA) [29] and field programmable transistor arrays [110] have been developed in the analog circuit domain. In massively parallel reconfigurable processor arrays (such as picoChip PC102 [85]), the interconnection network can easily be reconfigured. Linden has utilized reconfigurable antennas [68]. Various reconfigurable implementations of neural networks are popular in the soft computing domain (see a survey in [145]). Reconfigurable nanodevices (e.g. NanoCell [128]), reconfigurable liquid crystals [39], reconfigurable microelectromechanical systems and reconfigurable optics represent other types of reconfigurable devices [115]. We can see that granularity of reconfigurable elements ranges from molecules to microprocessors. Some of these devices are devoted for a wide class of applications (e.g. FPGAs); some of them have been developed specifically for evolvable hardware ([65, 110, 39]). From the evolvable hardware point of view, the time and style of reconfiguration are very important parameters. In general, the reconfiguration time should be a fraction of the time which is needed to evaluate a candidate configuration; otherwise the evolutionary approach is inefficient. We should be able to reconfigure the device partially—which means that only a part of the device is configured while the other parts remain functional. Moreover, we usually require that every randomly generated configuration is legal and does not destroy the reconfigurable device.

3.2.2 Evolutionary Search

The chromosome represents either the configuration bitsream directly or a prescription determining how to create the configuration bitstream. A particular variant of evolutionary algorithm and genetic operators are chosen according to the problem to be solved. Suitability of the algorithm is evaluated experimentally. Candidate circuits are evaluated in the following way: First, a configuration bitstream is extracted from a chromosome. Then, the bitstream is uploaded into a reconfigurable device and the circuit created is evaluated using fitness function. In case of digital circuits, some training vectors are applied at the primary inputs, corresponding output responses are collected and compared against the desired vectors. The fitness value is usually determined as the number of bits calculated correctly by the candidate circuit. In case of analog circuits, training signals are applied at primary inputs for a given time period and circuit responses are compared with the desired signals. The objective here is to minimize the difference between the output signals and desired signals. Other approaches are possible, for example, frequency characteristic is measured. The evolutionary algorithm (genetic operators etc.) is usually executed in a personal computer because its time requirements are negligible in comparison with the evaluation time of a candidate circuit. However, its implementation as an ASIC or in an FPGA is also possible, especially in case that the complete system should be realized on a single chip.

3.2.3 Extrinsic and Intrinsic Evolvable Hardware

Extrinsic evolution means that candidate circuits are evaluated using a circuit simulator, i.e. in software. The configuration bitstream of a physical reconfigurable circuit is created only for the best solution at the end of the evolution. This approach ensures the existence of a symbolic representation of the resulting circuit and hence the evolved circuit can be implemented on various platforms. Although the fitness evaluation is usually slower when it is carried out in a software simulator, the method is typical for most research conducted in the field of evolvable hardware nowadays. As we do not have suitable quantum computers yet, quantum circuits must be simulated; therefore, the extrinsic evolution is the only approach to evolve quantum circuits.

Intrinsic evolution means that every candidate circuit (i.e. circuit configuration) in every population is evaluated in a physical reconfigurable circuit. It usually leads to faster fitness calculation (and thus faster evolution). Furthermore, much more efficient and innovative designs can be evolved because the evolved circuits work in a real environment.

Let us suppose that we have evolved a circuit configuration C_1 intrinsically that shows a perfect a fitness value x_1 in the reconfigurable circuit RC_1. Then the configuration C_1 is uploaded into RC_2 (which is exactly of the same type as RC_1), but contrary to our presumption (that we must obtain fitness x_1 because it is the same chip) we get a different fitness x_2, although the process of the fitness calculation remains formally unchanged! And finally, the same circuit (with configuration C_1) is evaluated using the software circuit simulator, and after this evaluation another fitness value x_3 is detected. How is it possible? Adrian Thompson has given the answer [121, 122]. The evolved circuit (with configuration C_1) requires some special properties that are provided only by RC_1 and/or by some type of environment (e.g. by the environment with constant temperature) to operate perfectly (i.e. with fitness x_1). This approach, sometimes called *unconstrained evolution*, refers to such an approach where evolution can also exploit the physical properties of a chip and other environmental characteristics (such as temperature, electromagnetic field etc.). The evolved circuits operate differently in different environments and they are not purely digital. This problem is known as the *portability problem* [109].

On the other hand, the *constrained evolution* employs such a representation and/or fitness function that intrinsic as well as extrinsic evaluation give the same results, i.e. the portability problem is eliminated. The representation and/or fitness function typically disallow the emergence of "too exotic circuits", e.g. asynchronous circuits, strange feedback, etc.

Stoica et al. [109] has developed *mixtrinsic evolution* to overcome the portability problem. Some individuals are evaluated intrinsically and some extrinsically within the same population. The idea was tested during an AND gate design using an FPTA. As a result, the evolved AND gates operated in the same way independently of the place where the evaluation procedure was carried out.

Recently, Miller has used the more general term, *evolvable matter*, to address the usage of evolutionary algorithms for the design on any physical reconfigurable platform (e.g. chemical) in a real environment. The idea behind the concept is that applied voltages may induce physical changes that interact in unexpected ways with other distant voltage-inducted configurations in a rich physical substrate. In other words, it should theoretically be possible to perform the evolution directly *in materio* if the platform is configurable in some way. Miller and Downing reviewed this promising technology in [80] and showed that liquid crystals can be used as a platform to evolve various circuits including primitive logic functions, robot controllers etc. [39, 40]. Tour's group has presented another example—the Nanocell [128]. The nanocell is a 2D network of self-assembled metallic particles connected by molecular switches. The nanocell is surrounded by a small number of lithographically defined access leads. The nanocell is not constructed as a specific logic gate—the logic is created in the nanocell by training it postfabrication by changing the states of the molecular switches. The training algorithm does not know the connections within the nanocell or the locations of the switches. However, the configuration can be changed by voltage pulses applied to the I/O pins. A genetic algorithm was utilized to generate these pulses in order to form the required logic gates.

3.2.4 Application Domains

It is important to distinguish between two approaches: evolutionary circuit design and evolvable hardware. In case of the *evolutionary circuit design*, the objective is to evolve (i.e. to design) a single circuit. The circuit is usually designed in a design lab. The aim is typically to find an innovative implementation of a required behavior and, for instance, to reduce the number of utilized gates or to ensure testability of a circuit. Up to now, the technique was successfully applied to design a number of unique digital as well as analog circuits. The evolved circuits exhibit properties that we have never reached by means of traditional engineering methods. It was demonstrated on small-scale-circuits that creative circuit designers can effectively be replaced by machines (see Appendix). Moreover, J. Lohn et al. have evolved an antenna which has won a competition organized by NASA to develop a new antenna for the Space Technology 5 Project (ST5) mission [71]. The two winners of Humies 2005 are very good examples how evolutionary algorithms can be utilized to modify properties of a physical device. The following statements are adopted from the competition materials:

- "Photonic crystals are nanoscale patterns of high and low index materials that can trap and manipulate electromagnetic radiation. They have a growing number of application in the photonic industry, but their design is challenging and unintuitive. We used GP to design photonic crystal structures with large photonic band gaps (ability to trap light). Starting from randomly generated photonic crystals, the algorithm yielded a photonic crystal with a band gap defined as the gap to midgap ratio as large as 0.3189. This band gap is an improvement of 12.5% over the best human design using the same index contrast platform" [87].
- "Bartels uses an evolutionary optimization algorithm to shape laser pulses to distort molecules in specific ways to catalyze chemical reactions, with the ultimate goal of manipulating large molecules, for example proteins and enzymes, in a biological cell. The method has also been used to create new quantum behaviors at the atomic level. The learning component is a relatively simple evolution strategy. Human can probe quantum systems, but are not capable of exploring different quantum behaviors in a fast automated fashion. In a sense, the results are beyond human competitive. There is also no doubt that this is a major scientific result which was achieved using evolutionary computation. The results are completely unexpected and amazing from a physical point of view: behaviors are being evolved that were not known to be physically possible. This includes anti-correlated attosecond harmonics in quantum systems. The ability to move beyond the nanoscale to the attoscale is a major breakthrough, and the potential applications of controlling the behavior of materials at atomic level are enormous" [6].

In case of *evolvable hardware*, the evolutionary algorithm is responsible for continual adaptation. Evolvable hardware was applied to high-performance and adaptive systems in which the problem specification is unknown beforehand and can vary in time [43, 115, 94]. Its main objective is the development of a new generation of hardware, self-configurable and evolvable, environment-aware, which can adaptively reconfigure to achieve optimal signal processing, survive and recover from faults and degradation, and improve its performance over lifetime of operation. Evolutionary algorithm is an integral part of the target system. The use of evolvable hardware to achieve adaptation was mainly demonstrated by Higuchi's group which developed a number of ASIC-based evolvable systems. Examples include: a prosthetic hand controller chip, image compression chip for electrographic printers, self-reconfigurable neural network chip, analog EHW chip for cellular phones and post-fabrication adjustment technology to maximize system performance [43, 53].

Due to the existence of redundancy in reconfigurable devices, the evolvable hardware is *inherently fault tolerant*. It means that in case that a circuit element is damaged, the evolutionary algorithm is usually able to recover the functionality using the remaining elements. If a critical number of elements are damaged, the functionality cannot be recovered and the chip "dies". Hence evolvable hardware is a method for automatic designing of adaptive as well as fault-tolerant (e.g. self-repairing) systems [122]. The use of evolvable hardware to achieve functional recovery was mainly demonstrated by Stoica's group at JPL which developed an FPTA and performed evolutionary functional recovery of analog circuits in extreme environments, such as extreme temperatures and radiation [113, 112, 142].

3.2.5 The Problem of Scale

Promises, challenges and problems of evolvable hardware have been summarized in [34, 94, 141, 140, 126, 105]. The *scalability problem* is traditionally considered as major problem of evolvable hardware: it is difficult to evolve complex, and simultaneously useful systems. As Torresen commented on classical paradigm of evolutionary circuit design, large objects require longer chromosomes, i.e. the search space is also larger and so difficult to be effectively explored by evolutionary algorithm [127]. A great effort has been invested to allow the evolutionary design to produces complex objects. Let us note that it is not a problem to evolve a large object at all. The problem is that the evolved large object is not usually interesting (innovative) because a lot of domain knowledge has been included into the design method. Three basic methods have been introduced to overcome the problem of scale, namely the functional level evolution [82], incremental evolution [127] and developmental approaches [37, 38]. These methods will be introduced together with the experiments reported in this thesis.

Another issue, closely connected with complex systems, is a very time consuming fitness calculation process. As digital circuits become larger and the number of inputs and outputs increase, the time needed for the fitness calculation grows exponentially. Furthermore, we require obtaining correct outputs for all reasonable operational conditions. It is impossible to specify all these conditions in the fitness function. In case of conventional approaches, engineers often manage such generalization by imposing strong biases on the nature of the circuit. Various design constrains (for instance, synchronous circuits are allowed only) can be specified to ensure a correct behavior of the circuit in all reasonable environments. An approach to overcome the problem of *verification* of the evolved design is to employ only a subset of the initial training set during the evaluation process. However, experiments performed show that it is practically impossible to evolve perfect circuits (i.e. the circuits producing correct outputs for every combination of inputs) if we simplify the evaluation process [48]. An issue that arises here is whether the maximum fitness value corresponds to the best generalization [140].

4. Extrinsic Evolution of Digital Circuits

Extrinsic evolvable hardware refers to the approach in which a circuit simulator is used instead of a physical reconfigurable device. This chapter deals with the evolutionary design of digital circuits using circuit simulators. These simulators were developed ad hoc. We will demonstrate that the evolutionary approach is able to produce human-competitive results at various levels of description. In particular, we will investigate gate-level evolution, functional-level evolution, developmental techniques and some domain-specific representations.

4.1 Transcending the Space of Logic Representations

Consider that we have to implement n-input combinational circuit. Traditionally, its function is specified using a truth table. Alternatively, we can express the function in some canonical form, for example, in sum-of-products or product-of-sums forms (with operators AND, OR and NOT), using NAND operators, NOR operators, in Reed-Muller form or using binary decision diagrams [134]. Considering the chosen system, the goal of logic synthesis is usually to represent a logic function in the simplest way by reducing the number of gates. In order to obtain a good result, we have to choose efficient representation of Boolean functions. The original expression is manipulated by applying canonical Boolean rules, Reed-Muller algebraic rules or other rules that are legal in the chosen system in order to obtain its simpler form. However, one never escapes from the space of logically correct representations that is determined by the chosen model. We have to realize that the space of all representations of a function of n variables is *much larger* than the space of all logically correct representations. Figure 4.1 shows that an evolutionary algorithm combined with assemble-and-test can be used to explore over a much larger area of design space than it is possible using a top-down rule-based design algorithm. Miller explains [78]: "The [conventional] methods though powerful in that they can handle large numbers of input variables are not adaptable to new logical building blocks and require a great deal of analytical work to produce small optimizations in the representation. Assembling a function from a number of component parts begins in the space of all representations and maps it into the space of all the truth tables with m input variables ($m \leq n$). The evolutionary algorithm then gradually pulls the specification of the circuit towards the target truth table shown as a small cross in Fig. 4.1. Thus the algorithm works in a much larger space of functions many of which do not represent the desired function. . . . this is the only way one can discover radically new designs".

Via the space of all representations we can access all components that are available, for example, adders, multipliers, buffers, etc. Experimental results have confirmed that we can transcend the scope of conventional engineering approaches to reach novel and innovative solutions to design problems. A number of novel implementations were discovered using this idea. This chapter provides some examples.

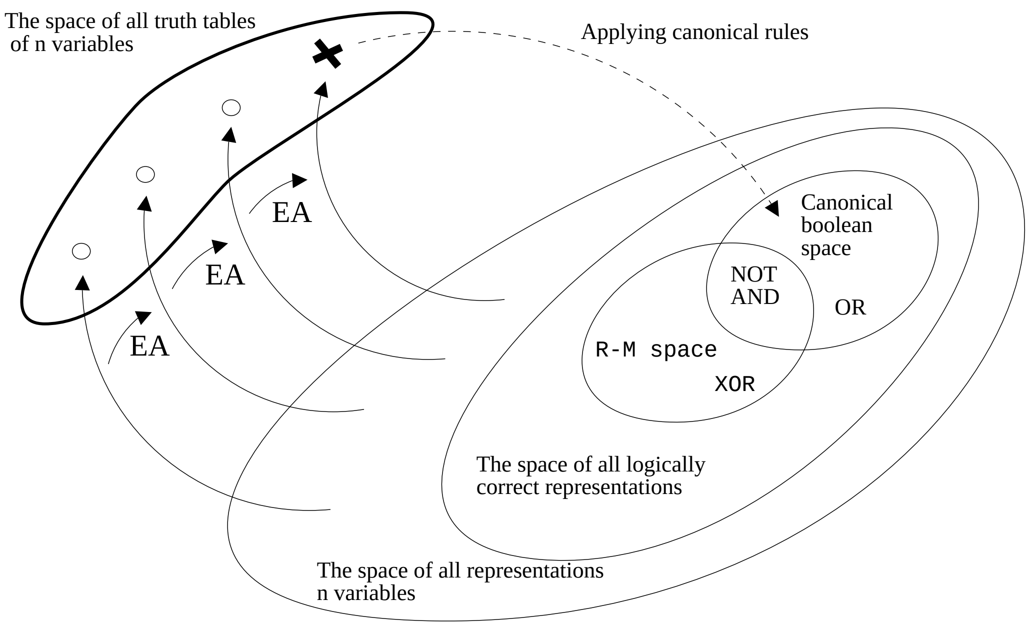

The space of all truth tables of n variables
Applying canonical rules
EA
EA
EA
Canonical boolean space
NOT AND
OR
R-M space
XOR
The space of all logically correct representations
The space of all representations n variables

Fig. 4.1. "Assemble-and-test" works by gradual improvement of the implementation by exploring unknown regions of the space of all representations. The conventional top-down approach is based on applying canonical rules (adopted from [78])

4.2 Gate-Level Evolution

Many novel circuit implementations were obtained using the gate-level evolution [78, 132]. Although this approach is not scalable, it offers us the possibility to explore novel design principles that are "unconstrained" by conventional hierarchical and modular design concepts. We are utilizing Cartesian Genetic Programming (CGP), a special variant of genetic programming developed for circuit evolution.

4.2.1 Cartesian Genetic Programming

Miller and Thomson introduced CGP that has recently been applied by several researchers especially for the evolutionary design of combinational circuits [79, 78]. In CGP, the reconfigurable circuit is modeled as an array of u (columns) $\times$ v (rows) of programmable elements (gates). The number of circuit inputs n_i and outputs n_o is fixed. Feedback is not allowed. Each gate input can be connected to the output of some gate placed in the previous columns or to some of circuit inputs. L-back parameter defines the level of connectivity and thus reduces/extends the search space. For example, if L=1 only neighboring columns may be connected; if $L = u$, the full connectivity is enabled. Each gate is programmed to perform one of functions, defined in the set Γ. Figure 4.2 shows an example and a corresponding chromosome. Every individual is encoded using $u \times v \times 3 + n_o$ integers.

The main advantage of CGP is that it uses the representation similar to real reconfigurable devices. However, the problem is that the corresponding search space is usually very rugged and thus difficult to search. Properties of CGP were analyzed in [78, 94]. Numerous experiments performed by Miller and at FIT have shown that a very simple evolutionary algorithm

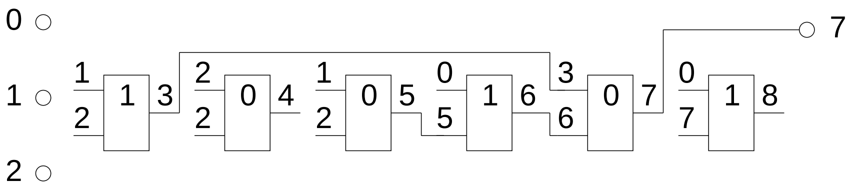


Fig. 4.2. An example of circuit encoding. CGP parameters are as follows: $L = 6$, $u = 6$, $v = 1$, functions AND (0) and OR (1). Gates 4 and 8 are not utilized. Chromosome: 1,2,1, 2,2,0, 1,2,0, 0,5,1 3,6,0, 0,7,1, 7. Three integers encode one gate. The last integer indicates the output of the circuit.

achieves the highest performance. The main features of the algorithm can be summarized as: EA operates with the population of λ individuals (typically, $\lambda = 5$). The initial population is randomly generated. Every new population consists of the best individual (i.e. elitism is ensured) and its mutants. The mutation operator modifies k randomly selected genes of an individual. In case that evolution has found a solution which produces the correct outputs for all possible input combinations, the number of gates is getting to minimize. Delay is not optimized. The evolution is stopped when no improvement has been observed during the last q generations or the maximum number of generations were exhausted.

4.2.2 Seven-Input Combinational Circuits

In order to illustrate CGP and compare the computational effort, we will deal with seven-input circuits in this section. In particular, the evolutionary design of 4×3-bit multiplier, 7-input sorting network and 7-input median network will be presented in this section. All the experiments were performed with the following parameters: $u \times v = 8 \times 7$ and 10×7, $\lambda = 5$, one gene is mutated per chromosome. The maximum number of generations is 20 million. The fitness value is defined as the number of output bits calculated correctly for all possible input combinations. When the maximal value is reached ($\Phi_m = n_o 2^{n_i}$) then the number of gates implementing the circuit is minimized using the formula $\Phi = \Phi_m + u \times v - g$ where g is the number of gates utilized by a given candidate circuit. The following functions are considered in programmable elements:

- 4×3-bit multiplier: $\Gamma = \{AND, OR, XOR, I\}$[1];
- 7-input sorting network: $\Gamma = \{AND, OR\}$;
- 7-input median network: $\Gamma = \{AND, OR\}$.

Next paragraphs present the evolved circuits and summarize the computational effort and other results of experiments.

4×3-bit multiplier. The 4×3-bit multiplier is considered here because of the same number of inputs as the other test problems and because its evolutionary design is more complicated than in case of the 3-bit multiplier (which is a popular benchmark circuit in evolvable hardware literature). As Figure 4.3 shows, the evolved circuit consists of 46 gates and has delay 7T (T is delay of a single gate)—thus reaching the same values as the best-know solution evolved by Vassilev [131].

Figure 4.4 shows that it is much easier to obtain significantly better solutions on the 10×7-element CGP than on the 8×7-element CGP. It confirms previous analysis indicating that a sufficient amount of redundant elements is necessary to evolve gate-efficient solutions [94].

[1] Symbol I denotes the identity operation.

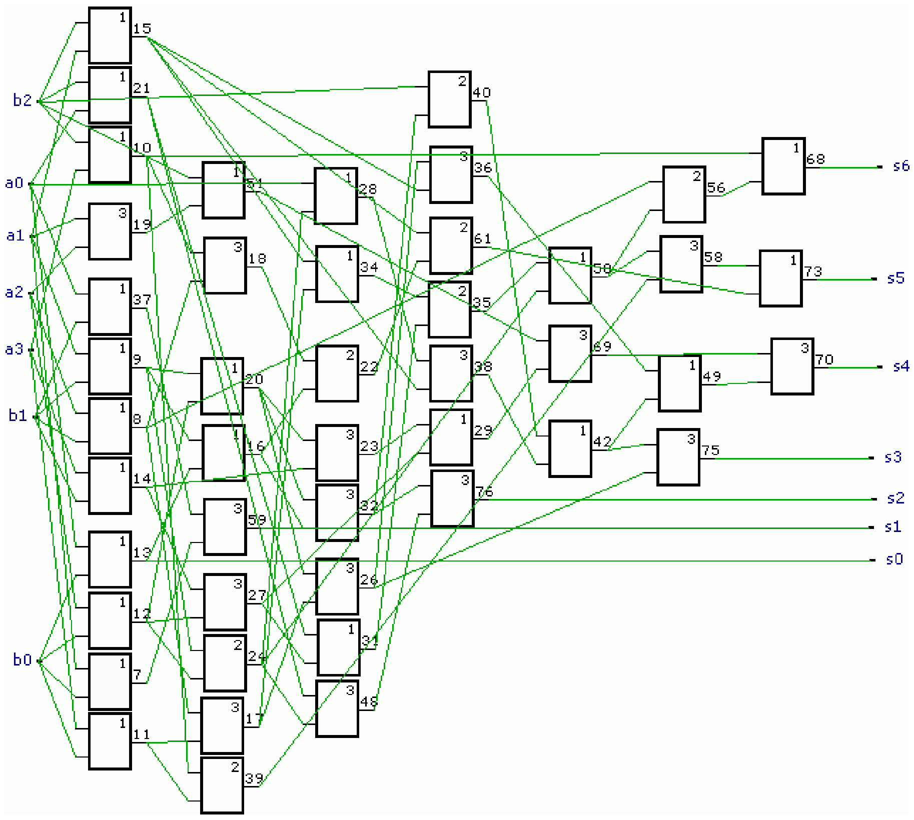


Fig. 4.3. One of the best evolved multipliers 4×3 bits (46 gates; delay 7T; functions used: AND (1), OR (2), XOR (3))

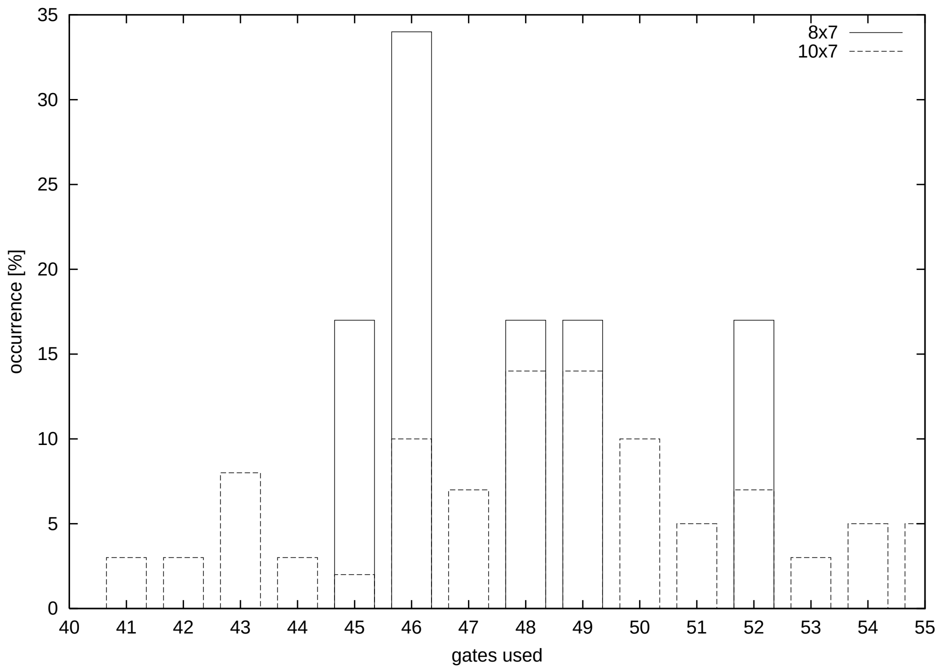


Fig. 4.4. The percentage occurrence of the multipliers that contain a given number of gates. Results are shown for the 10×7-element CGP and 8×7-element CGP.

Seven-input Sorting Network. A sorting network is defined as a sequence of compare–swap operations that depends only on the number of elements to be sorted, not on the values of the elements [55]. A *compare–swap* of two elements (a, b) compares and exchanges a and b so that we obtain $a \leq b$ after the operation. Although a standard sorting algorithm such as quicksort usually requires a lower number of compare operations than a sorting network, the advantage of the sorting network is that the sequence of comparisons is fixed. Thus it is suitable for parallel processing and hardware implementation, especially if the number of sorted elements is small. Figure 4.5 shows an example of a sorting network.

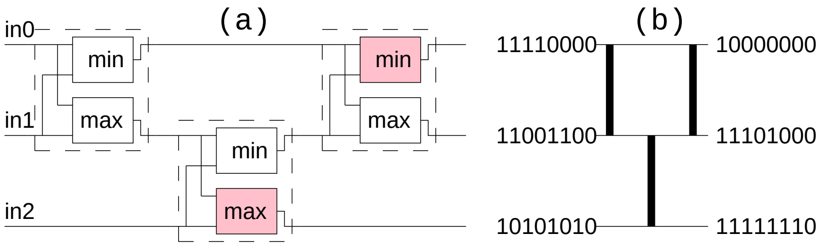


Fig. 4.5. (a) 3-input sorting network consists of 3 components, i.e. of 6 subcomponents (elements of maximum or minimum). 3-input median network consists of 4 subcomponents (white rectangles). (b) Alternative symbol; the complete test applied

Having a sorting network for N inputs, the *median* is simply the output value at the middle position (we are interested in odd N only in this thesis) [2]. Efficient calculation of the median value is important in image processing where median filters are widely used with $N = 3 \times 3$ or 5×5.

The number of compare–swap components and the delay are two crucial parameters of any sorting network. The following Table 4.1 shows the number of components of some of the best currently known sorting networks, i.e. those which require the least number of components for sorting N elements. Evolutionary approaches have been utilized to design optimal sorting networks, see, for example, [44, 52, 14].

Because we need the middle output value only in the case of the median implementation, we can omit some subcomponents (dead blocks at the output marked in gray in Fig. 4.5) and so to reduce the implementation cost in hardware. Hence if K components are needed to implement N-input sorting network then we can implement N-input median circuit with $2K - N + 1$ subcomponents (Table 4.1, line 3 with median*).

However, in addition to deriving median networks from sorting networks, specialized networks have been proposed to implement cheaper median networks. Table 4.1 (line 2) also presents the best-known numbers of subcomponents for optimal median networks. These values are derived from the table on page 226 of Knuth's book [55] and from papers [22, 56].

The *zero–one* principle helps with evaluating sorting networks (and median circuits as well). It states that if a sorting network with N inputs sorts all 2^N input sequences of 0's and 1's into nondecreasing order, it will sort any arbitrary sequence of N numbers into nondecreasing order [55]. This principle is used to make the evaluation of candidate sorting networks shorter.

Results: The evolutionary design of 7-input sorting network is much easier than in case of the multiplier. A perfect solution was found in most runs. Figure 4.6 shows one of the best

[2] In logic circuits, the median is often called "majority" function.

Table 4.1. Best known minimum-comparison sorting networks and median networks for some N. $c(N)$ denotes the number of compare–swap operations, $s(N)$ is the number of subcomponents. The last line holds for median networks derived from sorting networks using dead code elimination.

N	3	4	5	6	7	8	9	10	11	12	13	14	15	16	25
sortnet, $c(N)$	3	5	9	12	16	19	25	29	35	39	45	51	56	60	144
median, $s(N)$	4	-	10	-	20	-	30	-	42	-	> 52	-	> 66	-	174
median*, $s(N)$	4	-	14	-	26	-	42	-	60	-	78	-	98	-	264

solutions. It consists of 32 gates that were arranged to create 16 compare-swap components. It is much easier to find an optimal solution with L-back parameter set up to maximum (see Fig. 4.7). The proposed sorting network is superior to that presented by the inventors of sorting networks in their 1962 patent (O'Connor and Nelson 1962, U.S. Patent 3,029,413). Specifically, the 16-component seven-input sorting network has two fewer components than the sorting network described in the 1962 patent. In fact, CGP rediscovered what Floyd and Knuth [55] discovered several years after the 1962 patent, namely, that 16 components are sufficient for a sorting network for seven items. This result, also obtained by Koza who used a standard genetic programming [59], was considered as a human-competitive result satisfying criteria A and D (see Appendix, result 10).

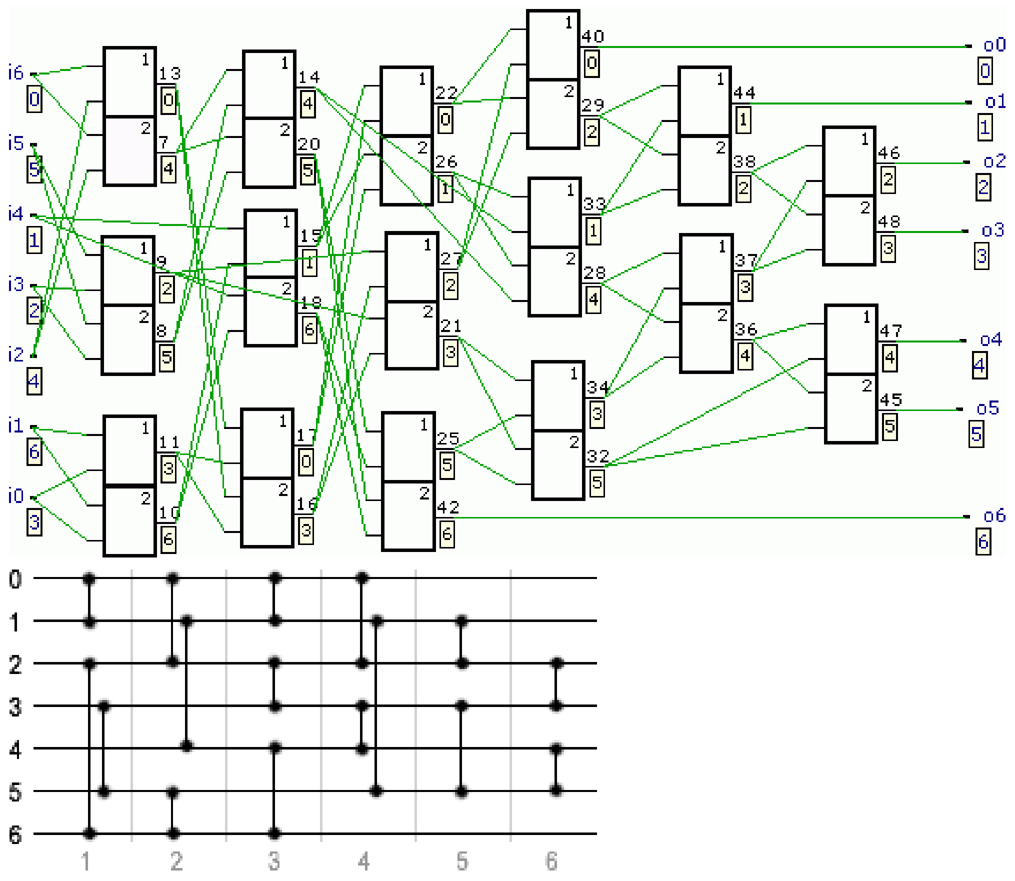


Fig. 4.6. One of the best evolved 7-input sorting networks (32 gates; delay 6T; functions used: MIN (1), MAX (2)) and its alternative representation.

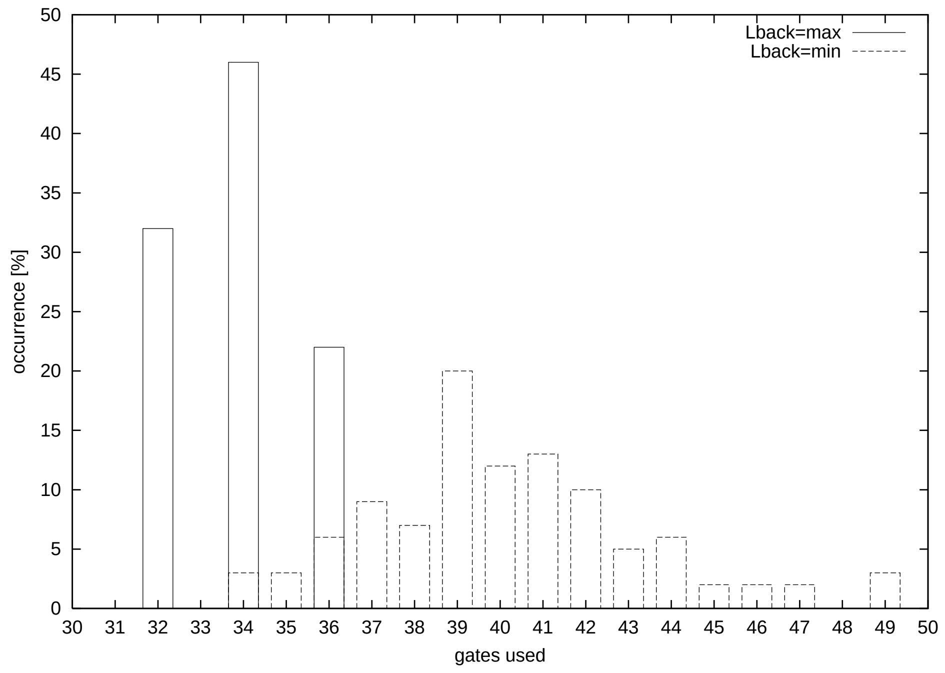


Fig. 4.7. The percentage occurrence of the sorting networks containing a given number of gates. Results are shown for the minimum and maximum values of L-back parameter of the 8 × 7-element CGP.

7-input Median Circuit. Figure 4.8 shows one of the best 7-input median circuits evolved using CGP. As this circuit has only one output, a perfect solution was obtained in almost all runs. This circuit contains the same number of gates as the best know solution. Figure 4.9 confirms again that it is much easier to obtain an optimal solution by using the maximum value of the L-back parameter than by using L-back= 1.

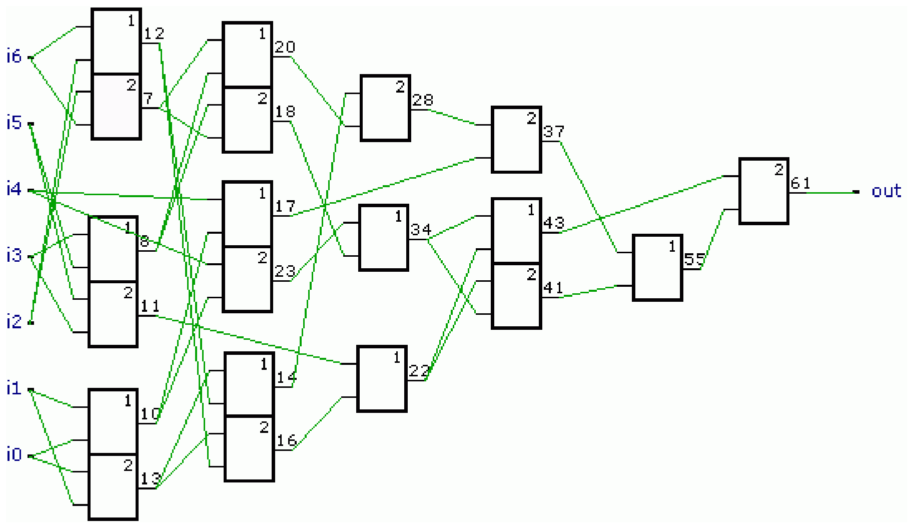


Fig. 4.8. One of the best evolved 7-input median circuits (20 gates; delay 6T; functions used: MIN (1), MAX (2)).

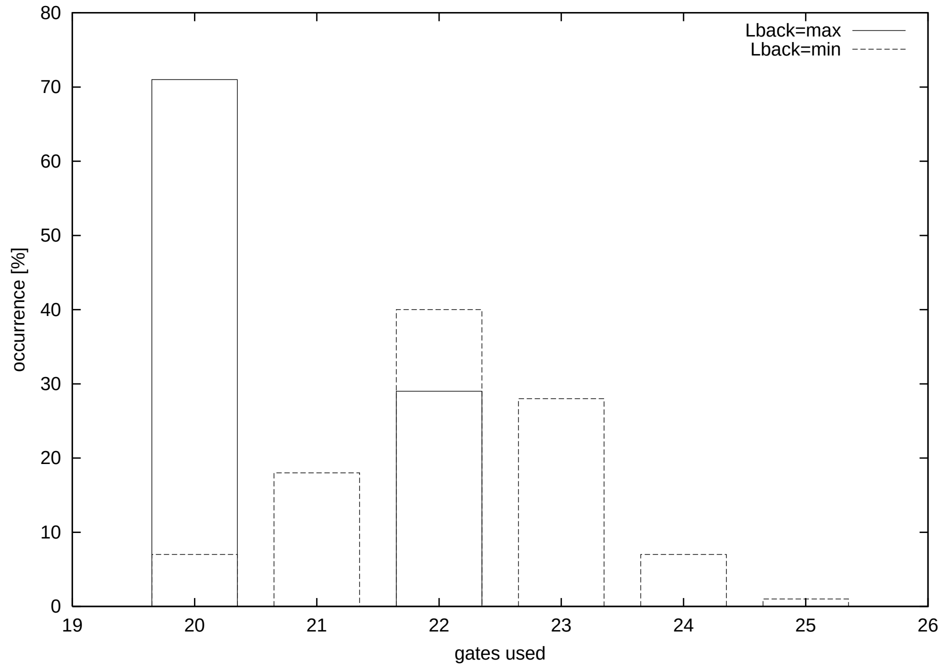


Fig. 4.9. The percentage occurrence of the median circuits containing a given number of gates. Results are shown for the minimum and maximum values of L-back parameter of the 8 × 7-element CGP.

Discussion. For all the three problems, human-competitive results were achieved. Table 4.2 summarizes the experiments performed using 8 × 7 and 10 × 7 programmable elements (100 runs for each setup). We utilized L-back parameter 1 and u. It can be seen that by setting L-back = u, the evolutionary process is more successful—the mean number of generations, the number of gates and delay are reduced. On the other hand, the problem is that the internal structure of circuits becomes more complicated and difficult to understand.

Table 4.2. Summary of results obtained from the evolutionary design of 4 × 3-bit multipliers, 7-input sorting networks and 7-input median circuits at 8 × 7-element CGP and 10 × 7-element CGP and with the L-back parameter set up to minimum and maximum. Averages are given for 100 independent runs (20g 6T denotes a circuit containing 20 gates and having delay 6T)

8x7	succ.	rate	avr.	gener.	gates		delay		best	solutions
L-back	1	max	1	max	1	max	1	max	1	max
Multiplier	0%	6%	–	11703k	–	45-52	–	7-8	–	48g 7T; 45g 8T
7-SN	69%	100%	1559k	534k	34-49	32-36	6-8	6-8	36g 6T	32g 6T
7-median	99%	98%	238k	275k	20-25	20,22	6-8	6-8	22g 6T; 20g 7T	20g 6T

10x7	succ.	rate	avr.	gener.	gates		delay		best	solutions
L-back	1	max	1	max	1	max	1	max	1	max
Multiplier	0%	59 %	–	9203k	–	41-55	–	7-10	–	46g 7T, 41g 9T
7-SN	87%	100%	1229k	390	36-53	32-38	7-10	6-9	36g 7T	32g 6T
7-median	99%	100%	234k	226k	20-25	20-22,28	6-9	6-9	22g 6T, 20g 7T	20g 6T

4.3 Alternative Representations

It is not typical to evolve sorting networks using CGP; simply because the approach is not scalable. The *compare-swap* (C&S) approach is usually applied in order to evolve larger sorting networks [14]. In comparison with CGP, we assume that the circuit is composed of compare-swap components, i.e. something is assumed about the solution a priori. Thus the search space is effectively reduced which in turn allows us to evolve larger sorting networks. Candidate solutions are represented as sequences of pairs $(a; b)$ indicating that a is compared/swapped with b. For example, the sorting network shown in Fig. 4.5 can be encoded as (0, 1)(1, 2)(0, 1). We compared CGP and the compare-swap approach in the median circuits evolutionary design problem. As median circuits can easily be derived from sorting networks, it seems that their evolutionary design is useless. However, we will show that interesting median circuits can be evolved from scratch.

4.3.1 CGP: Experimental Setup

As we are not going to follow the EA setup used in Chapter 4.2.2, we have to introduce CGP setup for this study. In order to evolve N-input median circuits, we utilized CGP with the following setting: $u = p$, $v = 1$, $L = p$, N inputs and a single output. p is the number of programmable elements depending on the problem size (for example, $p = 23$ for $N = 7$). Each of elements can be configured to operate as logic AND (minimum) or logic OR (maximum). Thus median circuits are constructed rather from subcomponents than from compare–swap components.

The evolutionary algorithm operates with population of 128 individuals. Four best individuals are considered as parents and every new population is generated as their clones. Elitism is supported. The initial population is generated randomly; however, the circuits with the maximum number of utilized elements are preferred. The evolution was typically stopped (1) when no improvement of the best fitness value occurs in the last 50k generations, or (2) after 600k generations. Mutation works as follows: either the circuit output is mutated with the probability 25% or one of the used elements is mutated—gate inputs are changed with the probability 85%, function is changed with the probability 15%. Only one mutation is performed per circuit; however, when the best fitness value stagnates for 10k generations, two mutations are employed.

During the fitness calculation, all possible input combinations are supplied at the circuit inputs (i.e. 2^N vectors). The fitness value of every candidate circuit is incremented by one if the circuit returns the correct median value for a given input vector.

4.3.2 CGP: Results

Table 4.3 shows that it is very difficult to evolve a perfect median circuit (with the fitness value 2^N) for more than 9 inputs. Furthermore, we were not able to reach optimal $s(N)$ for $N = 9$ and 11. The algorithm usually traps in local optima, which is very close to the perfect fitness value 2^N. It seems that the circuit evolution landscapes are very rugged.

Table 4.3 indicates that it is necessary to utilize much more programmable elements (the column denoted as p) than the resulting circuit requires in order to evolve a circuit with the perfect fitness. For instance, 50 elements were used to evolve the 36-element median circuit for $N = 9$. However, we performed 200 runs with $p = 43$ for $N = 9$, but no circuit has

Table 4.3. Summary of experiments performed to evolve small median circuits using CGP. N – # of inputs; p – # of columns in CGP; runs – # of runs performed; perfect – # of runs leading to the perfect fitness; best $s(N)$ – best obtained $s(N)$; best known $s(N)$

N	p	runs	perfect	best $s(N)$	best known $s(N)$
3	8	50	50	4	4
5	16	100	11	10	10
7	23	2000	22	20	20
9	50	2000	35	36	30
11	90	200	2	71	42
13	120	200	none	–	–

appeared with the perfect fitness 512. The best circuits evolved using CGP are depicted in Fig. 4.10.

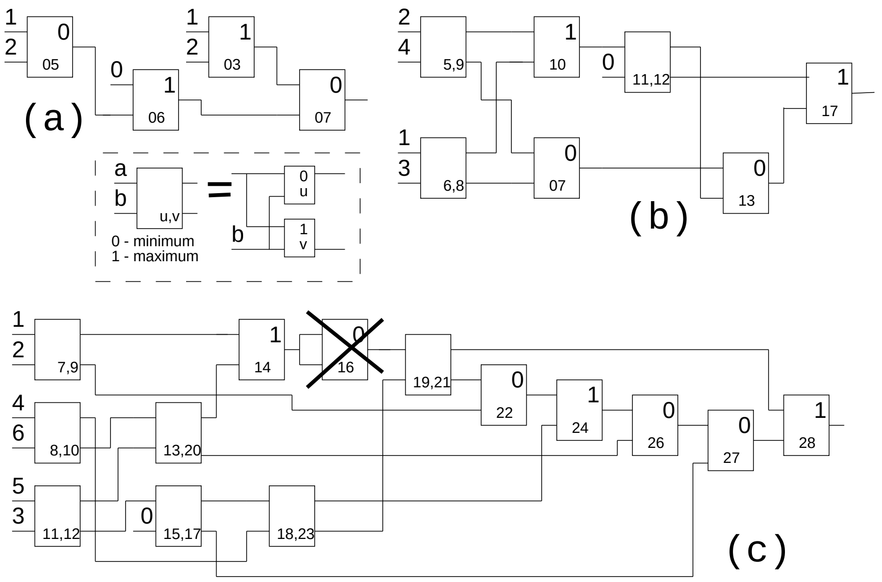


Fig. 4.10. Median circuits evolved using CGP: (a) 3-median, (b) 5-median and (c) 7-median circuits consist of the same number of subcomponents as the best–known conventional circuits. The evolved 7-median circuit contains one redundant subcomponent.

CGP has not allowed us to discover median circuits with lower $s(N)$ than the best known solutions exhibit. Furthermore, the approach produced median circuits only for small N. On the other hand, the obtained circuits are (1) totally different from the known median circuits that are based on the compare–swap approach and (2) much cheaper than those median circuits derived from classical sorting networks (see Table 4.1, line 3 with median*).

4.3.3 C&S: Experimental Setup

Each chromosome consists of a sequence of integers that represents a median circuit. We used variable-length chromosomes of maximum length ml; a sentinel indicates the end of valid sequence.

A typical setting of the evolutionary algorithm is as follows. Initial population of 200 individuals is seeded randomly using alleles $0 - (N - 1)$. New individuals are generated using mutation (1 integer per chromosome). Four best individuals are considered as parents and every newly formed population consists of their clones. The evolutionary algorithm is left running until a fully correct individual is found or 3000 generations are exhausted. We also increase mutation rate if no improvement is observable during the last 30 generations. Fitness calculation is performed in the same way as for CGP.

Because we would like to reduce the number of compare–swap components and because the fitness function does not consider the number of compare–swap operations, we utilized the following strategy. First, we defined a sufficient value ml for a given N (according to Table 4.1) and executed the evolutionary design. If a resulting circuit exhibits the perfect fitness, ml is decremented by 2, otherwise ml remains unchanged and the evolution is executed again. This is repeated until a predefined number of runs are exhausted. Because this approach works at the level of compare–swap components (i.e. $c(n)$), it was necessary to eliminate dead code to obtain $s(N)$.

4.3.4 C&S: Results

In contrary to CGP, perfectly operating median circuits were evolved up to $N = 25$ (see examples in Fig. 4.11). Table 4.4 shows that area-optimal circuits are up to $N = 11$. In spite of our best efforts, none specific values of $s(N)$ nor $c(N)$ were found in literature for median circuits with $N = 13 - 23$.

Table 4.4. The best median circuits evolved using the compare–swap approach. "U" denotes "Unknown".

N	3	5	7	9	11	13	15	17	19	21	23	25
$c(N)$	3	7	13	19	26	34	44	55	65	80	94	109
$s(N)$	4	10	20	30	42	56	74	94	112	140	166	194
best known $s(N)$	4	10	20	30	42	U	U	U	U	U	U	174

While evolved median circuits are area-optimal for $N = 3-11$ and perhaps close to optimal for $N = 13 - 19$, the evolved median circuits for $N = 21 - 25$ seem to be area wasting. The reason is that the best known value $s(25)$ is 174 [22]; however, we reached $s(25) = 194$. The evolution takes several days on a common PC for $N \geq 23$. Hence we tried to reduce the training set in our experiments; however, no solution has appeared producing correct outputs for all possible input combinations (the evolved solutions are not general).

4.3.5 Comparison of CGP and C&S for Medians

It was much easier to evolve perfect median circuits using the compare–swap encoding than using CGP. Similarly to the results obtained from CGP, the median circuits are not optimized for delay. We were able to evolve better median circuits than those median circuits that can be created by means of dead code elimination from the best-known sorting networks. In some cases (up to N=11), the evolved median circuits are area-optimal. CGP has been applied in order to obtain median circuits directly without the need to eliminate same gates. However, the approach is suitable only for small N.

4.3.6 Evolutionary Design Under Hardware Constraints

The proposed evolutionary approach allows designers to explore much larger portion of the design space than conventional methods. This is demonstrated on the following experiment.

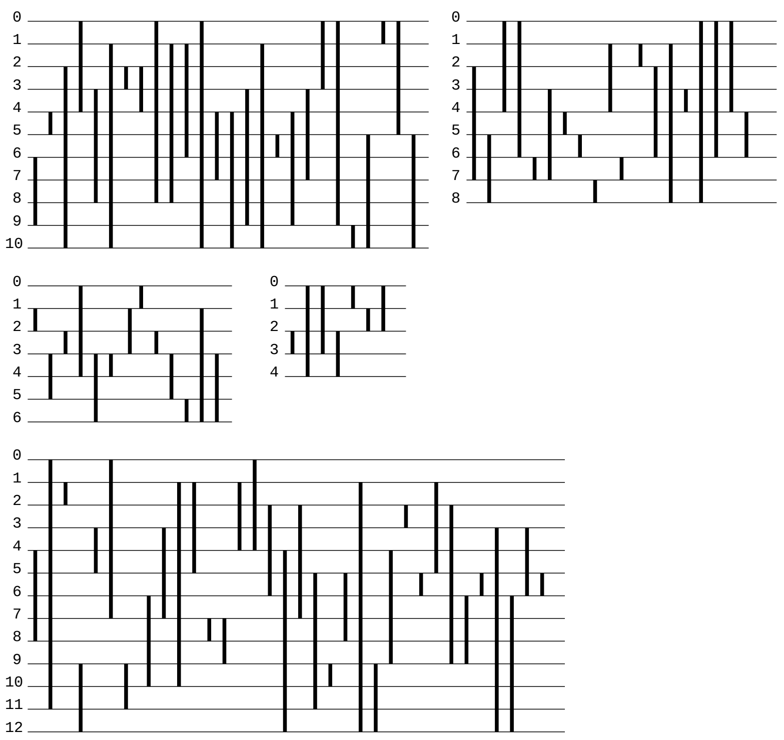


Fig. 4.11. Some of median circuits evolved using the compare–swap approach

The objective is to find out how many input combinations of all possible input combinations lead to wrong output values if the number of circuit elements (compare–swap operations) is continually reduced. It is useful to know this characteristic from the fault tolerance point of view or when only limited resources are available for the implementation.

In order to evolve median circuits under hardware constraints, we utilized the experimental background from Chapter 4.3.3. The 9-input median circuit was repeatedly evolved using 2, 3, . . . , 25 components. Figure 4.12 shows the obtained results: how many output (median) values remain correct if the number of utilized elements decreases. We can see that 19 elements still ensure the correct behavior (512). However, with decreasing resources the quality of the circuits gets down. It is interesting that the mere two elements yield 346 correct output values, i.e. more than 50%.

The method could be useful for designing area/energy-consumption efficient median filters for image preprocessing. In these applications, most input combinations should be processed correctly. Rare mistakes are not critical because our eye is not able to see them.

For instance, the following sequence utilizing 19 elements was evolved as the perfect 9-input median circuit: (7, 2), (5, 8), (4, 0), (6, 0), (6, 7), (7, 3), (5, 4), (5, 6), (7, 8), (1, 4), (7, 6), (2, 1), (2, 6), (8, 1), (4, 3), (0, 8), (6, 0), (4, 0), (6, 4). Here is an example of a 2-element circuit: (8, 6), (8,4).

As far as we know, no technique has been proposed in literature how to realize close-to-perfect median circuits if sufficient resources are not available. Our experiment has confirmed again that evolutionary techniques are able to produce novel designs in the situations in which conventional approaches fail. The evolution requires a few seconds on a common PC and hence it could be suitable for real-time applications.

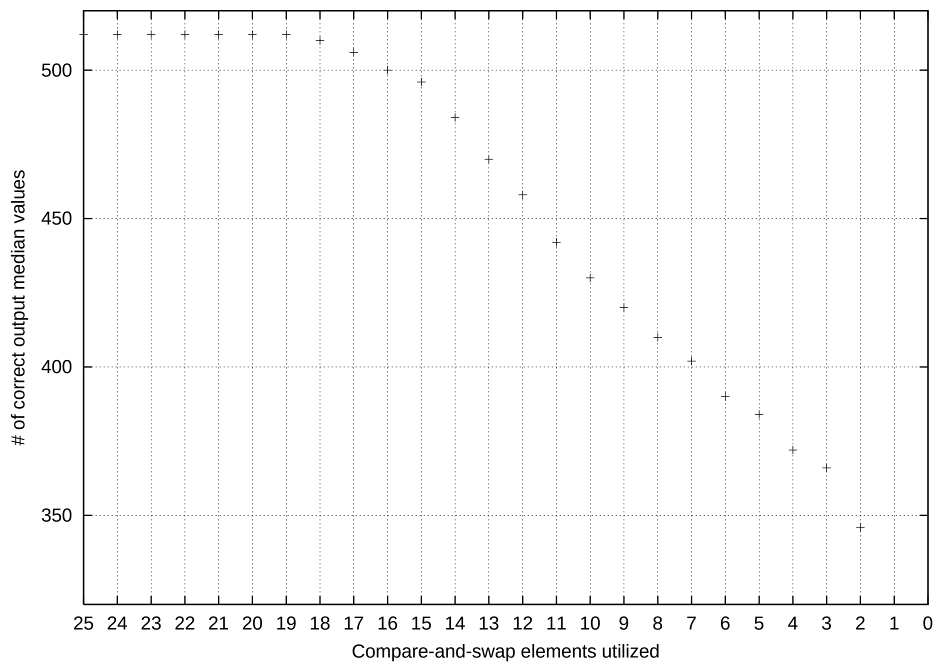


Fig. 4.12. The best fitness value (# of correct output median values) obtained for 2–25 components that were available as building blocks for the evolutionary design.

4.4 Development

The concept of development was adopted by *evolutionary algorithms* community in order to realize non-trivial genotype–phenotype mappings which are necessary to overcome the scalability problem of evolutionary design. In biological development, a multicellular organism is formed from the zygote in a process of cellular division and cellular differentiation. In the context of evolutionary algorithms, computational development might be utilized to achieve diverse objectives, including: adaptation, compacting genotypes, reduction of search space, allowing more complex solutions in solution space, regulation, regeneration, repetition, robustness, scalability, evolvability, parallel construction, emergent behavior and decentralized control [61]. Various approaches have been investigated in the area of evolvable designs; see, for example, [61, 73, 120, 38, 35, 37, 47, 81].

The proposed case study will demonstrate that arbitrarily large sorting networks can be evolved if a suitable developmental scheme is used. Recall that the largest median circuits we evolved (i.e. the design problem easier than the sorting network design problem) have only 11 inputs (CGP encoding) and 25 inputs (C&S encoding).

4.4.1 A Conventional Scalable Approach for Sorting Networks

Some conventional approaches exist for designing *arbitrarily* large sorting networks. Figure 4.13 shows two principles for constructing a sorting network for $N + 1$ inputs when an N-input network is given [55].

- Insertion – the (N+1)st input is inserted into a proper place after the first N elements have been sorted.

- Selection – the largest input value can be selected before we proceed to sort the remaining ones.

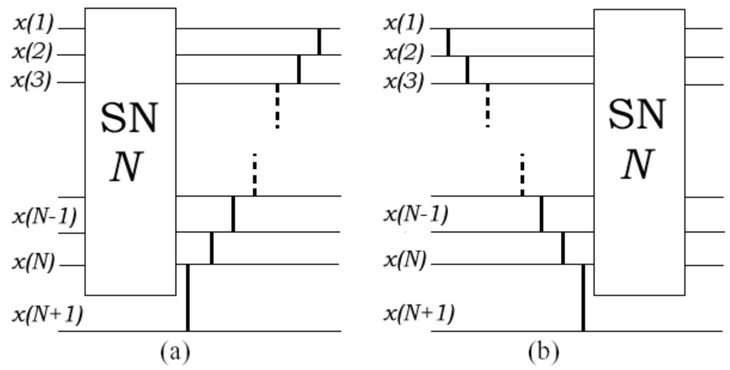


Fig. 4.13. Making (N+1)-sorters from N-sorters: (a) insertion and (b) selection principle

We can see that the insertion principle corresponds to the *straight insertion* algorithm known from the theory of sorting. The selection principle is related to the *bubble sort* algorithm. Examples of sorting networks created using the two principles are shown in Fig. 4.14.

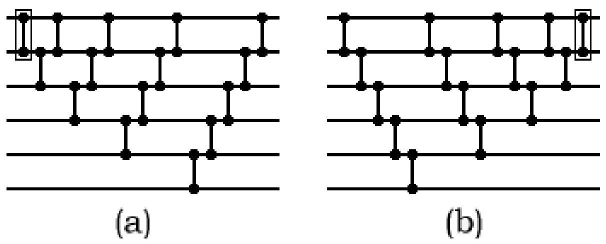


Fig. 4.14. Examples of sorting networks created using (a) insertion and (b) selection principle

It is obvious that the sorting networks created using insertion or selection principle are much larger than those networks designed for a particular N. However, the method can be treated as a general design principle for building *arbitrarily* large sorting networks. In next sections, the principle will be rediscovered firstly and then improved by means of evolutionary techniques.

4.4.2 A Developmental Scheme for Growing Sorting Networks

The objective is to propose an application-specific development for evolutionary algorithms, which, consequently, will be able to produce innovative arbitrarily large sorting networks. A genetic algorithm is used to design a program—constructor (consisting of application-specific instructions)—that is able to create a larger sorting network from a smaller one (the smallest one is called the embryo). Then the constructor is applied on its results in order to create a larger sorting network and so on.

Sorting networks are encoded as sequences of pairs of integers, similarly to encoding proposed in Chapter 4.3.3. A constructor is a sequence of instructions, each of which is encoded

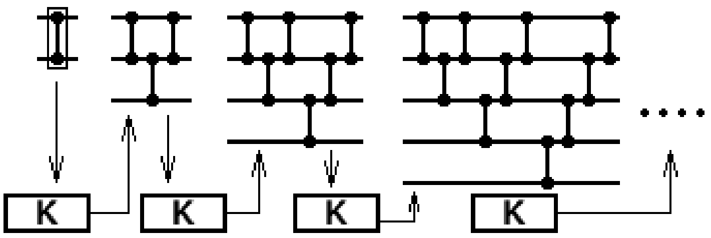


Fig. 4.15. Designing larger sorting networks from smaller sorting networks by means of a constructor K

as three integers—operational code, argument 1 and argument 2. Only two instructions are utilized: *copy* and *copy-and-modify*. Table 4.5 introduces their semantics, variants, operational codes and paramemters. The "modify" instructions read the indices of inputs of a comparator and add the values of their arguments to them. Modulo-operation ensures that the created comparator remains inside the sorting network of a given number of inputs. This type of instructions may be considered as creating a new comparator which is shifted to another position against the original one. The "copy" instructions copy some comparators (beginning from the actual one) to the next instance without shifting. The number of comparators to be copied depends on the instruction argument and the number of inputs of the sorting network being created. The instruction ModifyS (resp. CopyS) differs from ModifyM (resp. CopyM) in handling the *ep* pointer (see below). Note that we (as designers) designed these instructions for this particular task. Hence we call the approach an *application-specific development.*

A sequence representing sorting networks is implemented using a variable-length array. A sequence representing the constructor is implemented as a constant-length array.

Instruction	arg1	arg2	description
0: ModifyS	a	b	$c1 = (c1 + a) \, mod \, w, c2 = (c2 + b) \, mod \, w, cp = cp + 1,$ $np = np + 1$
1: ModifyM	a	b	$c1 = (c1 + a) \, mod \, w, c2 = (c2 + b) \, mod \, w, cp = cp + 1,$ $ep = ep + 1, np = np + 1$
2: CopyS	k	–	copy $w - k$ comparators, $cp = cp + 1, np = np + w - k$
3: CopyM	k	–	copy $w - k$ comparators, $cp = cp + 1, ep = ep + w - k,$ $np = np + w - k$

Table 4.5. Instruction set utilized in development. "*mod*" denotes the modulo operation.

Let $c1$ and $c2$ (i.e. the pair $(c1, c2)$) denote indices of inputs of a comparator in embryo that is processed by an instruction from Table 4.5. Instructions utilize three pieces of information: (1) operation codes and (2) argument values given by GA, and (3) w, which is the number of inputs (width) of the currently constructed sorting network. This value must be inserted into the developmental process externally (from environment). Three pointers are utilized in order to indicate the current position in sequences:

- *ep* – pointer to the source sorting network (embryo pointer),
- *np* – pointer to the first free position in a constructed network (next-position pointer), and
- *cp* – constructor pointer.

As Fig. 4.16 shows, instructions of the constructor are sequentially executed processing the comparator pointed by the embryo pointer (*ep*). The comparators of the embryo are also processed sequentially. Before execution of the first instruction, an auxiliary variable (*e_end*) is initialized by the value of *np*. This auxiliary value marks the end of embryo and is invariable during actual application of the constructor. The process of construction terminates when either all instructions of the constructor are executed or the end of embryo is reached (i.e. $ep = e\_end$). After a single application of the constructor, the obtained sorting network is evaluated. If we apply the constructor again, we obtain a larger sorting network and so on. In such case, the pointers *ep* and *np* possess their values resulted from previous application; only *cp* and *e_end* are updated. Note that the sorting network obtained by repeated application of the constructor possesses all the comparators of its precursors.

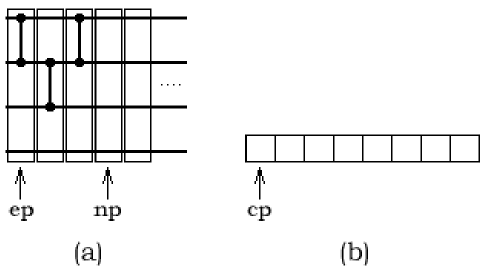


Fig. 4.16. Initialization of the development: (a) growing sorting network and (b) chromosome, i.e. instructions in a constructor

The goal is to find such a constructor that will create valid sorting networks with the minimal number of comparators and/or delay. Because the delay of constructed sorting networks should be minimized, the following special condition has to be satisfied in order to execute a Copy-and-Modify instruction: the result of this instruction is valid only in case that $c1 < c2$ holds for the created comparator. Otherwise, the new comparator is not included in the sorting network and the instruction only updates the embryo pointer.

4.4.3 Genetic Algorithm

A steady state genetic algorithm and a simple genetic algorithm implemented using Galib [135] have been utilized. The GA operates with constant-length chromosomes (programs) represented by triplets of positive integers. Initial population is generated randomly. The probabilities of uniform crossover and mutation and other parameters will be given together with the results in Chapter 4.4.4. The mutation operator is applied on all offspring. We would like to evolve arbitrarily large sorting networks. However, because of problems with the scalability of fitness evaluation, only several instances of the growing sorting network can be evaluated in the fitness calculation process. Assume that we start with a 3-input sorting network. In our case a candidate constructor is used to build the 4-input, 5-input, 6-input and 7-input sorting networks from the 3-input embryo. The fitness value is calculated as follows:

$$\Phi = \Phi(4) + \Phi(5) + \Phi(6) + \Phi(7),$$

where $\Phi(j)$ is the fitness value for a j-input sorting network. This value is calculated using the zero–one principle as the number of input sequences of zeroes and ones sorted correctly. Hence $2^4 + 2^5 + 2^6 + 2^7 = 240$ represents the best possible value that we could obtain. At the end of evolution, we have to test whether the evolved constructor is *general*, i.e. whether it generates infinitely large sorting networks which sort all possible input sequences. If a constructor is able to create a sorting network for a sufficiently high N ($N = 28$ in our case) then we consider the constructor as general.

4.4.4 Results

This section summarizes the experiments that we performed. Figure 4.17 shows the embryos that we utilized.

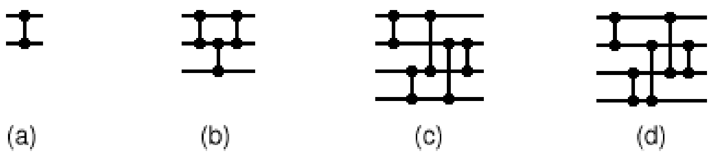


Fig. 4.17. Embryos tested: (a) 2-input, (b) 3-input, (c) 4-input, (d) 4-input – another type

Rediscovering the Conventional Principle. In the first set of experiments, the sorting networks were evolved from a three-input embryo. We used a simple GA, operating with 60 individuals, with the probability of crossover $p_c = 0.75$ and the probability of mutation $p_m = 0.08$. In this experiment, the conventional straight insertion algorithm has been rediscovered (see Fig. 4.18). We were not able to improve the principle of construction in this way. Hence we have tried to change parameters of the development and GA. Especially, we were interested in even-input and odd-input sorting networks.

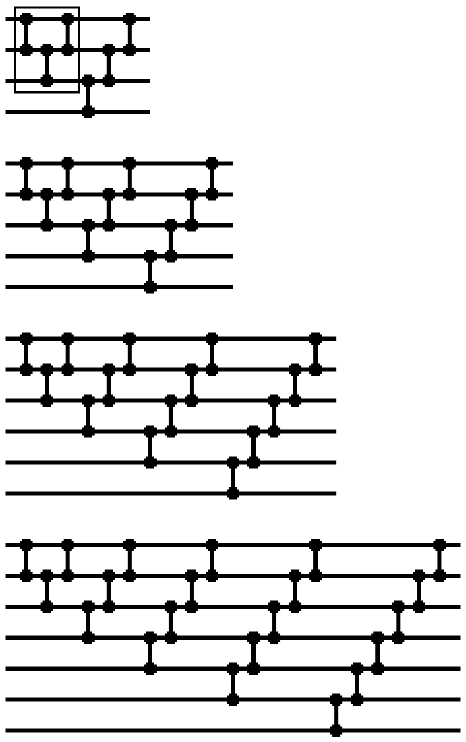


Fig. 4.18. The insertion principle rediscovered using instructions: [ModifyS 2 2] [ModifyS 1 1] [CopyM 3 2] or [ModifyS 3 2] [ModifyS 2 2] [ModifyS 1 1] [CopyM 3 3].

Evolving Odd-input Sorting Networks. Surprisingly, the most interesting odd-input sorting networks were generated by using a 4-input embryo. The best results were produced by a steady-state genetic algorithm with $p_c = 0.74$ and $p_m = 0.1$. The population consists of 400 individuals with overlapping 12 individuals. Table 4.6 shows chromosomes of some evolved constructors[3]. As Table 4.7 indicates, we were able to reduce the number of comparators substantially in this set of experiments. Delays are given in Table 4.8.

Constructor	**Instructions**
g8-4odd_4	*[0 2 2] [0 3 3] [0 2 2] [0 1 1] [0 2 2] [3 0 0] [3 3 2] [3 0 0]*
g6-4odd_2	*[0 2 2] [1 2 3] [0 3 2] [0 1 1] [3 1 3] [3 3 4]*

Table 4.6. Constructors of odd-input sorting networks for a four-input embryo.

N	**5**	**7**	**9**	**11**	**13**	**15**	**17**	**19**	**21**	**23**	**25**	**27**
conventional	10	21	36	55	78	105	136	171	210	253	300	351
g6-4odd_2	*12 (3)*	*22 (4)*	*35 (5)*	*51 (6)*	*70 (7)*	*92 (8)*	*117 (9)*	*145 (10)*	*176 (11)*	*210 (12)*	*247 (13)*	*287 (14)*

Table 4.7. The number of comparators for odd-input sorting networks created using constructor g6-4odd_2. The values in parentheses indicate the number of redundant comparators.

If the number of comparators is measured then the best-evolved sorting network is given in Fig. 4.19. In case of minimizing the delay, the best solution is shown in Fig. 4.20. However, all the sorting networks contain redundant comparators (see parentheses in the tables) which make their delay unnecessarily long. After their removal we can obtain the delay typical of the conventional solution.

N	**5**	**7**	**9**	**11**	**13**	**15**	**17**	**19**	**21**	**23**	**25**	**27**
conventional	7	11	15	19	23	27	31	35	39	43	47	51
g8-4odd_4	*11 (6)*	*19 (11)*	*27 (15)*	*35 (19)*	*43 (23)*	*51 (27)*	*59 (31)*	*67 (35)*	*75 (39)*	*83 (43)*	*91 (47)*	*99 (51)*

Table 4.8. Delay of odd-input sorting networks created using constructor g6-4odd_2. The values in parentheses indicate delay obtained after removing the redundant comparators.

Evolving Even-input Sorting Networks. In the previous section, we discovered better constructors than the conventional approach offers for the odd-input sorting networks. In contrast to the previous section, 2-input, 3-input and 4-input embryos have been confirmed as useful for constructing novel even-input sorting networks. We applied a simple genetic algorithm with $p_c = 0.7$, $p_m = 0,023$ and population size 60. The *g8-4even_2* constructor (represented by instructions [1 4 4] [1 2 1] [0 4 3] [0 2 2] [0 3 3] [3 4 1] [0 2 2] [3 1 3]) is one of the best constructors we have ever evolved. This constructor uses a four-input embryo and produces sorting networks with a better comparator count and delay than the best conventional solution (after removal of redundant comparators). Tables 4.9 and 4.10 summarize results for *g8-4even_2* constructor. Sorting networks created using the constructors are shown in Fig. 4.21.

[3] Operation codes are given instead of symbolic names according to Table 4.5.

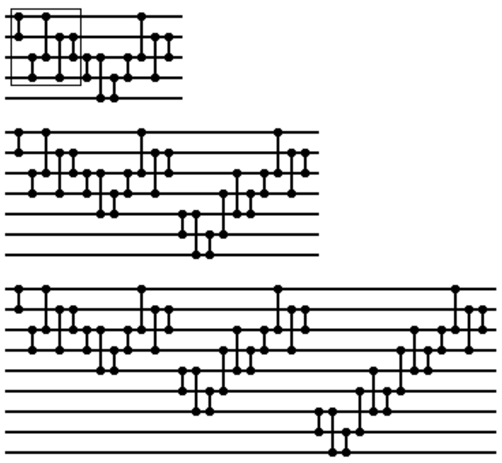


Fig. 4.19. Comparator-efficient odd-input sorting networks created by means of the constructor *g6-4odd_2*. The embryo is marked.

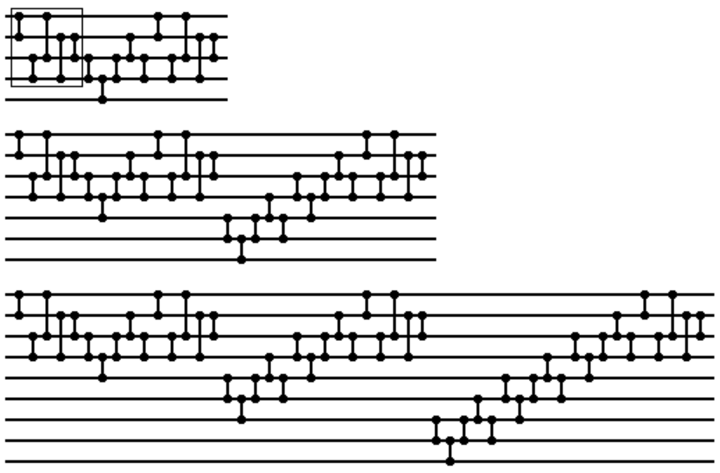


Fig. 4.20. Delay-efficient odd-input sorting networks created by means of the constructor *g8-4odd_4*.

N	**6**	**8**	**10**	**12**	**14**	**16**	**18**	**20**	**22**	**24**	**26**	**28**
conventional	15	28	45	66	91	120	153	190	231	276	325	378
g8-4even_2	*13 (1)*	*24 (2)*	*38 (3)*	*55 (4)*	*75 (5)*	*98 (6)*	*124 (7)*	*153 (8)*	*185 (9)*	*220 (10)*	*258 (11)*	*299 (12)*

Table 4.9. The number of comparators in sorting networks created using the g8-4even_2 constructor

N	**6**	**8**	**10**	**12**	**14**	**16**	**18**	**20**	**22**	**24**	**26**	**28**
conventional	9	13	17	21	25	29	33	37	41	45	49	53
g8-4even_2	*6 (6)*	*9 (9)*	*14 (12)*	*19 (15)*	*23 (18)*	*26 (21)*	*31 (24)*	*36 (27)*	*41 (30)*	*46 (33)*	*51 (36)*	*56 (39)*

Table 4.10. Delay of even-input sorting networks created using the g8-4even_2constructor

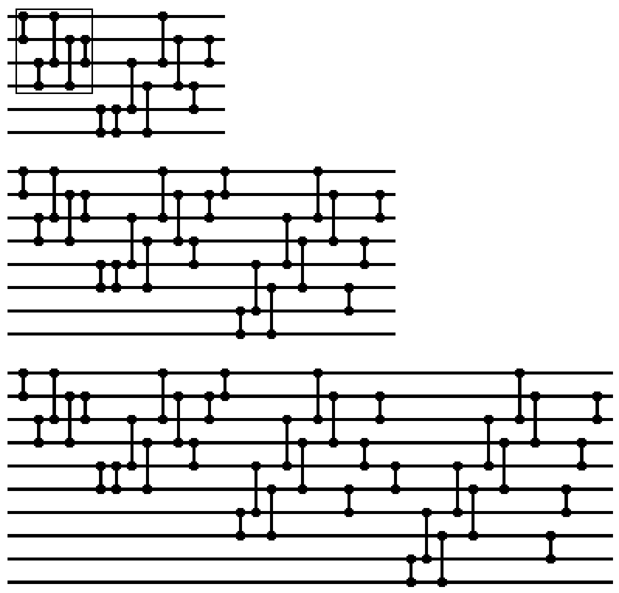


Fig. 4.21. Efficient even-input sorting networks created using the constructor g8-4even_2.

4.4.5 Discussion

The presented evolutionary approach produced sorting networks with better implementation cost (the number of comparators) than the conventional approach for even-input as well as odd-input sorting networks. Delay of even-input sorting networks was also improved. However, in case of odd-input sorting networks, none of the presented constructors is better than a conventional one in terms of delay.

We have discovered that the best-known constructor for even-input sorting networks (*g8-4even_2*) can be utilized to improve delay in case of odd-input networks. Figure 4.22 shows that by removing the bottom line together with "connected" comparators, the odd-input sorting network is established. We verified the improvement of created sorting networks for $N \leq 29$.

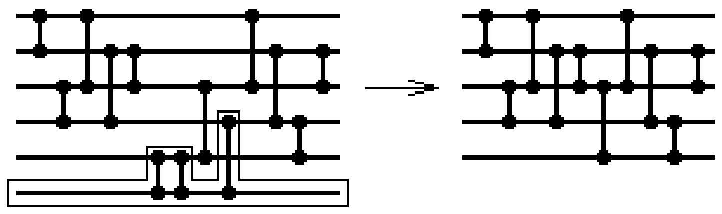


Fig. 4.22. Creating delay efficient odd-input sorting networks from even-input sorting networks by removing the bottom line of comparators. The original six-input sorting network: (0,1) (2,3) (0,2) (1,3) (1,2) (4,5) (4,5) (2,4) (3,5) (0,2) (1,3) (3,4) (1,2). The new five-input sorting network: (0,1) (2,3) (0,2) (1,3) (1,2) (2,4) (0,2) (1,3) (3,4) (1,2).

More than 10,000 independent runs of evolutionary algorithm were performed. The number of generations needed for gaining a solution varies from about 150 to many thousands. We have found out the limit 10,000 generations to be sufficient to get some solutions in reasonable time. If the evolution does not terminate successfully up to this limit, no result is produced.

Considering this restriction, 58% of independent runs of evolutionary process terminated successfully. Figures 4.23 and 4.24 clearly illustrate the obtained improvement.

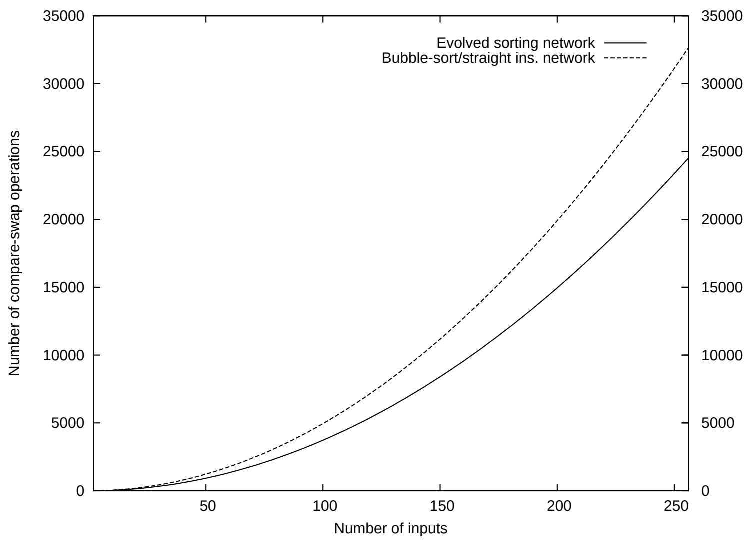


Fig. 4.23. The number of comparators reduced in comparison with Bubble/Straight Insertion Network

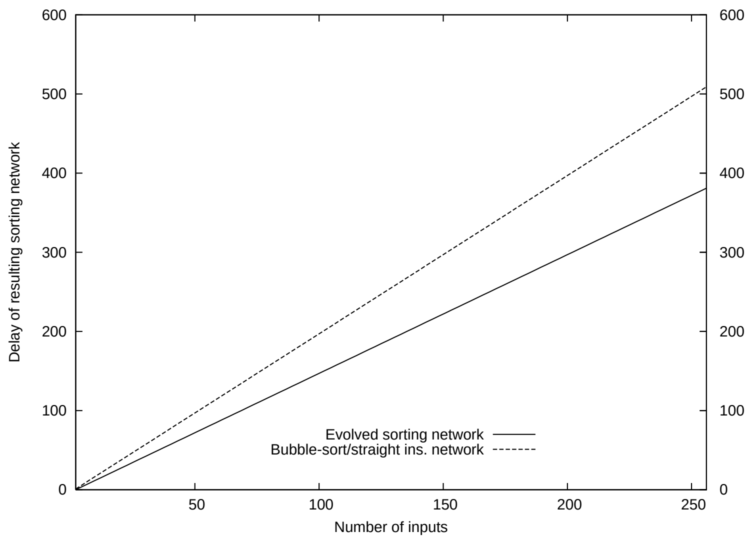


Fig. 4.24. Delay reduced in comparison with Bubble/Straight Insertion Network

The main feature of the proposed developmental system for genetic algorithm is that a lot of problem-domain knowledge (such as the definition and use of copy and modify instructions) has been presented in its inductive bias. We do believe that the idea of evolving constructors for infinitely growing objects is generally applicable. However, it is difficult to define an embryo and appropriate domain knowledge for a particular problem. It seems that the design of an efficient developmental system is as difficult as the design of an efficient genetic algorithm for a given problem.

Except the instructions that we had to design manually for this particular application, the proposed developmental scheme has utilized another information—the size of the currently constructed network, N. This information is not a part of our artificial genetic code. Therefore, we can understand it as a property of environment, which surrounds a growing sorting network. It is obvious that as positional information is crucial for biological development [2], no correct sorting network can be created without a correct N.

The reported research represents the rare case in which a new scalable principle is discovered by an evolutionary algorithm. In most cases, evolutionary algorithms are being used to find a single suitable solution. Other developmental approaches, inspired in the proposed method, are now investigated by M. Bidlo. He also formally proved that the evolved constructors are general [9].

4.5 Functional-Level Evolution

Another approach introduced to overcome the problem of scale is called the *functional-level evolution* [82, 94]. In contrast to the concept of development, which was borrowed from biology, the functional-level evolution represents an engineering approach to problem solving. The idea can easily be illustrated on CGP. If one replaces gates by higher-level functional elements (such as adders, multipliers, comparators etc.) and one-bit connections by, say, 8-bit datapaths then much more complex systems can be evolved. An important property is that the size of chromosome remains unchanged. Stoica et al. noted in [110]: "From the evolvable hardware perspective, it is interesting to have programmable granularity, allowing the sampling of novel architectures together with the possibility of implementing standard ones. The optimal choice of elementary block type and granularity is task dependent." It is the designer's challenge to define the *representational bias* in order to obtain suitable search space within the entire space of possible solutions.

In this section, first, we will mention the extension of CGP to the functional level. This approach will be elaborated in detail in Chapter 5.4. Then the evolutionary design of benchmark circuits will be presented.

4.5.1 Functional-Level CGP

We extended CGP to operate at the functional level in order to evolve novel image operators [90]. These operators have nine 8-bit inputs and a single 8-bit output and they are composed of functions such as addition, minimum, maximum and logic functions operating at 8 bits. A complete hardware implementation of these evolvable filters will be presented in chapter on intrinsic evolution (Chapter 5.4). The system can be used either to adapt image filters in situ or to assist the designer in the design process of a particular filter. In the second case, the resulting filter is available at the level of synthesizable VHDL code. When implemented in an FPGA, the filters consist of 1000–5000 equivalent gates [94]. Considering the fact that the gate level evolution is tractable for circuits approx. up to 100 gates, the functional level allowed us to increase the complexity by two orders of magnitude in this case. In order to evolve this type of circuits, we have to provide some domain knowledge. In our case we had to define suitable functional blocks and interconnection strategy. Note that none useful filters of this type have been evolved at the gate level. Finally, we extended the approach to generate easily-testable image operators. In order to obtain easily testable circuits, we restricted the design space to the class of testable circuits. Surprisingly, we obtained better filters (in terms of visual quality of output images) than we have ever evolved without the requirement on testability [93].

4.5.2 Evolutionary Design of Benchmark Circuits

We mentioned in Chapter 3.2.5 that the evaluation time of a candidate digital circuit grows exponentially with the increasing number of circuit inputs. If we were able to completely evaluate a candidate solution in a linear or quadratic time (with respect to the number of circuit inputs/components) the evolutionary design process should be more effective and scalable. Fortunately, some methods exist to predict testability of a digital circuit in a quadratic time [118]. Hence we utilized EAs to design benchmark circuits.

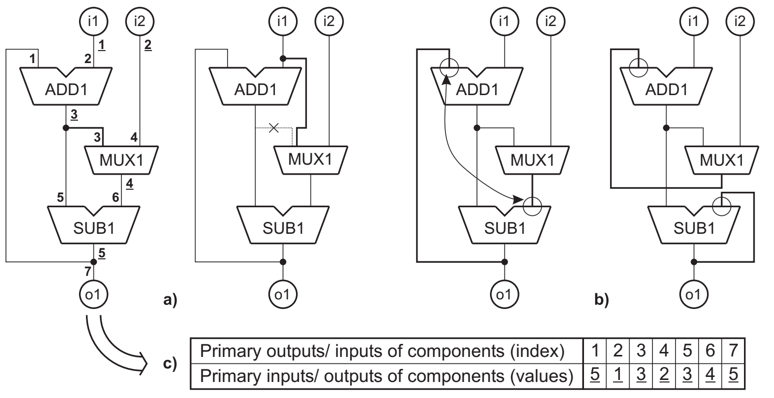
Testability of a design is estimated and numerically evaluated by a testability analysis (TA) method the goal of which is to detect hard-to-test parts of the design. The proper utilization of TA information can result in testability improvement of the design. Therefore, it is reasonable to evaluate and characterize EDA tools using benchmark circuits that are designed to verify not only the function but also the testability. We will show that relatively large and complex (benchmark) circuits consisting of thousands gates can be evolved at the functional level in case that only a given testability property is required and the function is not taken into account. Detailed motivation for this approach from a point of view of diagnostics is presented in [83].

Usually, testability of the design is evaluated by means of *controllability* and *observability* parameters. Existing TA approaches differ in the way in which controllability and observability are defined and measured. In general, controllability (e.g. of an internal circuit node) is understood as an ability to control the node inputs from circuit primary inputs. If it is possible to control the node inputs then such a node is called a controllable node. Similarly, a node is referred to as an observable node if the value present at the node outputs can be observed at a circuit primary output. The goal of controllability (observability) measures is to evaluate the easiness of controlling (observing) signal values. On the basis of these values, testability of the design is evaluated.

Our objective is to produce RT level (Register Transfer level) benchmark circuits automatically. The user is supposed to specify the number of primary inputs and outputs of the circuit, the number and type of components, the requirements on testability (average controllability and observability) and parameters of the evolutionary algorithm. The program generates a benchmark circuit according to the predefined requirements. The resulting circuit consists of components, each of them described behaviorally in VHDL. All the generated circuits are synthesizable. The program inserts the registers automatically in order to meet the requirements on testability and to minimize the number of registers.

Circuit structure representation and EA used. Every circuit is considered as a graph represented by a fixed-size integer array. Evolutionary algorithm operates on these arrays. Registers are not reflected in the representation; they are “inserted” into a circuit before the testability analysis and synthesis procedures are performed.

A circuit consists of components whose inputs and outputs are uniquely numbered. Primary inputs and primary outputs are numbered too. Because any component input can be connected to only a single component output, we can represent the circuit as an array, in which the index is the component input and the value is the identification of the connected output. Primary inputs are treated as outputs of a component and primary outputs are treated as inputs of a component connected to the testbench circuit. Figure 4.25 provides a simple example.



Primary outputs/ inputs of components (index)	1	2	3	4	5	6	7
Primary inputs/ outputs of components (values)	5	1	3	2	3	4	5

Fig. 4.25. Two types of mutation (a), (b). Circuit representation in chromosome (c)

The initial population consisting of P individuals is generated randomly. New populations are formed using tournament selection and mutation operator (which modifies connections of components). The N weakest candidate circuits are replaced by mutated parents, elitism is ensured. The evolution is left to run for a given number of generations. The fittest individual is considered as an acceptable result and transformed to VHDL code. Mutation works as follows (see Figure 4.25):

1. The input of a component is selected on which the mutation operator will be applied.
2. If the output connected to this input is also connected to another component(s) then the selected input is simply reconnected to a randomly selected output of another component or to one of the primary circuit inputs (see Figure 4.25a).
3. If the output connected to this input is not connected to another component(s) then another input of a component or primary output of the circuit under design is randomly selected and selected gates are simply reconnected (see Figure 4.25b).
4. The mutation operator always respects the size of data path.

Fitness calculation. The fitness function, which has to be maximized here, combines three objectives: (1) circuit structure, (2) component interconnections and (3) circuit testability.

At first, the circuit structure is evaluated. The circuit is analyzed with the goal to identify isolated subcircuits and for structures which are possibly to be removed during synthesis, i.e.

$$structure = 0.25 \cdot \left(1 - \frac{useless\_comps}{comps\_count}\right) \tag{4.1}$$

where *useless_comps* denotes the number of components in isolated subcircuits and the number of components to be removed during the process of synthesis and *comps_count* denotes the number of all circuit components. The value of *structure* parameter is a real number from $\langle 0; 0.25\rangle$ interval.

If isolated subcircuits and structures which could be removed are found, the fitness evaluation process is stopped and the value of *connects* and *testability* parameters is set to 0. If no separated subcircuits and structures which could be removed are found in the circuit, the circuit interconnection and testability analysis follows.

The goal of interconnection analysis is to evaluate the variability of interconnections of the circuit (see Equation 4.2). The circuit is analyzed for components whose inputs are connected to the same point (*short_cuts*) and for direct connections from primary inputs to primary outputs (*direct_connects*). *Comp_inputs* denotes the sum of all components inputs and *pri_outputs* denotes the number of circuit primary outputs. The value of *connects* parameter is evaluated by a real number from $\langle 0;1\rangle$ interval, i.e.

$$connects = 1 - \frac{short\_cuts + direct\_connects}{comp\_inputs + pri\_outputs} \tag{4.2}$$

The testability analysis follows after interconnection analysis. The testability parameters are calculated using Strnadel's algorithm [118]. The algorithm is able to evaluate average controllability *(avg_cont)* and observability *(avg_obs)* of a candidate circuit in the quadratic time complexity. The values are compared with the values of controllability *(req_cont)* and observability *(req_obs)* required by the user. The value of *testability* parameter is a real number from $\langle 0;1\rangle$ interval, i.e.

$$testability = 1 - 0.5 \cdot (req\_cont - avg\_cont)^2 - 0.5\,(req\_obs - avg\_obs)^2 \tag{4.3}$$

The fitness function given by Equation 4.4 combines the results for circuit structure, components' interconnections and testability analysis. The resulting fitness value is a real number from $\langle 0;1\rangle$ interval, i.e.

$$\Phi = structure + 0.25 \cdot connects + 0.5 \cdot testability \tag{4.4}$$

Examples of evolved circuits. Table 4.11 gives examples of parameters of benchmark circuits evolved according to different specifications (mean values are calculated on the basis of 5 independent runs). We required 33% observability and 33% controllability in average. In each experiment we produced 200 generations using 25-member population and the mutation probability 5%. The experiments were performed at PC with Intel Xeon 2.8GHz, 1GB RAM. Although the resulting testability parameters are slightly different from the required testability parameters, we can see that the evolved circuits tend to have desired properties and we can classify these circuits as very good for our purposes. Notice that sometimes it can be impossible to create a circuit which exhibits the desired testability properties using the predefined components and input and output ports. Therefore, we are looking for a good approximation in this task.

Inp/Out	Components	Controllability	Observability	Gates	FF's	Time [hour:min]
5/5	50	33.32%	32.89%	7003	640	00:09
10/10	100	31.61%	29.82%	14178	1312	00:16
40/40	500	34.75%	29.52%	70655	6510	02:50
80/80	1000	34.87%	26.79%	144182	13193	12:20

Table 4.11. Parameters of benchmark circuits evolved using the specifications given in the first two columns (33% observability and 33% controllability required)

Figure 4.26 shows an example of benchmark circuit developed with our methodology. The circuit consists of 20 components and requires 83 Virtex slices after synthesis to Xilinx Virtex II FPGA. We required 80% observability and 80% controllability on average; the obtained results are 80.6% for observability and 74.5% for controllability. Notice that registers were included to the circuit after the design of the structure of the circuit, i.e. the figure contains more elements. Two types of circuits are analyzed in detail in next paragraphs.

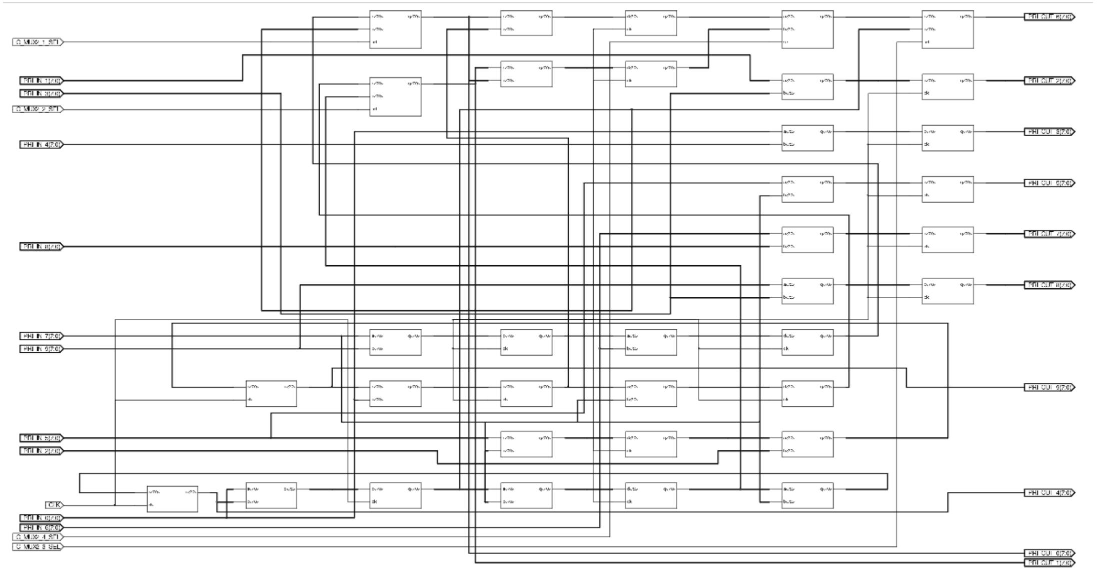


Fig. 4.26. An example of evolved benchmark circuit

20-component circuit. The structural requirements on the produced circuits were: 80% controllability, 80% observability, 5 inputs, 5 outputs, 20 components: 8 × ADD(8bit), 8 × SUB(8bit) and 4 × MUX2(8bit)[4]. Figure 4.27a shows the differences between required and obtained controllability and observability values for 20 evolved 20-component circuits. The obtained controllability ranges between 79.35% and 79.82% on average. The obtained observability ranges between 76.22% and 77.46% on average.

500-component circuit. The structural requirements on the produced circuits were: 80% controllability, 80% observability, 40 inputs, 40 outputs, 500 components: 80 × ADD(8bit), 80 × ADD(16bit), 80 × SUB(8bit), 80 × SUB(16bit), 80 × MUX2(8bit), 80 × MUX2(16bit) and 20 × MUL(8,16bit). Figure 4.27b shows the differences between required and obtained controllability and observability for 10 evolved 500-component circuits. The obtained controllability ranges between 77.02% and 78.82% on average. The obtained observability ranges between 72.99% and 75.18% on average.

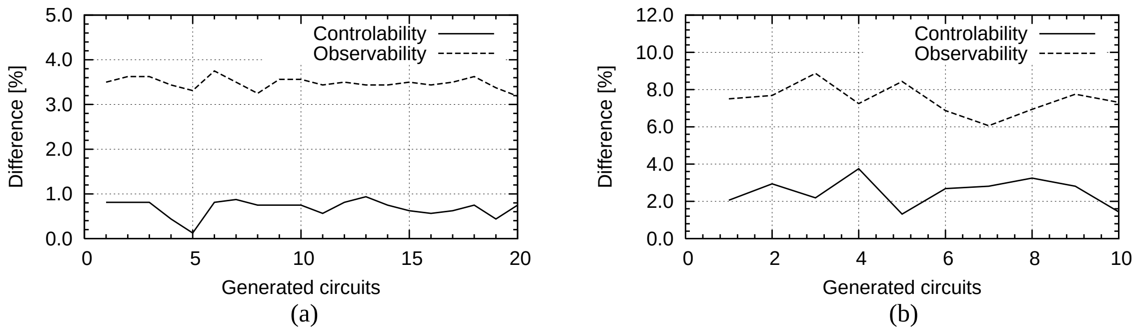


Fig. 4.27. The differences of required and obtained controllability and observability for 20 evolved (a) 20-component circuits and (b) 500-component circuits.

[4] ADD – adder, MUL – multiplier, SUB – subtracter, MUX – multiplexer

Controllability/observability design space exploration. The objective of this task was to utilize the proposed evolutionary method to explore the design space in case that controllability (observability) is changed and observability (controllability) remains constant.

This experiment was performed for a 50-components circuit (8 inputs, 8 outputs, 50 components: 8×ADD(8bit), 8×SUB(8bit), 8×MUX2(8bit), 8×ADD(16bit), 8×SUB(16bit), 8×MUX2(16bit) and 2×MUL(8,16bit). The parameters of EA are as follows: population size 30, generations 200, mutation probability 2% and replacement probability 95%. Results were obtained as average value from 20 independent runs. We required the following testability properties: (a) 50% controllability (fixed), observability is incremented with step of 10%; (b) 50% observability (fixed), controllability is incremented with step of 10%.

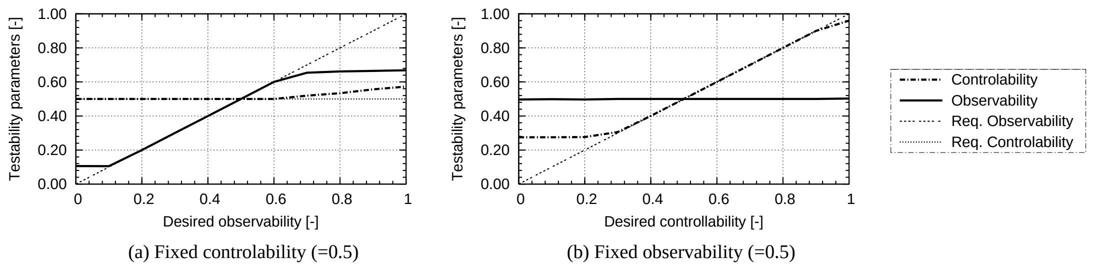


Fig. 4.28. Testability parameters obtained when (a) observability increases from 0.0 to 1.0 and controllability is fixed (=0.5). and (b) controllability increases from 0.0 to 1.0 and observability is fixed (=0.5).

It can be seen in Figure 4.28 that requirements on controllability and observability parameters of circuit could be satisfied only on a specific range of possible values of controllability and observability parameters. This restriction is caused by structural properties of the circuit (the number of circuit's primary inputs/outputs, the number and diagnostic properties of used components). For example, the structure of circuit does not allow creating circuits with the 0% controllability (observability) parameters because the 0% controllability (observability) circuit corresponds to a circuit with none controllable input (observable output).

Summary. There are two interesting features of this approach: (1) We are able to evolve relatively complex circuits with the required testability properties. According to our knowledge, no approach exists to accomplish this task. (2) The evolved circuits are the largest circuits evolved so far.

4.6 Summary

In this chapter, we presented digital circuits evolved at the gate level, functional level, using development and some application-specific encoding. These circuits, together with circuits that will be presented in next chapters, represent the data that will be analyzed and evaluated in Chapter 8. Almost in all cases, the presented results can be qualified as human-competitive.

5. Intrinsic Digital Evolvable Hardware Using FPGAs

In case of intrinsic evolution of electronic circuits, every candidate circuit is evaluated in a physical device, i.e. no simulator is utilized. This approach has several advantages:

1. Evaluation of candidate circuits is usually faster. For example, Stoica reported that hardware implementation (JPL's FPTA) can reduce the time of the evolutionary design of analog circuits by 4+ orders of magnitude if compared with PSPICE running at Pentium II 3000 Pro [110].
2. As the complete evolvable system can be implemented on a single board (or on a single chip), the resulting system can be very small, low-power and application-specific in comparison with a general-purpose PC (see, for example, [43, 110, 119]).
3. The evolutionary algorithm can exploit properties of the given reconfigurable device operating in the given environment (as initially demonstrated by Thompson [121]). Therefore, novel and unexpected circuits and behaviors can be discovered.

We have surveyed reconfigurable devices suitable for evolvable hardware in Chapter 3.2.1. FPGAs represent the most popular platform for digital evolvable hardware. The FPGA-based implementations of evolvable hardware can be divided into two groups:

(1) The FPGA serves in the fitness calculation only. The evolutionary algorithm (which is executed on a personal computer or in a digital signal processor (DSP)) sends configuration bitstreams representing candidate circuits to the FPGA in order to obtain their fitness values.

(2) The entire evolvable system is implemented in an FPGA. As an example, we can mention Tufte's and Haddow's research in which they demonstrated on a simple example the Complete Hardware Evolution [129]. These implementations require a hardware realization of the evolutionary algorithm—this area is relatively independent of evolvable hardware. Various solutions have been proposed, for example, see [104].

Most FPGA families can be configured only *externally* (i.e. from an external device connected to the configuration port). The *internal reconfiguration* means that a circuit placed *inside* the FPGA can configure the programmable elements of the same FPGA (which is important for evolvable hardware). Although the internal configuration access port (ICAP) has been integrated into the newest Xilinx Virtex II family [10], it is still too slow for our purposes. We will show in Chapter 5.3 that approximately 640 ns are needed to evaluate a candidate circuit. As soon as the reconfiguration time should be much shorter than the evaluation time (to make the approach reasonable), we need to reconfigure the circuit in tens of nanoseconds—but it is not possible using the existing configuration subsystems of FPGAs.

In order to overcome the problem of internal reconfiguration, we have developed virtual reconfigurable circuits (VRC) [94]. The use of VRCs has allowed us to introduce a novel approach to the design of complete evolvable systems in a single FPGA. That approach will be presented in Chapter 5.1. The objective is to speed up the evolutionary design process as much as possible. The three evolvable systems completely implemented in a single FPGA will illustrate the concept.

5.1 Virtual Reconfigurable Circuits

Virtual reconfigurable circuits were introduced for digital evolvable hardware as a new kind of rapidly reconfigurable platforms utilizing conventional FPGAs [91, 94]. When the VRC is uploaded into the FPGA then its configuration bitstream has to cause that there will be created the following units in the FPGA: an array of *programmable elements* (PE), programmable interconnection network, configuration memory and configuration port. Fig. 5.1 shows that the VRC is in fact a second reconfiguration layer (consisting of 8 PEs in the example) developed on the top of an FPGA in order to obtain fast reconfiguration and application-specific PEs.

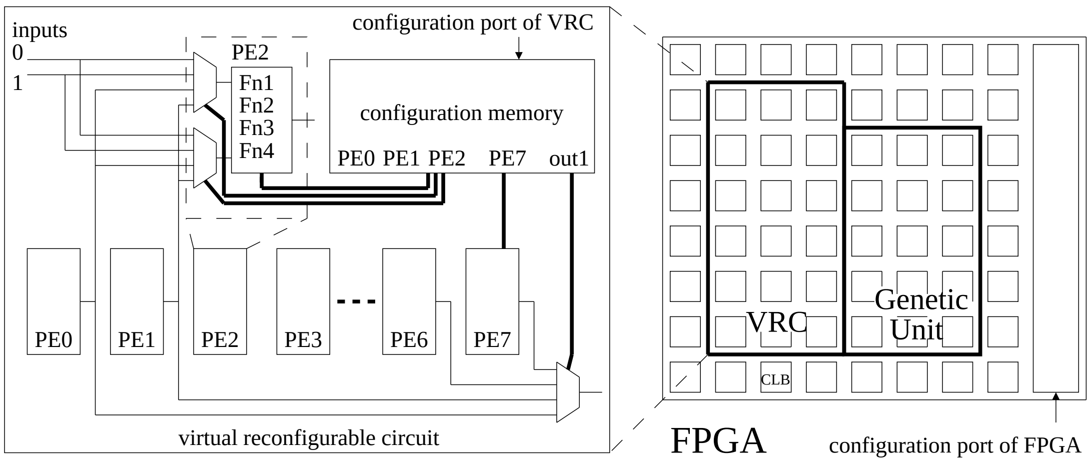


Fig. 5.1. Example of the internal organization of the virtual reconfigurable circuit. The programmable element PE2 is shown in detail.

Designer has the opportunity to design the VRC exactly according to requirements of a given application. In most cases the VRC takes a form of a regular two-dimensional array of programmable elements. The VRC can also be understood as a hardware implementation of the computational model used in CGP.

The implementation of the VRC is based on multiplexers. Fig. 5.1 shows an example. The "virtual" PE2 is controlled using 6 bits determining the selection of its operands (2+2 bits) and internal function (2 bits). The routing circuits are created using 4-input multiplexers. The configuration memory of the VRC is typically implemented as a register array. All bits of the configuration memory are connected to multiplexers that control the routing and selection of functions in PEs.

Because the array of PEs, routing circuits, configuration memory, style of reconfiguration and granularity of the new VRC can be designed exactly according to the requirements of a given application, designers can create an optimized application-specific reconfigurable device. Furthermore, the VRC is described in HDL, i.e. independently of a target platform. It is crucial from our perspective that the VRC can directly be connected to a hardware implementation of the evolutionary algorithm placed on the same FPGA. If the structure of the chromosome corresponds to the configuration interface of the VRC then a very fast reconfiguration can be achieved (e.g. consuming a few clock cycles only)—which is impossible by means of any other technique.

5.2 Uniform Approach to the Design of Evolvable Systems in FPGAs

Figure 5.2 shows a general template for implementation of an evolvable system in a single FPGA by means of VRCs. The evolvable system consists of the VRC, evolutionary algorithm, circuit evaluation unit (fitness calculation) and interface to the PCI bus. As VRC and Fitness Calculation Unit are application-specific, they will be described for each application separately. The Genetic Unit and PCI interface are reused by all applications.

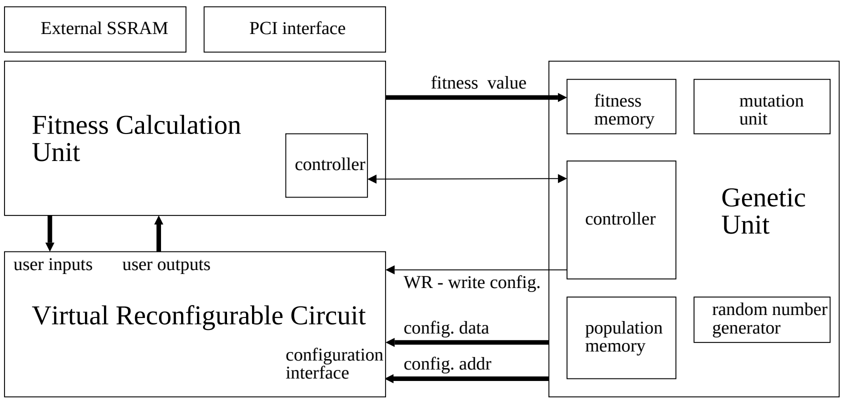


Fig. 5.2. Uniform approach to implementation of evolvable systems in an FPGA

5.2.1 Genetic Unit

All genetic operations are realized as special circuits in the FPGA. Figure 5.3 shows details of the genetic algorithm datapath. The genetic algorithm we utilized is based only on a single mutation operator (bit inversion). The population is stored in a memory whose size is configurable. Another memory is used to store fitness values. Every new population is always generated from the best members of the previous one. Genetic algorithm operates in following steps:

- Initialization Unit generates the first population at random. We use a linear feedback shift register seeded from software.
- Mutation Unit changes a given number of genes (bits) of a population member (this number is configurable) and the modified member is loaded into the VRC. The reconfiguration of VRC is pipelined in order to overlap the reconfiguration by useful computation. Simultaneously, the chromosome is copied into a FIFO memory.
- Genetic Unit is waiting for the evaluation performed by Fitness Unit. If the fitness value is better than the parent's fitness then the chromosome replaces its parent (the chromosome is copied from FIFO to the population memory). Otherwise, the chromosome is removed from FIFO.
- This is repeated until an appropriate number of generations are produced.

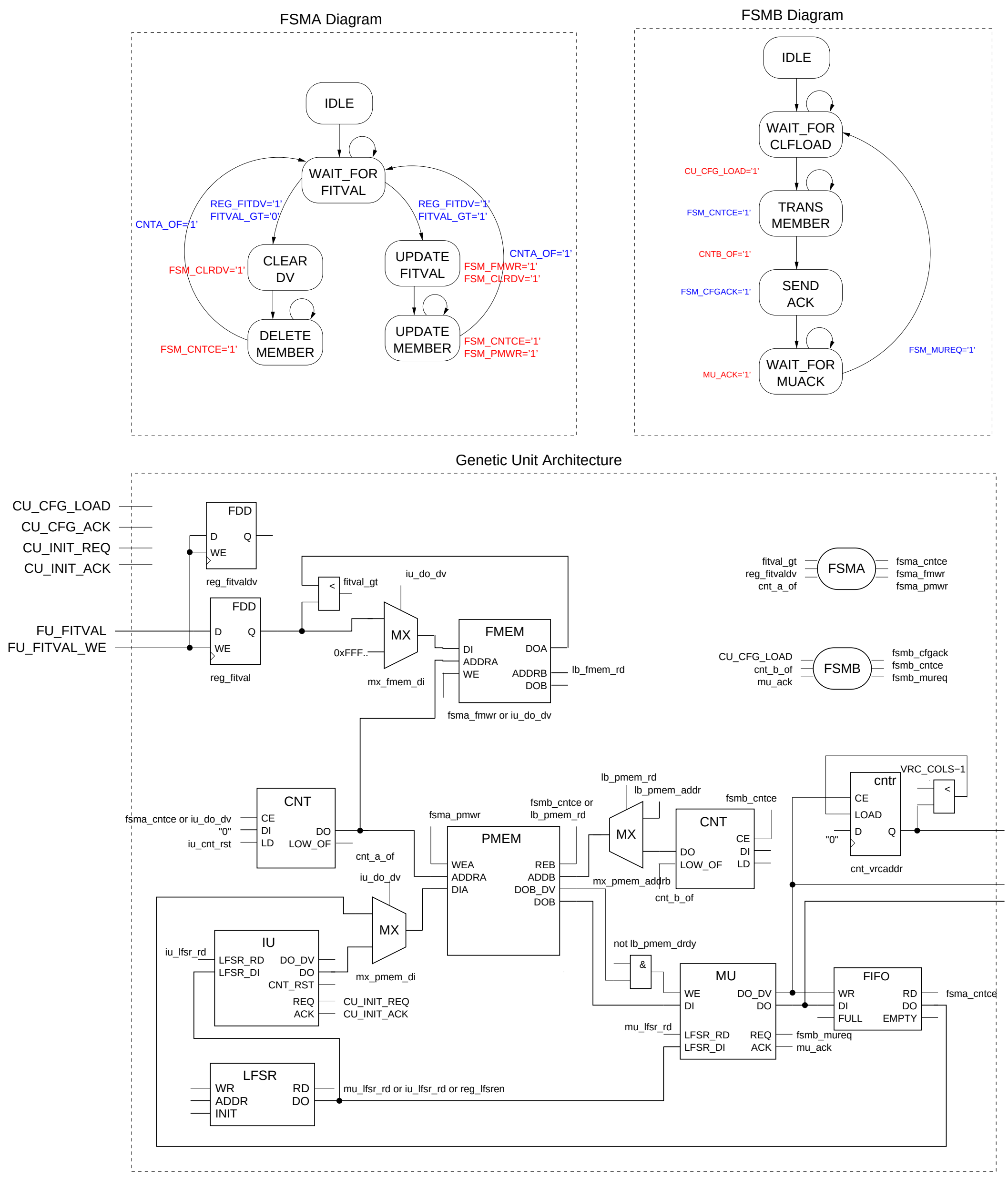


Fig. 5.3. Block diagram of the genetic unit. Symbols used: fitness memory (FMEM), population memory (PMEM), linear feedback shift register (LFSR), finite state machine (FSM), mutation unit (MU), multiplexer (MX), counter (CNT), initialization unit (IU)

This loop is in fact divided into two parts executed concurrently. Because of the partial reconfiguration of VRC, we can send the next population member to VRC although the previous one has not been evaluated yet.

5.2.2 PCI Interface

A special interface has been established to read configurations of VRC from FPGA. That makes the analysis of evolved circuits easier. Furthermore, the evolved configuration can be converted to a program (e.g., C function) doing the same task in software.

5.2.3 Target Platform

Any platform containing a sufficiently large FPGA and external memories can be utilized. We used a COMBO6 card, developed in the Liberouter project [67], which is a PCI card primarily dedicated for a dual-stack (IPv4 and IPv6) router hardware accelerator. This board contains FPGA Virtex XC2V3000 by Xilinx, Inc. with more than 3 mil. equivalent gates, up to 2GB DDR SDRAM, up to 9Mbit context addressable memory, and the three 2MB SSRAM memories. We decided to use this card for our experiments because it offers us a sufficient performance and capacity of FPGA. Furthermore, various hardware modules (such as a circuit allowing the communication with a personal computer via PCI bus) and some design software are available for free. All the presented systems were described in VHDL, simulated using ModelSim and synthesized using LeonardoSpectrum and Xilinx ISE tools to Virtex FPGA XC2V3000bf957 chip which is available on the COMBO6 card.

5.3 Evolvable Combinational Unit

The proposed evolvable combinational unit is an application-specific system for the evolution of combinational circuits, of six user inputs and six user outputs, in a few seconds.

5.3.1 Virtual Reconfigurable Circuit

The VRC depicted in Fig. 5.4 consists of 80 PEs equipped with flip-flops allowing pipelined processing. Every PE can be programmed to perform one of the eight logic functions shown in the same figure. The functions 0 ($c = a$) and 1 ($c = b$) realize a delay. Any PE can be connected to some of circuit inputs or to some of the outputs of PEs placed in the previous column. For instance, in case that the VRC is utilized for the evolution of 3×3-bit multipliers, the inputs 0-2 serve for the first operand and the inputs 3-5 serve for the second operand of the multiplier. In all cases, the 6-bit output is directly connected to the middle PEs of the last column.

The 880-bit configuration bitstream is divided into eight banks of 110 bits. A single bank contains the configuration bits for a single column of the VRC. The configuration bits are stored in the 880-bit configuration register array realized using flip-flops available in the FPGA. We need 8 clock cycles to completely change the configuration information and thus the behavior of the VRC.

5.3.2 Fitness Function

A special pipeline unit was designed to evaluate the circuits uploaded into the VRC. In our case, the unit generates $2^6 = 64$ test vectors (all possible input combinations are generated using a six bit counter), applies them at the VRC input, reads the output vectors from the VRC and compares them against the required vectors (that are stored in a table inside the FPGA). The fitness value is incremented for every output bit calculated correctly. Therefore, the maximal fitness value is 384. In the current implementation, the required behavior (the truth table) is defined in software and can be changed dynamically to simulate a dynamic environment.

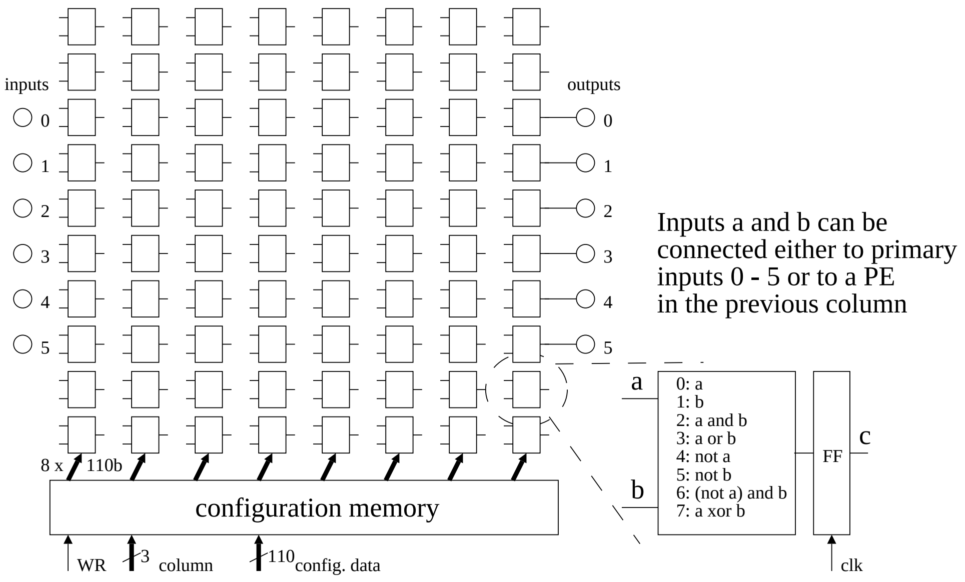


Fig. 5.4. The virtual reconfigurable device for evolution of 6-input/6-output combinational circuits

5.3.3 Synthesis

The entire evolvable system requires 470,385 equivalent gates. Table 5.1 summarizes the results of synthesis. For instance, while the genetic unit requires 1937 function generators and 1001 flip-flops (or latches), the VRC utilizes 2141 function generators and 940 flip-flops (or latches).

Table 5.1. Results of synthesis

Resource	Used	Available	Utilization
IOs	40	684	5.85%
CLB Slices	2761	14336	19.26%
Function Generators	5522	28672	19.26%
Dffs or Latches	2594	30724	8.44%
Block RAMs	5	96	5.21%
Block Multipliers	0	96	0.00%

The population is stored in Block RAMs for this implementation. The design can operate at 77.4 MHz. The results that will be described in the next section were obtained using 50 MHz only because of easier synchronization with the PCI interface. However, there is a potential to go beyond 120MHz by optimizing some parts of the design.

5.3.4 Results

Evolution of Conventional Modules. We successfully evolved conventional combinational circuits, such as 3-bit multipliers, 3-bit adders, multiplexers and parity encoders. Table 5.2 summarizes all results for conventional circuits measured on the evolvable unit running at 100 MHz. These results were obtained after 100 independent runs for each problem.

In case of multipliers, we obtained the fully correct solutions in all cases and in generation 4,975,829 on average. In total, 125 million generations were allowed. Considering the average number of generations, the time of evolution is

Table 5.2. The results obtained for conventional circuits: the number of required generations, the obtained fitness values and the average design time (assuming $f_m = 100\,\mathrm{MHz}$)

Circuit		Required generation	Fitness value	Design time [sec]
3-bit multiplier	average	4,975,829	384	14.330
	std. dev.	3,112,608	0	
	min	1,375,846	384	
	max	16,489,055	384	
3-bit adder	average	86,144	384	0.248
	std. dev.	55,037	0	
	min	13,186	384	
	max	334,053	384	
multiplexor	average	17,880	384	0.051
	std. dev.	7,600	0	
	min	4,590	384	
	max	43,601	384	
parity enc.	average	3,176	384	0.009
	std. dev.	1,775	0	
	min	555	384	
	max	9,689	384	

$$t = \frac{g.p(v+c)}{f_m} = \frac{4975829.4(64+8)}{100.10^6} = 14.3 \text{ s}, \tag{5.1}$$

where g is the number of generations, p is the population size, v is the number of test vectors and c denotes the configuration overhead.

Randomly Generated Functions. In order to explore the limits of the evolvable unit, we tried to evolve combinational circuits of six inputs and k outputs (where $k = 1 \dots 6$) whose behaviors were specified by randomly generated truth tables. For each k, we randomly generated 100 behaviors (truth tables) and performed the evolutionary design of corresponding circuits, allowing 125 million generations (i.e. 6 minutes) for each run. Table 5.3 shows that a correct solution (i.e. the circuit with the fitness value 384) was found practically in all cases when $k = 1$ or 2. No correct solution appeared for larger ks. We measured the average, minimal and maximal fitness values and the standard deviation for each set of circuits for a given k. As soon as the problem becomes harder (k increases), the average fitness value decreases. However, in the worst case ($k = 6$), the average fitness value is still 339, which corresponds to 88% of the required value.

Adaptation Time. Although the unit is able to generate only simple combinational circuits, it could still be suitable for various applications in which the adaptation time can take several seconds and a possible non-perfect solution is acceptable (for example, robot controllers). Figure 5.5 shows a typical adaptation of the 6 input/2 output circuit in case that only the minor changes are injected into the specification (truth table). The five bits were inverted in the truth table after reaching the maximal value (384) for the previous specification. In this example, a perfect circuit was obtained after 46 seconds (on average for the unit running at 100 MHz) in all 20 cases. We can observe that very good suboptimal values are reached in less than one second.

5.3.5 Discussion

As the number of outputs increases, it becomes harder to evolve a perfectly operating circuit in case that its behavior is described by means of a randomly generated truth table. We were not able to evolve any perfect circuit with three or more outputs although a lot of time

Table 5.3. The results obtained for 100 randomly generated circuits in each set of 1-6 outputs: the number of generations produced, the obtained fitness value and the average design time (assuming $f_m = 100\,\text{MHz}$, max. 125 million generations).

Circuit		Generation	Fitness	Design time [sec]
6inp1out	average	876,334	384.00	2.524
	std. dev.	636,457	0	
	min	101,353	384.00	
	max	3,769,178	(100x) 384.00	
6inp2out	average	33,206,207	383.97	95.634
	std. dev.	24,640,136	0.17	
	min	1,688,659	383.00	
	max	116,978,551	(97x) 384.00	
6inp3out	average	75,257,854	379.44	216.743
	std. dev.	30,099,855	1.71	
	min	10,467,567	375.00	
	max	124,955,462	(1x) 383.00	
6inp4out	average	79,666,503	368.57	229.440
	std. dev.	29,168,135	2.51	
	min	14,488,953	361.00	
	max	124,523,767	(2x) 374.00	
6inp5out	average	74,457,446	354.18	214.437
	std. dev.	28,600,970	3.07	
	min	13,424,232	348.00	
	max	122,953,216	(3x) 360.00	
6inp6out	average	68,708,267	338.92	197.880
	std. dev.	30,050,336	4.14	
	min	8,579,893	330.00	
	max	124,429,008	(1x) 351.00	

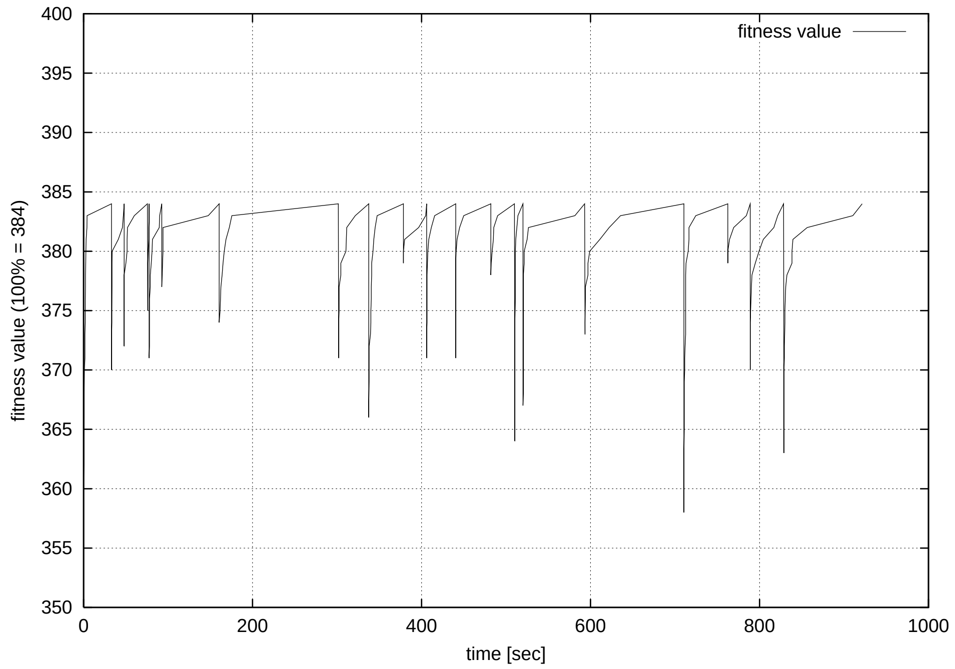


Fig. 5.5. A typical adaptation of a 6 input/2 output circuit in case that five bits are inverted in the specification (truth table) after reaching the maximal value (384) for the previous specification. These results are given for 100 MHz.

was available for the evolution. It seems that the proposed unit is not able to do it at all. We think that the reason is twofold. First, the unit does not have sufficient resources (PEs, interconnection options, etc.). Second, the evolutionary algorithm is not efficient. However, these reasons represent the trade-off. A small improvement in the performance will probably lead to more expensive hardware implementations. On the other hand, the unit is able to produce close to perfect circuits in a few seconds (with the fitness value better than 90%) for almost all specifications expressed by means of truth tables.

A software tool for the evolutionary design of combinational circuits has been developed allowing 1 million generations to be processed in 18 seconds on Pentium IV at 2.6 GHz. The proposed unit operating at 100 MHz is able to process 1 million generations in 3 seconds, i.e. the evolution is six times faster. However, the software simulator operates with 32-bit data units, so it can evaluate 32 test vectors in parallel. Considering this, the FPGA implementation is in fact 192 times faster. Note that the hardware implementation of the 32-bit reconfigurable PEs is very area-consuming in the VRC. In contrast to the evolutionary design of analog circuits using essentially slow analog circuit simulators, the software simulator of digital CGP is very fast, so the obtained speedup is not impressive.

5.4 Evolvable Image Filter

The evolutionary algorithm will be used to find a configuration of an array of programmable elements in order to perform the image filtering task. This problem was simulated in software [90] and the experience learned used to create its hardware implementation. Candidate image filters will be evaluated using a training image. The filters have to minimize the difference between the corrupted and uncorrupted version of the training image. The corrupted version will contain a particular type of noise. We will show that the evolved filters work effectively for a reasonable class of test images corrupted by the same type of noise. A special filter will be evolved for a given type of noise.

Figure 5.6 shows the architecture of the evolvable image filter. External SSRAM memories are utilized to store the corrupted image, original image and filtered image. In contrast to previous application, the evolutionary design of image filters is based on the *functional-level* CGP.

5.4.1 Virtual Reconfigurable Circuit

Every image operator will be considered as a digital circuit of nine 8bit inputs and a single 8bit output, which processes gray-scaled (8bits/pixel) images. As Fig. 5.7 shows, every pixel value of the filtered image is calculated using a corresponding pixel and its eight neighbors in the processed image.

The reconfigurable circuit consists of two-input Configurable Functional Blocks (CFBs). Any input of each CFB may be connected to the primary circuit inputs or to the output of a CFB, which is placed anywhere in the preceding column. The interconnection is implemented using multiplexers. Any CFB can be programmed to realize one of functions given in Table 5.4. These functions were recognized as useful for this task in [93]. The reconfiguration is performed column by column. The computation is pipelined; a column of CFBs represents a stage of the pipeline. Registers are inserted between columns in order to synchronize the input pixels with CFB outputs.

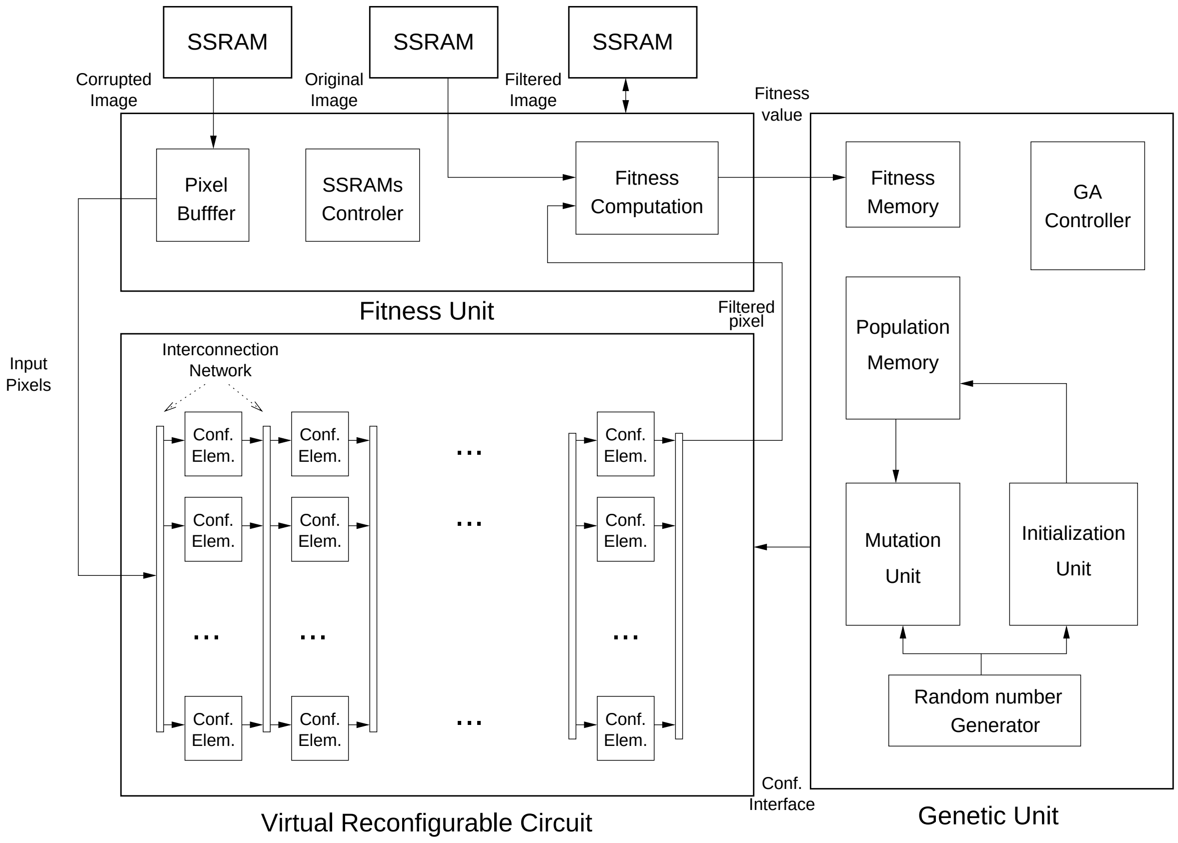


Fig. 5.6. Architecture of the evolvable image filter in FPGA

Table 5.4. Functions in CFBs operating over two pixels

	function	description
0	$x \vee y$	binary or
1	$x \wedge y$	binary and
2	$x \oplus y$	binary xor
3	$x + y$	addition
4	$x +^s y$	addition with saturation
5	$(x + y) >> 1$	average
6	$Max(x, y)$	maximum
7	$Min(x, y)$	minimum

It is evident that the VRC implements a combinational behavior. As many image filters exhibit this property and pipelining of combinational circuits is very effective, we decided to implement only this type of filters. Of course, a feedback, multiplication operators or time delays could be introduced into the filter in order to obtain more complex behaviors. However, it requires much more resources in FPGA and an additional time to evaluate a candidate circuit. The chosen solution represents a reasonable compromise.

5.4.2 Fitness Function

The fitness calculation is realized in Fitness Unit. The pixels of corrupted image u are loaded from external SSRAM memory and forwarded to inputs of VRC. Pixels of filtered image v are sent back to the Fitness Unit, where they are compared with the pixels of original image

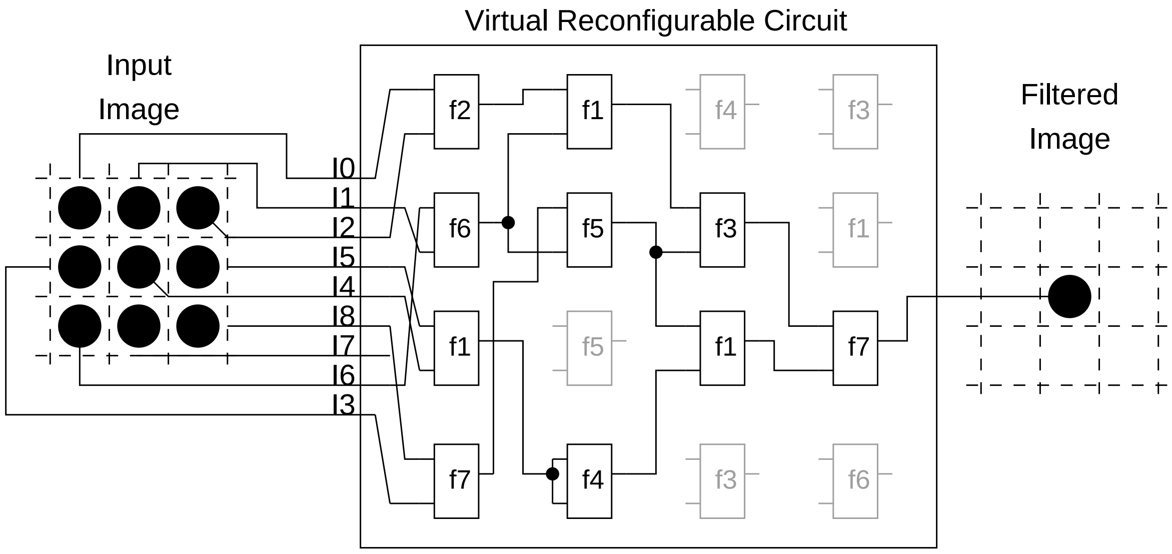


Fig. 5.7. A 3 × 3 image operator

w. Filtered image is simultaneously stored into the additional SSRAM memory. Note that all image data are stored in external SSRAM memories due to the limited capacity of internal RAMs available in the FPGA chip.

The design objective is to minimize the difference between the filtered image and the original image. The image size is $K \times K$ pixels but only the area of $(K-2) \times (K-2)$ pixels is considered because the pixel values at the borders are ignored and thus remain unfiltered. The fitness value of a candidate filter is obtained similarly to author's original work [90], i.e.: (1) the VRC is configured using a candidate chromosome, (2) the circuit created is used to produce pixel values in the image v, and (3) the fitness value is calculated as

$$\Phi = \sum_{i=1}^{K-2} \sum_{j=1}^{K-2} |v(i,j) - w(i,j)|. \tag{5.2}$$

The above formula is implemented as a special circuit in the FPGA. Similar circuits also ensure reading the original and corrupted images from external memories. This task is not trivial if we consider that nine pixels are needed at each moment of processing and these pixels are not stored at neighboring addresses of the memory. Therefore, special addressing circuits, FIFOs and comparators (used to detect the end of row and the end of image) are implemented.

The evaluation of candidate filters consists of three basic activities: (1) preparation of a new candidate chromosome (filter), (2) reconfiguration of VRC circuit according to the prepared chromosome, and (3) evaluation of the filter. As most time is spent in the filter evaluation, the architecture of evolvable image filter is designed in order to overlap the evaluation by other activities (1, 2). Therefore, because there is no overhead for reconfiguration of VRC (a new candidate configuration is prepared as well as VRC is reconfigured during evaluation of the previous filter), it is possible to express the time of evaluation of a single filter as:

$$t_{eval} = (K-2)^2 . \frac{1}{f} = (256-2)^2 . \frac{1}{50.10^6} = 1.29 \text{ ms} \tag{5.3}$$

if the size of images is 256 × 256 pixels and the FPGA operates at 50 MHz. Time of evolution can be expressed as follows:

$$t_e = t_{init} + g.n.t_{eval}, \tag{5.4}$$

where g is the number of generations, n is population size and t_{init} is time needed to generate the initial population (t_{init} is negligible). Considering $n = 4$ and $g = 10,000$, the time of evolution is 51.6 s for the 256 × 256 pixel-images with FPGA running at 50MHz.

5.4.3 Synthesis

In order to compare different implementations, we synthesized the whole system with VRC of size 4×8, 5×8, 6×8 and 7×8 CFBs. The evolutionary algorithm operates in the same way for all implementations; however, the size of chromosome depends on the number of CFBs. Table 5.5 shows the resources increase in FPGA with the increasing number of CFBs.

Table 5.5. Results of synthesis for Virtex II xc2v3000

VRC size	func. gens.	Dffs or latches	block RAMs	IO Blocks
avail	28,672	28,672	96	684
4x8	10,331	3,104	2	236
utilization	36%	10%	2%	34%
5x8	12,324	3,269	2	236
utilization	42%	11%	2%	34%
6x8	15,861	3,468	2	236
utilization	55%	12%	2%	34%
7x8	17,753	3,632	3	236
utilization	61%	12%	3%	34%

5.4.4 Results

Because of stochastic nature of the method, it is impossible to evaluate the proposed architecture for all possible types of images, noise, time of evolution and setting of CGP parameters. The following experiments were arranged to evaluate the most important properties of the evolvable unit, such as time of evolution and the quality of evolved filters.

Benchmark Problems. The following problems were utilized as benchmark tasks for the proposed architecture (see examples in Table 5.8):

- Noise removal from images corrupted by Gaussian noise ($\sigma = 16$)
- Noise removal from images corrupted by salt and pepper noise (5% pixels with white or black shots)
- Noise removal from images corrupted by random shot noise (5% pixels with a random value of shot)
- Edge detection

These benchmark problems were utilized in literature dealing with the evolutionary design of image filters [143, 93]. Conventional realizations of these filters are based on mean and median filtering. There are many conventional edge detectors, for example, Sobel and Canny detectors. References [94, 106] discuss these conventional filters and their implementation cost in hardware.

In our experiments, we utilized Lena image in the fitness function. The resulting filters were verified using a test set consisting of various images. The quality of filtered images is expressed in terms of *mean difference per pixel* (*mdpp*). This value is obtained by dividing the fitness value (Eq. 5.2) by the number of filtered pixels.Alternatively, signal-to-noise ratio (*snr*) is used in literature; however it is easier to implement a circuit calculating *mdpp* than *snr* in the FPGA.

Parameters of the Genetic Unit. A number of experiments have been arranged to find suitable parameters of Genetic Unit. We learned that it is useful to mutate 2.5% of the chromosome in average. In average, the best *mdpp* is obtained for 5×8 CFBs.

Another issue is whether it is better to maintain larger population or to produce more generations (considering that 160k fitness evaluations is allowed). For the salt and pepper noise problem, we repeated the filter evolution 200 times for each setup using the 128×128 pixel Lena image, mutation rate 10 bits/chromosome and with 4×8 CFBs in VRC. In our implementation, large populations imply lower *mdpp*s in average; however, it is difficult to obtain at least one very good solution. Thus we can suggest that if one is going to use the evolvable filter in FPGA to find a really good filter, it is better to generate more generations with a small population size. On the other hand, when the system should be used to achieve "real-time" adaptation, it is better to maintain larger population because the probability that the resulting solution is "good" is higher (assuming the same time available to the evolutionary design in both cases).

Another question is how many pixels the training image should contain. Naturally, larger training images should lead to more general filters; on the other hand, the evaluation of many pixels in the fitness function makes the evolution very time consuming. In comparison with the 256×256-pixel training image, the evolutionary process will be 4 times faster if only 128×128 pixels are examined, 16 times faster for 64×64 pixels and 64 times faster for 32×32 pixels. Experiments we arranged have confirmed that the use of the 128×128 pixel-training image is reasonable because the evolved filter are general enough. The usage of smaller training images leads to filters that are not general (i.e. they are optimized only for the given training image). However, we can imagine a real-world application in which it could be useful to have a filter optimized not only for a given type of noise but also for certain image(s). For example, a camera situated over a production line (say, with bottles) scans very similar types of images (bottle by bottle). The filter evolved using a small training image would be an advantage because of short adaptation time (a few seconds).

The quality of filters evolved for different types of noise. Table 5.6 summarizes the results we obtained in searching for the best filter for Gaussian, salt and pepper, random shot noise and for edge detector. We repeated the evolution 500 times (50k generations in each run) for all problems using the 256×256-pixel Lena image, population size 4, mutation rate 10 bits/chromosome, and 4×8 CFBs in VRC. The third column of the table indicates the generation in which the best filter was discovered. The last column represents the average number of generations in which the evolution has stagnated. We applied the evolved filters on other test images. The figures included into Table 5.7 demonstrate that the visual quality of the filtered images is sufficient and so the filters are general enough. All types of noise and examples of filtered images are shown in Table 5.8.

Table 5.6. The best filters evolved for test problems

Problem	best *mdpp*	#gen.	avr. *mdpp*	std.dev.	avr.#gen.
Salt&pepper	0.31	49808	0.95	0.51	29388
Random shot	1.11	40616	1.65	0.31	26781
Gausian noise	6.36	27179	6.73	0.24	31032
Edge detection	1.16	49608	1.73	0.41	35074

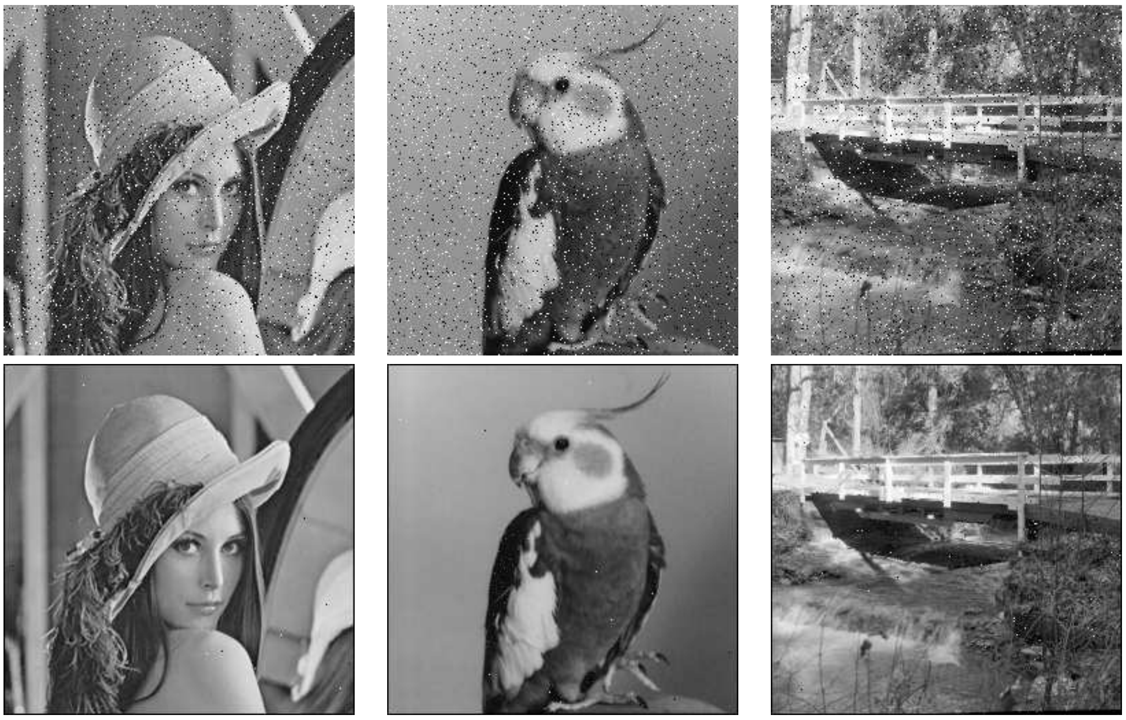


Table 5.7. The salt and pepper noise filter was evolved using Lena image and tested on other images

Overall Performance. We were able to evolve more than 4500 image filters every day (considering a training image of 128×128 pixels, population size 4 and 30k generations in average) which is totally impossible by means of a software approach. As the hardwired evolutionary design process is very fast, we could explore much larger portion of the design space than by using a conventional approach. The change of problem specification is very easy; the user has to supply only corrupted and uncorrupted images (these images are uploaded into SSRAMs at COMBO6) and some parameters of evolution (the mutation probability and the number of generations). In case that more changes are required (e.g. VRC is changed substantially), it is necessary to perform synthesis of the whole application which requires a few minutes.

From Table 5.6 can be derived that 30k generations (i.e. 20 seconds in FPGA operating at 100 MHz) are needed in average to find a filter. The design time is very reasonable if the proposed system should operate "instead" of a designer in the image filter design task. For some applications, our solution could also operate as a real-time evolving filter. If we consider that training images could consist only of 64×64 pixels then the time of evolution is 4.7 seconds. Note that the speedup we obtained against the software approach (Pentium III/800MHz) is 50 if the FPGA operates at 100 MHz.

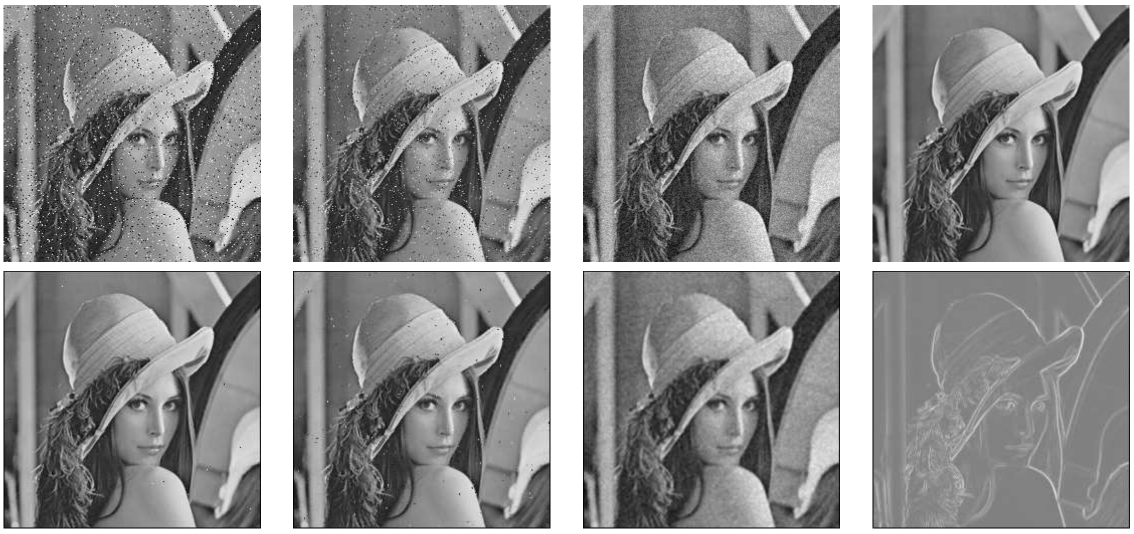


Table 5.8. Corrupted and filtered images: Salt and pepper noise, random shot noise, Gaussian noise, edge detector

Table 5.9 compares the best-achieved results with the results obtained using a software simulator [93] and with the conventional implementations. We can observe that the filters evolved here and in [93] exhibit very similar quality. In comparison with conventional implementations of image filters (i.e. to the median or mean filters that are general and not specialized to this problem), we obtained very good specific filters trained for a particular type of noise. Similarly to [94], we can assume that the evolved filters, when extracted from VRC, transformed to VHDL and synthesized again for FPGA, will be cheaper than conventional filters (in terms of the number of equivalent gates utilized in the FPGA).

Table 5.9. Comparison of *mdpp* of the best filters evolved in software and hardware

Operator	SW	HW	conventional
Gaussian	6.24	6.36	6.43 (mean)
Salt and pepper	0.38	0.31	2.95 (median)
Random Shot	1.08	1.11	2.98 (median)
Edge detection	1.20	1.16	–

5.5 Evolution of Sorting Networks in the FPGA

In Chapter 4.3.3, median circuits were evolved up to 25 inputs. However, the extrinsic evolutionary design is very time consuming for the circuits containing more than 20 inputs even if 32 candidate circuits are evaluated in parallel using a special encoding. Note that the evaluation time cannot be reduced by evaluating only a subset of all possible input combinations because this approach does not guarantee that the resulting circuit is general.

The objective of this study is to evolve as large sorting networks (circuits slightly complicated than the median circuit) as possible in a reasonable time. In order to perform these investigations, a novel virtual reconfigurable circuit architecture optimized for evolution of sorting networks has been proposed and implemented on the top of a conventional FPGA.

The chromosome encodes the functions performed by virtual programmable elements; however, in contrast to the VRCs reported in the previous chapters, the interconnection of these elements remains fixed. Since the FPGA implementation of the programmable element is inexpensive, it can operate as a wire and thus, in fact, the evolutionary algorithm also modifies the interconnection. As the fitness calculation is also carried out in the same FPGA, we can benefit from pipeline processing allowing reasonable time of a candidate circuit evaluation. The main objective is to find as large correct sorting network as possible in minimal time; neither area nor delay are optimized.

5.5.1 Virtual Reconfigurable Circuit

The VRC consists of elements that can perform different operations according to the selected configuration. Figure 5.8 shows its interface.

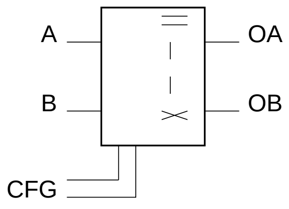


Fig. 5.8. A single element of VRC

Every element consists of two input and two output ports (everything is 1 bit). The functionality is determined by two configuration bits, which are used to select one of four different operations. All operations can be performed in one clock cycle. The result is stored into the register (local in each element) which offers to use VRC elements in a pipelined structure.

Sorting networks are usually composed of the Compare&Swap components with various interconnection. For this reason, the proposed VRC element is designed to perform the Compare&Swap operation. Alternatively, it can operate as a wire or cross-wire. The relation between configuration bits and functionality is shown in the following list:

- 00 – Direct connection from inputs to outputs
- 01 – Compare&Swap operation – maximum on the upper output, minimum on the lower output
- 10 – Compare&Swap operation – minimum on the upper output, maximum on the lower output
- 11 – Cross connection inputs to outputs

The VRC unit is composed of VRC elements in a fixed structure which is shown in Figure 5.8. Although the interconnection is invariable it can be changed if CFG = 00 or 11 is selected.

A VRC element is connected to the four nearest neighbors. Even columns have inputs and outputs shifted by one item. This one-item-shift is necessary to establish interconnection between arbitrary two Compare&Swap components or to compare different items. The architecture for odd Ns differs only in the first and last row where the remaining item is always connected to the next column via the synchronization register.

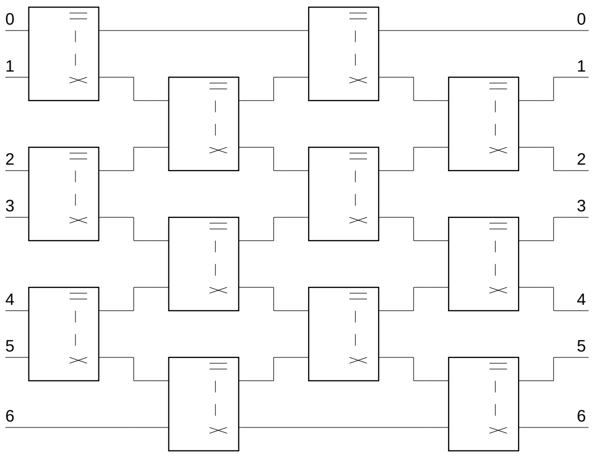


Fig. 5.9. The VRC array for evolution of sorting networks

The proposed VRC array supports fully pipelined processing because every element contains synchronization register for the output values. Therefore, the VRC can produce one result in a clock cycle. Unlike VRCs presented in the previous chapters this VRC architecture is also optimized for hardware resources. Only two LUTs are utilized to create one VRC element. This optimization enables to fit a large VRC within the single chip and thus to find sorting networks with many inputs.

5.5.2 Fitness Function

The Fitness unit is used to generate unsorted sequences and evaluate the results calculated by the VRC unit. N-bit counter generates the unsorted input vectors, i.e. all possible combinations over N bits. In the evaluation process, all vectors unsorted by VRC (i.e. by a candidate sorting network) have to be identified.

The fitness value is defined as the number of unsorted vectors coming from the VRC unit. The unsorted vector is detected if $a_i < a_{i+1}$ does not hold for any item a_i in the vector. This is performed in parallel by a set of comparators in one clock cycle. The number of unsorted vectors is stored in a counter which is incremented when an unsorted vector is identified. The content of the counter is presented as a fitness value which is valid after what all input vectors are evaluated.

5.5.3 Synthesis

The proposed architecture is designed for evolution of sorting networks having the number of inputs variable. From this point of view, the size of the VRC has to be scalable and support as many rows and columns as possible.

The synthesis results in Table 5.10 show hardware resources utilization of the XC2V3000 FPGA for different sizes of the VRC and N. It can be seen that the FPGA utilization is is 86% for the largest VRC consisting of 18×108 elements.

5.5.4 Results

Various VRC architectures were synthesized up to $N = 20$. For every size of VRC, 80 independent experiments were performed and analyzed. We used a four-member population

N length	VRC Elements	Slices	Chip Utilization
10	5×30	1731	12 %
12	6×36	2262	15 %
14	7×42	2735	19 %
16	8×48	3207	22 %
18	9×54	3795	26 %
20	10×60	4431	30 %
22	11×66	5675	39 %
24	12×72	6412	44 %
26	13×78	7314	51 %
28	14×84	8173	57 %
30	15×90	9232	64 %
32	16×96	10223	71 %
36	18×108	12468	86 %

Table 5.10. XC2V3000 FPGA utilization for various VRCs and N

and produced 50,000 generations in each run. Only the mutation operator was used (4 bits inverted in average).

Table 5.11 shows that it is possible to find correct sorting networks (for relatively large Ns) in a reasonable time. The first column shows the vector length (i.e. N). The number of correct sorting networks discovered out of 80 runs is reported in the third column. In the 4th column there is the average number of generations needed to find the perfect solution (its standard deviation is given in 5th column). The last column indicates the time needed to generate and evaluate one candidate solution (i.e. 2^N test cases). For example, the evaluation of a candidate network requires 1.3 ms for $N = 16$ and 40 s for $N = 32$ (at 50 MHz).

N length	VRC size (elements)	# of perfect solutions	Average # of generations	Standard deviation	Evaluation time of one candidate
4	2×8	80	94	2.55	512 ns
6	3×16	80	458	31.44	1.28 us
8	4×16	80	2217	24.66	5.12 us
10	5×32	80	6378	65.66	20.48 us
12	6×32	76	8673	666.88	81.92 us
14	7×32	75	11322	718.55	327.67 us
16	8×32	66	19467	477.44	1.31 ms
18	9×64	20	25306	3732.77	5.24 ms
20	10×64	17	31344	150.00	20.97 ms

Table 5.11. Sorting networks evolved in FPGA

In order to evolve larger sorting networks, we applied an adaptive mutation. With respect to N we mutated $4 - 12$ bits per chromosome. If no improvement in fitness value is observed in last 1000 generations, the number of mutated bits is incremented by 2. If an improvement is observed, the mutation ratio is decremented back to the previous value. Table 5.12 presents some of the evolved sorting networks up to $N = 28$. The evolution of a 28-input sorting network requires more than 20 hours (at 50 MHz).

5.5.5 Discussion

We evolved relatively large combinational circuits (28 inputs, 28 outputs) from scratch in a relatively short time (about 20 hours) and in a relatively low-cost commercial off-the-shelf hardware. On the other hand, we used a lot of domain knowledge to solve this problem (the

N length	VRC size (elements)	# of generations	Evaluation time of one candidate	Total time of evolution
22	11×64	4044	83.89 ms	5.6 min
24	12×64	4804	335.54 ms	26.9 min
26	13×64	10027	1.342 s	3.7 h
28	14×64	13483	5.368 s	20.1 h

Table 5.12. Large sorting networks evolved in FPGA

usage of compare&swap components and invariable interconnection of components is typical only for this problem). The evaluation of a single candidate sorting network for $N = 28$ was compared against a highly optimized SW implementation running in Xeon 3 GHz. Our FPGA evaluation running at 100 MHz is 40× faster than the software approach.

In order to illustrate the performance of our system, consider Koza's approach [59]. Koza et al. evaluated candidate sorting networks in Xilinx XC6216 FPGA. Genetic programming software utilized for the design of sorting networks was running in PC. For example, using population size 60k, minimal 8-input sorting network was evolved in generation 58, and using a population size 100k, the minimal 9-input sorting network was evolved in generation 105. The evolution of minimal 7-input sorting network required 69 minutes on the FPGA (31 generations, population size 1000). The evaluation of a candidate sorting network in XC6216 FPGA was 46 times faster than in Pentium 90MHz [59].

5.6 Summary

The three evolvable units were implemented in an FPGA. The first one has operated at the gate level, the second one at the functional level. The third unit contains modifiable programmable elements placed in a fixed-structure of connections. The first two units allowed the changes in interconnection of programmable elements. We have obtained a significant speedup against the software implementations. The proposed architectures, based on VRCs, represent a very constrained approach to the evolutionary circuit design since they only implement the software model in hardware. Therefore, it is practically impossible to obtain a circuit that cannot be evolved in software.

We have also to mention that it was not our goal to minimize the number of gates used in the evolved circuits. Considering evolvable adaptive systems (in which the evolutionary algorithm is a part of the target system), there is usually no requirement to minimize the number of elements utilized in evolved circuits. All the programmable elements physically exist in the system and they are available for free for the evolutionary purposes (as opposed to the strategy used in the evolutionary design of a single circuit [94]). The advantage of the evolved circuits is that they are inherently pipelined, which is useful for processing large data sets. Except for the evolvable sorting network, the implementation cost of the unit is relatively high if compared to the size of evolved circuits.

A strongly generic approach was utilized during VHDL design. All the implemented units are parameterized using various constants (such as the size of chromosome, the number of mutations etc.). Therefore, it is easy to modify the design and to obtain a totally different evolvable system in a very short time. The FPGA communicates with PC via a special software allowing the designer to prepare scripts describing experiments that have to be performed. Typically, designer specifies the VRC, EA and fitness function, perform synthesis, upload the evolvable system into FPGA and execute all experiments described in scripts. This approach can be considered as user-friendly interface to evolvable hardware (for example, a system for hardware evolution of sorting networks was derived from the evolvable image unit in a few hours). Furthermore, the evolvable units can be offered as an IP (intellectual property) cores to FPGA designers. Another important feature is that the evolvable unit can be uploaded to the FPGA dynamically only when some adaptation is needed. Once the system is adapted, only the resulting circuit remains in the FPGA (e.g. in the second VRC realized in FPGA) and some other circuits can replace the evolvable unit in the FPGA. This approach has been called *evolvable computing* [97]. We can summarize that hypothesis H2 was confirmed.

6. Intrinsic Analog Evolvable Hardware Using FPTAs

Although the previous chapter has dealt with an intrinsic approach, we were not interested in exploiting physical characteristics of FPGAs (as Thompson et al. did [122]). The main objective was to make the evolutionary design faster. Therefore, the evolutionary design process has worked in the same way as a corresponding circuit simulator should work.

In order to investigate whether computational elements such as logic gates, latches and flip-flops can be evolved at sub-gate level, we utilized JPL's Field Programmable Transistor Array FPTA-2. As far as the reconfigurable device is analog, it is natural for evolution to exploit physical properties of transistors, configurable switches, other components and environment to find the required behavior. Typically, the evolutionary process escapes the space of implementations we are able to represent in advance (compare Figures 6.1 and 4.1) and produces circuits which are difficult to understand and reconstruct in a circuit simulator. Furthermore, we will investigate whether these elementary computational elements can be evolved in extreme low temperatures.

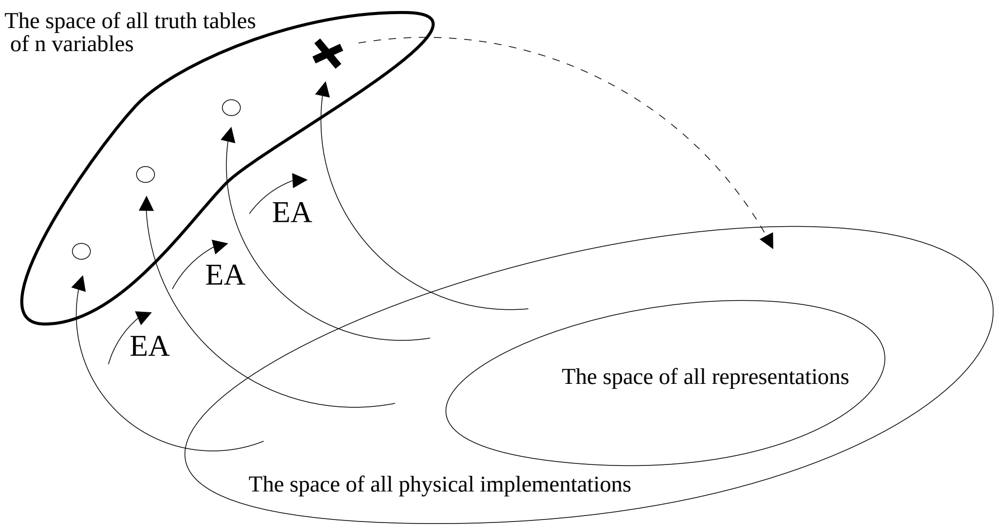

The space of all truth tables
of n variables
EA
EA
EA
The space of all representations
The space of all physical implementations

Fig. 6.1. Evolutionary search in the space of possible physical implementations

6.1 Field Programmable Transistor Array FPTA-2 and SABLES

For research purposes, the evolvable hardware group at JPL has developed a series of chips that are classified as evolution-oriented reconfigurable architectures (EORA) [111, 112]. Field

Programmable Transistor Array FPTA-2 was fabricated in 2002 using TSMC 0.18u 1.8V technology. The FPTA-2 has transistor level reconfigurability, supports any arrangement of programming bits without danger of damage to the chip (as is the case with some commercial devices). The FPTA-2 consists of an 8 × 8 array of reconfigurable cells. Each cell has a transistor array as well as a set of other programmable resources, including programmable resistors and static capacitors. Figure 6.2 and 6.3 provide a broad view of the chip architecture together with a detailed view of the reconfigurable transistor array cell. The reconfigurable circuitry consists of 14 transistors connected through 44 switches and is able to implement different building blocks for analog processing, such as two- and three-stage operational amplifiers, logarithmic photo detectors etc. Note that the architecture of FPTA is adjusted to implement analog circuits, not digital ones. Each cell is configured using five 16bit words; however, only 77 bits are utilized. Configurations are sent to the FPTA-2 via a configuration port (16bit data bus, 9bit address bus). The FPGA-2 uses 96 inputs (cells have got 1-2 external inputs) and 64 outputs (one for each cell). Details of the FPTA-2 can be found in [111].

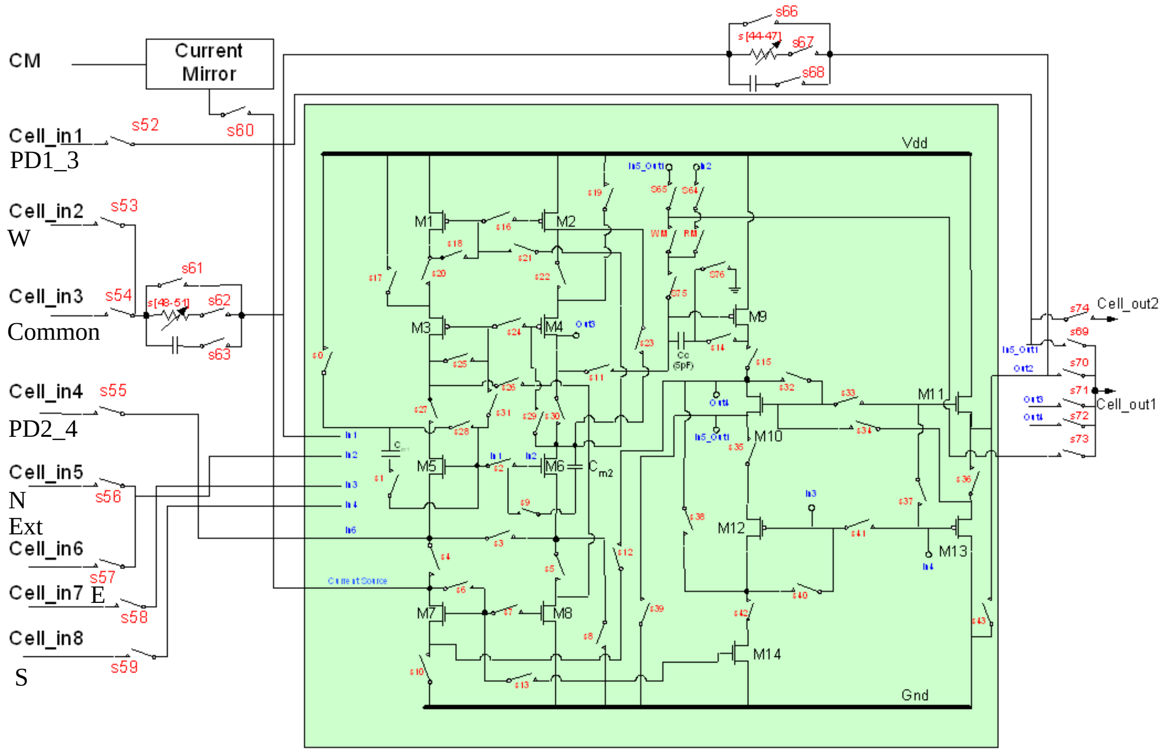


Fig. 6.2. A reconfigurable cell of FPTA-2. Its inputs come from four neighboring cells (N, S, W, E) and primary inputs (Common and Ext)

Stand Alone Board Level Evolvable System (SABLES) integrates an FPTA and a DSP implementing the evolutionary platform (Fig. 6.3). The system is stand-alone and is connected to the PC only for the purpose of receiving specifications and communicating back the results

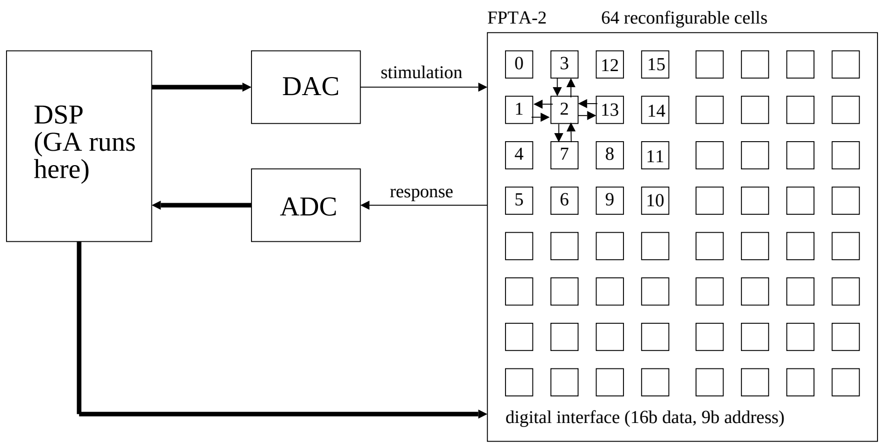


Fig. 6.3. Evolutionary design using FPTA-2 in the SABLES

of evolution for analysis. Evolutionary algorithm (running in DSP TI 320C6701) works with the population of candidate solutions, i.e. configurations for FPTA-2. In the evaluation process of an individual, stimulations are generated, responses collected and compared against the desired values.

Stoica reports that over four orders of magnitude speed-up of evolution was obtained on the FPTA-2 chip compared to SPICE simulations on a Pentium processor (this performance figure was obtained for a circuit with approximately 100 transistors; the speed-up advantage increases with the size of the circuit) [112].

6.2 Evolution of Primitive Gates

Evolutionary design of primitive gates at the transistor level was performed in [141, 109, 64]; here we repeat the approach to illustrate the basic ideas. A single cell of FPTA can be configured to operate as inverter. In order to evolve the two-input gate $y = f(A, B)$ we utilized two cells of FPTA (cell 0 and 1). The input of cell 0 was connected to A and the input of cell 1 was connected to B. The output y is read from the output of cell 1. Genetic algorithm operates over configuration bits of the two cells; those 2×77 bits define functionality of cells as well as their interconnection. Parameters of GA are as follows: population size 100; crossover probability 70%; mutation probability 10%; 1000 generations were produced in each run. The fitness value is calculated using formula:

$$\Phi = \sum_{i=1}^{K} |P(i) - T(i)|. \tag{6.1}$$

DSP generates K (e.g., $K = 240$) samples that are converted using D/A converter to stimulate the FPTA. All possible input combinations are considered. FPTA's responses are converted using A/D converter and compared against the desired values. The objective is to minimize the difference between target signal T and signal P generated by the FPTA.

Figure 6.4a shows the behavior of one of evolved NAND gates. A correct functionality was obtained for almost all runs for NAND, NOR, AND and OR gates. We have not been able to evolve an exclusive-or gate using this setup.

Figure 6.4b shows the output of the gate when a fault is injected artificially into the cell 0 (seven configuration bits were inverted in this case). GA is usually able to recover the original function for this type of problems. Similar experiments were performed in [54].

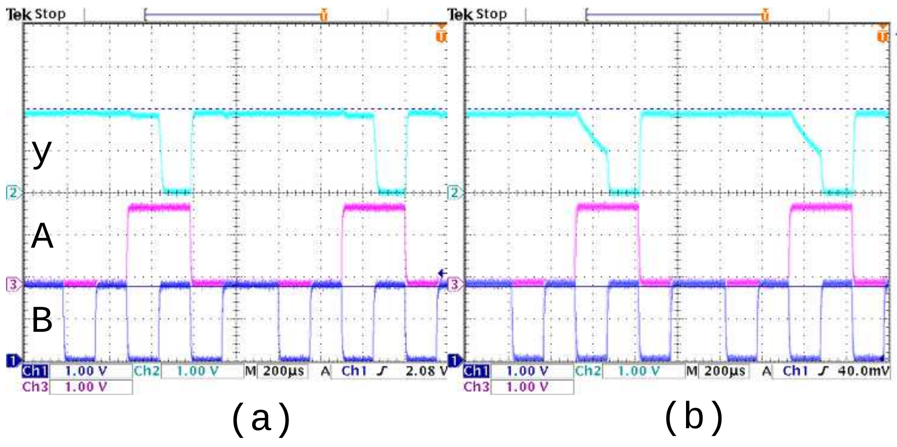


Fig. 6.4. (a) Behavior of an evolved NAND gate, (b) a fault injected

6.3 Evolution of Sequential Circuits

The evolutionary design of sequential circuits is considerably less mature than the evolutionary design of combinational circuits. This task is usually considered as more difficult than the evolution of combinational circuits, because: (1) Longer chromosomes are needed in order to encode feedbacks. Longer chromosomes usually imply larger search spaces that are usually difficult to search. (2) The fitness calculation is more complicated, because in addition to testing all input/output combinations, it has to take into account the internal states. The evaluation time doubles by incrementing the number of internal states.

Sequential circuits are crucial for implementation of digital computational devices because they represent the circuit implementation of an abstract concept of the discrete state. Only in some cases they were evolved (see Table 6.1). We can observe that no result is available for the evolution at the sub-gate level (e.g. transistor level).

Table 6.1. Typical sequential circuits designed using an evolutionary approach

Circuits	Level	Reference
D-latch	Gate	[32]
Freq. divider, detectors, serial adder	PLD	[74]
Seq. Filters	Functional	[123]
Quadrature decoder	Look-Up Tables	[70]
Counters, detectors	Gate, flip-flop	[3]

An open question addressed by this chapter is whether the evolutionary approach can rediscover the concept of the discrete state in a physical hardware, which can be reconfigured at the transistor level (H3). The evolution must be able to recognize that the required outputs are not pure combinations of the input values and to build internal structures to store the state. One possible application is that this approach may open the door to building new sequential circuits with a rich/non-linear dynamics in a more compact way than traditional combinational-memory way offers.

6.3.1 Objectives and Experimental Setup

The objective is to evolve Reset-Set circuit (RS) and D (data) latch circuit in the FPTA. The RS circuit is the simplest sequential circuit which is able to hold a logic value. The D-latch is a sequential circuit typically utilized in registers and counters. Figure 6.5 shows the specification and conventional implementation of these circuits at the gate level.

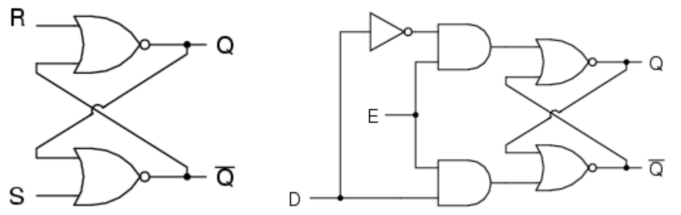


Fig. 6.5. Specification and conventional implementation of (a) RS circuit: if (R=1 and S=0) then Q=0; if (S=1 and R=0) then Q=1; if (S=0 and R=0) then hold the previous state; (b) D-latch: if E=1 then Q=D; if E=0 then hold the previous state.

The RS and D-latch circuits will be designed using a standard genetic algorithm operating directly with configurations of FPTA-2. Only a few cells of the FPTA will be utilized for the experiments. In the fitness evaluation, a sequence of input signals consisting of 12 combinations of logic values (a training set) is applied at the circuit inputs. The genetic algorithm must minimize the differences between the produced and required output values (see Fig. 6.1). In particular, K=240 values are sampled, digitized and utilized during the evaluation of a candidate circuit. The training input values can be seen in the following snapshots from the oscilloscope. In contrast to combinational circuits, the evolved sequential circuits must be able to produce different output values for the identical input values, depending on the state of the circuit. It represents the main difficulty. The evolved circuits are also verified using various test input sequences.

Two strategies are considered in order to accomplish the objectives: (1) Components (gates) are evolved separately and then connected together to build a sequential circuit. (2) A complete sequential circuit is evolved from scratch.

6.3.2 RS Circuits Composed of the Evolved Gates and Conventional Inverters

The idea behind this experiment is that two evolved gates can be connected in order to establish the RS circuit. Therefore, two independent AND gates were evolved together (which is possible using four cells). The gates have to be as similar as possible; hence only the configuration bits of cells 0 and 1 were evolved; the cells 4 and 5 were configured according to cells 0 and 1 (see Fig. 6.6a).

Then we connected the evolved gates according to Fig. 6.6a and assumed that the behavior will correspond to the behavior of the RS circuit. Fig. 6.6a also shows that the outputs of gates were connected through a conventional inverter (occupying a single cell of FPTA-2).

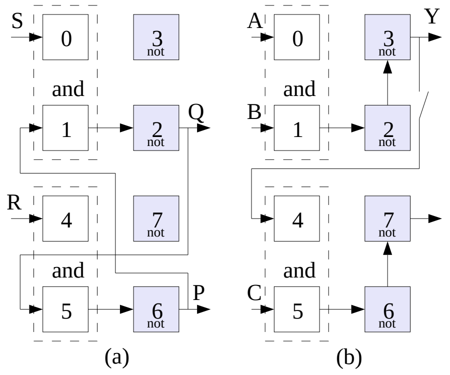


Fig. 6.6. Two evolved AND gates and conventional inverters used to compose (a) RS circuit and (b) a three-input gate

The reason for doing so is twofold: (1) NAND gates are needed for obtaining the RS circuit. (2) We have assumed that the inverters could help to make the output voltage levels better.

Fig. 6.7 shows the best behavior we have obtained. Unfortunately, it is wrong. We can observe an undefined voltage value (Vdd/2) in case that the circuit should hold the previous state. Subsequently, we recognized that the evolved gates neither work correctly as a three-input gate (i.e. without a feedback and independently of the use of inverters) when connected to make a three-input gate (see Fig. 6.6b). Despite of a great effort to evolve and connect two gates together in order to create a larger circuit, we have never been successful. It seems that the main problem is that the evolved two-input gates do not show standard behavior (i.e. fan-in and/or fan-out are not sufficient). Note that we have also attempted to evolve a two-input gate surrounded by conventional invertors (in order to ensure a good "environment" for all inputs and outputs) and considered a load during evolution. We performed these experiments for various target gates and under various experimental setups; however, with no success.

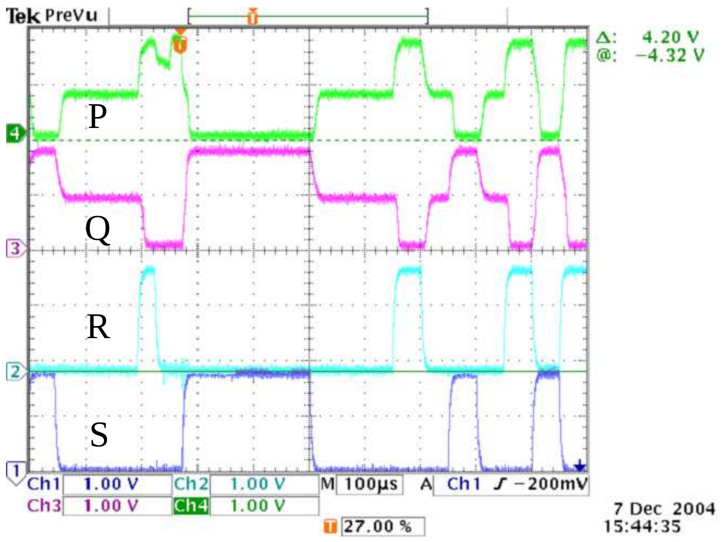


Fig. 6.7. Behavior of the "RS circuit" that was created from evolved AND gates and conventional invertors

6.3.3 RS Circuit Evolved From Scratch

Since no RS circuit was obtained using the approach based on two-input gates evolved separately, we decided to evolve the RS circuit from scratch (see Fig. 6.8). Again, only the configuration bits of cells 0 and 1 (i.e. 2×77 bits) are stored in the chromosome. In order to establish a candidate circuit consisting of four cells, the configuration bits of cells 0 and 1 are copied into cells 3 and 2. The solid lines in Fig. 6.8 denote external physical connections (wires) used to connect the cells. These connections were utilized to promote a specific design pattern which is typical for elementary sequential circuits (see Fig. 6.5). In addition to these connections, the evolution was left to interconnect the cells using the internal switches of the FPTA. Parameters of GA are as follows: population size 100, crossover probability 70% and mutation probability 10%. Depending on experiment, 300–1000 generations were produced.

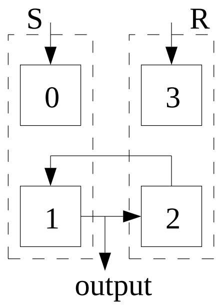


Fig. 6.8. Experimental setup for the evolution of the RS circuit using 4 cells (from scratch).

Figure 6.9 shows the behavior of the two best RS circuits we evolved. When both input values are at logic 0 and the previous output value is at logic 1, the circuit is still able to hold the logic 1. This value is not as strong as if $S = 1$; however, it is still possible to consider the output value as correct. As Figure 6.9 shows, it is easy to improve the output value using an additional standard inverter gate. The evolved circuits were tested using various input sequences generated for the same time domain. We have found very difficult to evolve a correct RS circuit (approximately one successful run out of 30 runs).

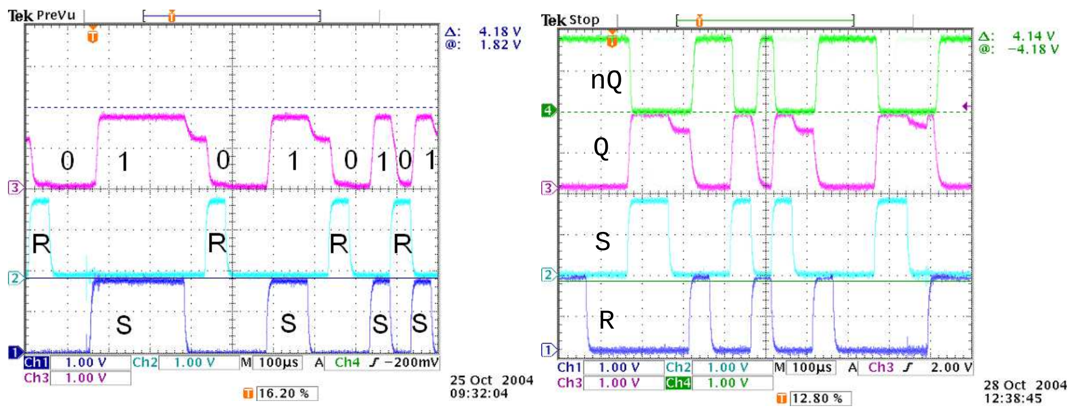


Fig. 6.9. Behavior of two different RS circuits evolved form scratch. nQ is obtained from a conventional inverter connected to Q.

6.3.4 D-Latch Evolved From Scratch

Figure 6.10 shows five cells and their physical interconnection that we utilized to evolve D-latch from scratch. The evolution could also interconnect the cells using the internal switches. The chromosome contains configuration bits of cells 2, 4 and 5. The configuration bits for cell 7 (6, respectively) are copied from cell 4 (5, respectively).

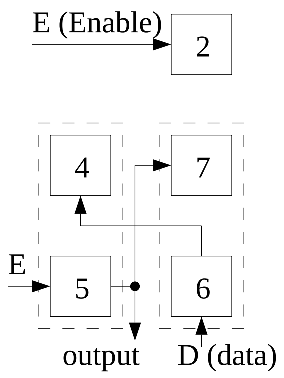


Fig. 6.10. Experimental setup for the evolution of D-latch circuit (from scratch).

Although we performed more than 100 experiments, we obtained only one close-to-perfect D-latch. Figure 6.11 shows its behavior for the training set. The output values are not perfect; however, they can be improved by means of two conventional inverters connected to the output (see the upper signal in Fig 6.11). Figure 6.12a illustrates that the circuit also works for a test set. However, we were able to find a specific case for which the circuit does not work (see Fig. 6.12b). Hence the circuit cannot be considered as a perfect D-latch.

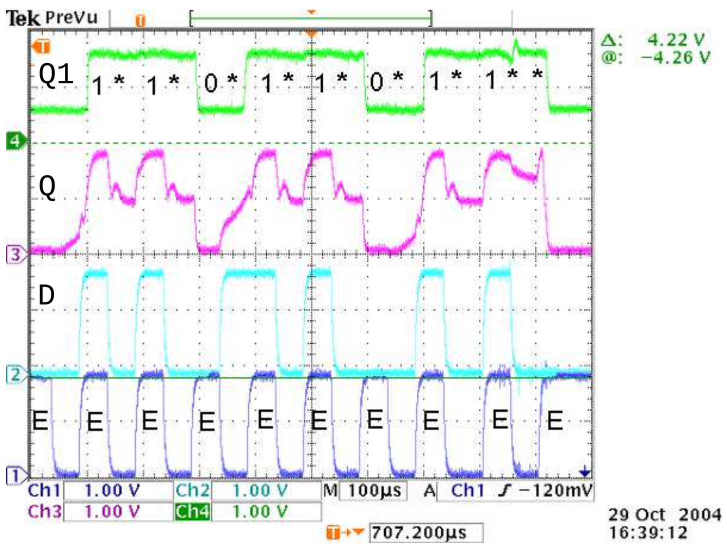


Fig. 6.11. Behavior of an imperfect D-latch evolved form scratch. Q1 is obtained from two conventional inverters serially connected to Q.

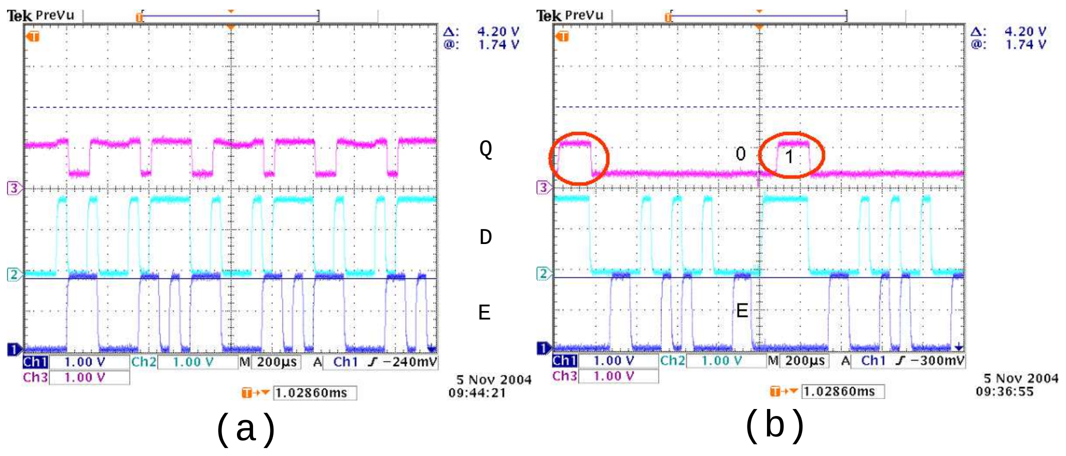


Fig. 6.12. Analysis of an imperfect D-latch evolved form scratch: (a) test – OK, (b) test – failed (the output should be at logic 0).

Fig. 6.13 shows two typical imperfect behaviors corresponding to two different "D-latch" circuits we evolved very often. In the case (a), logic 1 is weak, which means that the circuit has problems to hold logic 1 when the both inputs are set at logic 0. There are no problems to hold logic 0. On the other hand, in the case (b), logic 0 is weak and there are no problems to hold logic 1. The evolution very often converges to one of these results. It seems that it is very difficult for our GA and FPTA (perhaps impossible using the considered cells) to obtain something "between" these two behaviors which should correspond to the perfect D-latch.

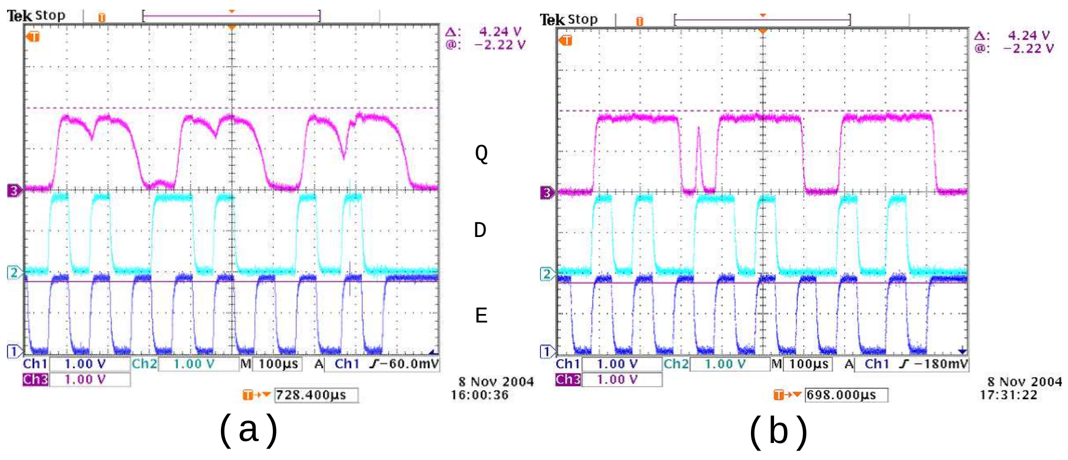


Fig. 6.13. Typical behaviors which the evolution of a D-latch often converges to: (a) weak logic 1, (b) weak logic 0.

6.3.5 Discussion

We have addressed the fundamental question whether the evolutionary approach is able to discover the concept of the discrete state at the sub-gate lavel (transistor level, in our case). Although the resulting circuits do not work perfectly, the answer is positive, i.e. the transistors available for the evolutionary design can be composed together by means of an automated evolutionary process in order to establish a simple sequential circuit. No surprise that it was easier to evolve the RS circuit than D-latch. The resulting circuits are not area-optimal, they do not probably operate correctly for various time domains and they can not easily be connected to some other circuits. However, those features were not required. We supplied sufficient resources and the evolution was able to discover the crucial concept—the internal state—directly in the reconfigurable transistor array.

In case of combinational circuits, there are usually many options how to put the available components together to obtain the required behavior. It seems that only several options exist for the sequential circuits. Their connection is very tricky and difficult to discover. Perhaps this is why we were not able to evolve these circuits routinely. Although we consider the method used as the evolution from scratch, we had in fact to supply some little domain knowledge in the form of "promoted design pattern" (i.e. the connection of external wires etc.). No sequential circuits were evolved without this domain knowledge.

6.4 Evolution of Computational Elements in Extreme Environment

Future NASA missions will face extreme environments, including environments with large temperature differences (–233°C and 460°C) and high radiation levels as 5MRad Total Ionizing Dose (TID) [142]. Conventional circuit design exploits device characteristics within a certain temperature/radiation range; when that is exceeded, the circuit function degrades. On a reconfigurable device, although component parameters change in extreme environment, a new circuit design, suitable for new parameter values, may be mapped into the reconfigurable structure to recover the initial circuit function.

Conventional approaches to extreme environment electronics include *hardening–by–process* (i.e. fabricating devices using materials and device designs with higher tolerance to extreme environment) and *hardening–by–design* (i.e. the use of special design/compensation schemes). Both these hardening approaches are limited, in particular for analog electronics, by the fact that current designs are fixed and, as components are affected by extreme environment, these drifts alter functionality [86]. A recent approach pioneered by JPL is to mitigate drifts, degradation, or damage on electronic devices in extreme environment by using reconfigurable devices and an adaptive selfreconfiguration of circuit topology. This new approach is referred as *hardening–by–reconfiguration.* As the only investigator in this area, JPL's evolvable hardware group has demonstrated for simple circuits created in the FPTA-2 that evolutionary approach can recover functionality lost (1) after a fault artificially injected into FPTA-2, (2) in extreme high temperatures and (3) in high radiation environment [54, 112, 113]. This section reports first results obtained for circuit evolution and recovery at extreme *low* temperatures (–196.5°C).

6.4.1 Electronics at Extreme Low Temperatures

The following brief survey of extreme-temperature electronics is based on [27]. The term extreme-temperature electronics is used here to mean electronics operating outside the "traditional" temperature range of –55/–65°C to +125°C. At the low end, operation of semiconductor-based devices has often been reported down to temperatures as low as about –270°C. This includes devices based on Si, Ge, GaAs and other semiconductor materials. On the high end, "laboratory" operation of discrete semiconductor devices has been reported at temperatures as high as about +700°C (for a diamond Schottky diode) and 650°C (for a SiC MOSFET). Integrated circuits based on Si and GaAs have operated to +400°C. Silicon integrated circuits have been reported to operate at +300°C for more than 1000 hours. Covering both extremes, there are reports of the same transistor working from about –270°C to about +350 to +400°C, an operating temperature span of over +600°C. At the low-temperature end, practical operation of devices and circuits is reasonably achievable to as low a temperature as desired, bearing in mind that materials and designs appropriate to the temperature must be used. The various characteristics of a device may improve or degrade. In particular, below about –230°C Si devices often exhibit significant changes in characteristics.

Semiconductor devices operate by means of the movement of charge carriers (electrons and holes). The key is controlling their movement through arrangement of n-type, p-type, and intrinsic regions (and insulators) that have different electrical properties. The lower temperature limit is typically determined by the ionization energy of the dopants. Dopants usually require some energy to ionize and produce carriers in the semiconductor. This energy is usually thermal, and if the temperature is too low, the dopants will not be sufficiently ionized and there will be insufficient carriers. The result is a condition called "freeze-out". For example, Si (dopant ionization energy 0.05 eV) freezes out at about 40K and Ge (ionization energy 0.01 eV) at about 20K. Thus, for example, Ge devices in general operate to lower temperatures than Si devices. The low-temperature limit of field-effect devices depends on the particular type and material. For example, characteristics of FETs (field-effect transistors) generally improve with cooling, such astransconductance, leakages, and white (high-frequency) noise; low-frequency noise is less predictable.

Although device parameters change in extreme environment, while devices still operate (albeit on a different point of their characteristic) a new circuit design, suitable for new parameter values, can be mapped into the reconfigurable system to recover the initial circuit functionality. Partly degraded resources are still used, while completely damaged resources are bypassed. The new designs, suitable for various environmental conditions, can be determined prior to operation or determined in-situ by reconfiguration algorithms running on a built-in digital controller. It is assumed that this controller is a digital circuit; therefore not so sensitive to extreme environment as analog electronics.

The low temperature test bed for these experiments used liquid nitrogen, establishing a temperature of –196.5°C. In order to study the effect of low temperatures on the FPTA device only (the DSP was at room temperature), the chip in a standard ceramic package was placed on a separate board that was immersed into liquid nitrogen. Temperature was measured directly at the die. In the proposed experiments, we do not use evolutionary algorithm to repair a circuit; rather we perform evolutionary design from a completely random initial population.

6.4.2 NOR Gate

A NOR gate was evolved at -196.5°C. Two FPTA cells were used and the experiment processed 100 individuals along 300 generations. We used the same experimental setup as in Chapter 6.2. Figure 6.14 shows the oscilloscope picture of the evolved solution at –196.5°C. The same solution was tested at room temperature using another FPTA chip, producing an almost identical behavior.

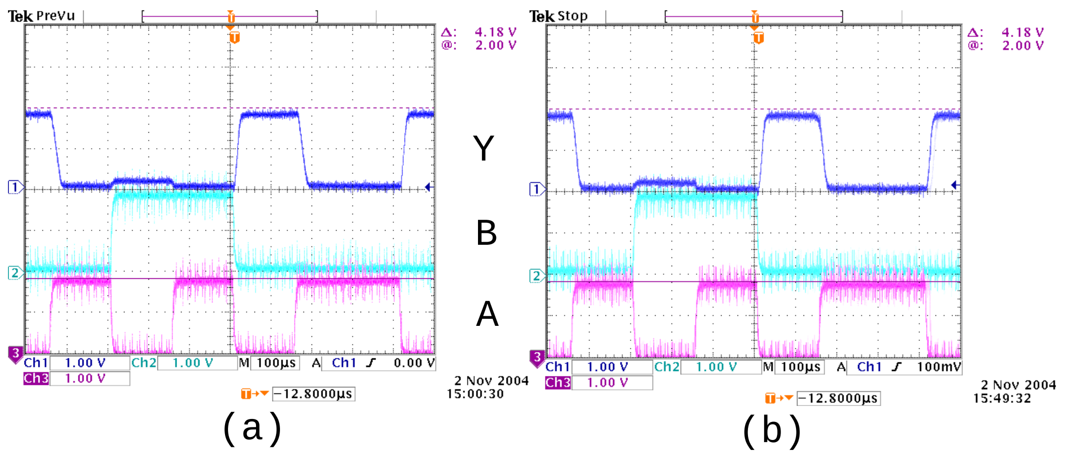


Fig. 6.14. (a) Behavior of NOR circuit evolved and tested at –196.5°C; (b) the same circuit was tested successfully at room temperature. An environmental noise signal is also present at the circuit input.

6.4.3 Controllable Oscillators

Figure 6.15 shows that four cells of the FPTA were used to evolve a controllable oscillator [100]. This circuit receives a digital input and it should oscillate when the input is at one digital level and stay at ground for the other level. No external components (such as RC circuits) were considered for these experiments. The frequency of oscillations depends only on the configuration and internal characteristics (such as delay of transistors) of FPTA-2.

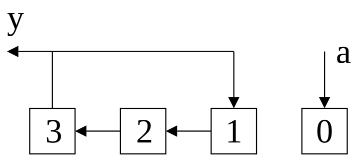


Fig. 6.15. Cells and connections used to evolve controllable oscillators

The genetic algorithm running in a DSP uses the roulette-wheel selection, crossover (70%) and mutation (10%). The genetic algorithm must promote the chromosomes that cause oscillations if they are required and keep the output invariable otherwise. In particular 240 values are sampled, digitized and utilized during the evaluation of a candidate circuit. Because of simplicity we decided to evaluate candidate circuits in the time domain. Oscillators controlled using input signal $a[i]$ are designed using the fitness function whose basic structure is given in the following pseudo-code:

```
Algorithm 6.1:
i = 0; fitness = 0;
while (i < samples)
{
  // oscillations
  ones = 0; zeroes = 0; penalty = 0;
  while (i < samples and a[i] is High)
  {
     if (y[i] < LL) zeroes = zeroes + 1;
     else if (y[i] > HL) ones = ones + 1;
     else penalty = penalty + 1;
```

```
    }
    fitness = fitness + $k_1$.abs(ones -- zeroes) + $k_p$.penalty;

    // no oscillations
    ones = 0; zeroes = 0; penalty = 0;
    while (i < samples and a[i] is Low)
    {
       if (y[i] < LL) zeroes = zeroes + 1;
       else if (y[i] > HL) ones = ones + 1;
       else penalty = penalty + 1;
    }
    fitness = fitness + $k_2$(zeroes + ones - abs(zeroes + ones)) + $k_p$.penalty;
}
```

If $a[i]$ is at logic 1, the circuit should oscillate; otherwise, the circuit should not. Here, $i = 1 \ldots 240$ samples are evaluated at the circuit output $y[i]$. The *zeroes* counter indicates the number of output values that are considered as lower than a given threshold value LL ($LL = 0.45M_V$ where M_V determines the maximum output voltage 1.8V). The *ones* counter indicates the number of output values that are considered as higher than a threshold value HL ($HL = 0.55M_V$). The situation in which the circuit should oscillate (i.e. the number of *zeroes* and *ones* is similar but non-zero) is evaluated in the first nested while loop. The second nested loop deals with the situation in which the output should not oscillate. *Penalty* counter is used to avoid staying in the middle of M_V range. The values of constants k_1, k_2 and k_p are determined experimentally, and $k_p \gg k_1 = k_2$.

Initially, a controllable oscillator was evolved at room temperature; the circuit behavior is depicted in Fig. 6.16a. The circuits output a 70kHz sine wave (with a small degree of harmonic components) when the input is at logic 0. When the same circuit is tested at –196.5°C, it can be observed a distortion (increase in harmonics) at the output (Fig. 6.16b). The controllable oscillator was evolved again at –196.5°C; the response being displayed in Fig. 6.17. It can be observed that the output distortion has largely been removed.

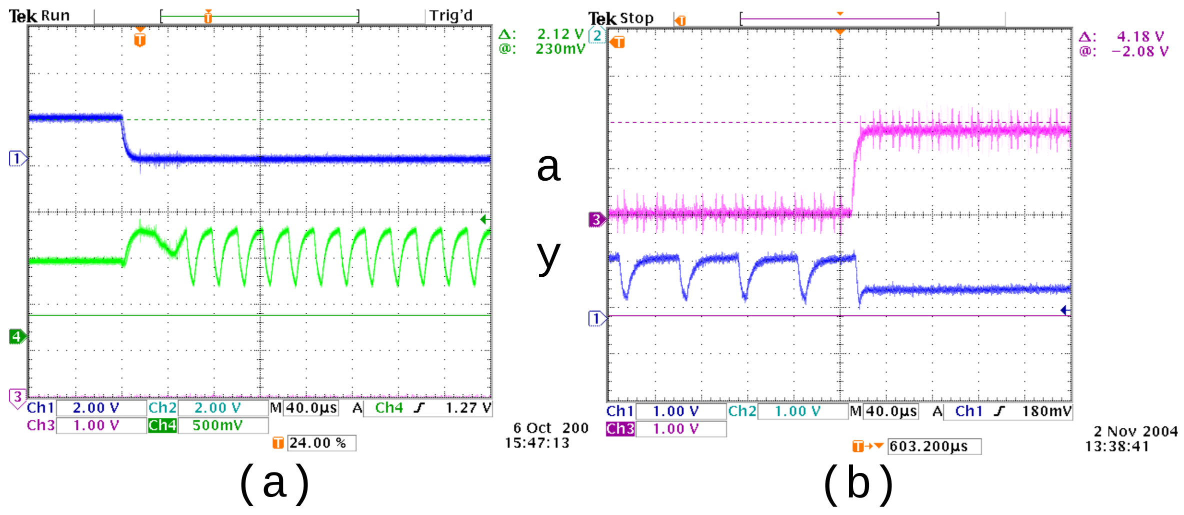


Fig. 6.16. Behavior of the controllable oscillator evolved at room temperature (a) and deteriorated response at –196.5°C (b)

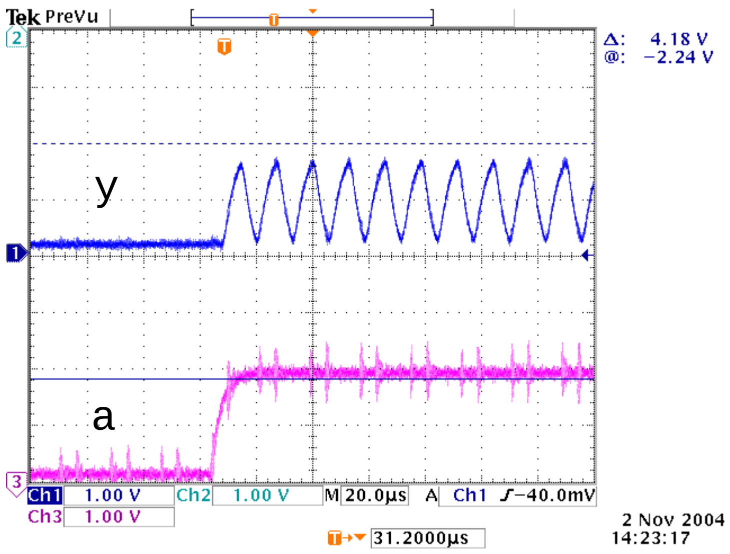


Fig. 6.17. Behavior of the controllable oscillator evolved at low temperature

6.4.4 Discussion

The results have proved the idea that EA could be able to find a suitable solution in extreme low temperatures. However, with the following limitations: 1) the tests were of short duration, 2) did not implement temperature cycling, 3) did not use the combined EHW system (DSP and FPTA) at low temperature simultaneously, 4) were not demonstrated on complex analog or digital circuits performing in an application. Particularly, the DSP board worked down to –110°C, but failed for further lower temperatures [142]. Next goal of evolvable hardware group at JPL is to test other Evolutionary Processor implementations, such as FPGAs, for an extended operation at –180°C. The objective is to develop and validate Self Reconfigurable Electronics for Extreme Environments technology by demonstrating a Self-Reconfigurable Analog Array (SRAA) integrated circuit in sustained (over 200 hours) operation at temperatures between –180°C and 120°C, and irradiated to 300 kRad total ionizing dose.

6.5 Summary

In this chapter, we have shown that the concept of discrete state can be re-discovered at the sub-gate level by the evolutionary algorithm. This result is important for future research in the area of evolutionary design of computational devices directly "in materio". Second, we have demonstrated that low temperatures influence behavior of a circuit uploaded into the FPTA and that the EA is able to form computational elements in this environment. Therefore, we confirmed hypotheses H3 and H4.

7. Evolution of Polymorphic Modules

A non-stable environment is usually seen as something what we do not like. Evolutionary experiments conducted at the transistor level in a variable environment have demonstrated that a single circuit can perform different *useful* functions in different environments. The papers [114, 116, 117] show that it is possible to design and effectively implement multifunctional digital gates whose function can be controlled in a non-traditional way: by temperature, power supply voltage (Vdd), some external signals etc. As an example, we can mention a novel topology created for the multifunctional NAND/NOR gate which operates as NOR in the case that Vdd=1.8V and as NAND in the case that Vdd=3.3V. No conventional design is available with this logic function controlled by Vdd [114]. There is a great potential for various applications of polymorphic electronics in many areas because devices composed of these gates are inherently adaptable to the changes of a particular environment and this feature is practically for free; with no reconfiguration overhead and external sensors.

The available literature describes only the implementations of polymorphic gates. No concrete circuits composed of these gates have been reported so far. The design using these unconventional gates seems to be a difficult task and no conventional design technique is available. We believe that compact circuits can be created by using the multifunctional gates as building blocks rather than by trivial multiplexing the outputs of several conventional modules using a polymorphic multiplexer (as shown in [103]). Hence the use of evolutionary algorithms could be a promising approach for the design of compact and useful adaptive circuits.

This chapter will illustrate that multifunctional digital combinational circuits can be evolved using the multifunctional gates as building blocks. It is assumed that suitable multifunctional gates exist. Only the bi-functional gates are considered here. The evolved circuits will perform the first required function in the first environment and the second required function in the second environment. The change of their functionality will be determined by the change of a control variable. For example, a circuit should operate as the adder for the temperature 30°C and as the multiplier for 200°C, i.e. the temperature is the sensitive variable in that case. Two approaches will be utilized to evolve polymorphic circuits: (1) CGP and (2) a simple developmental scheme as proposed in Chapter 4.4.

7.1 Polymorphic Gates

In polymorphic electronics, a function change does not require reconfiguration as in traditional approaches in which n different modules and a switch are needed to perform n different functions. Instead, the change comes from modifications in the characteristics of components (e.g. in the transistor's operation point) involved in the circuit in response to controls such as temperature, power supply voltage, light, etc. [116]. Polymorphic circuits are able to work in several modes of operation. In the most straightforward approach, there are only two modes (it will also be our case). The existence of digital polymorphic circuits is based on polymorphic gates. Table 7.1 gives examples of the polymorphic gates reported in literature. Most of them have been designed by means of evolutionary techniques.

The NAND/NOR gate is the most famous example [114]. The evolution obtained a creative novel topology more compact than by multiplexing NAND/NOR gate which is a conventional solution using a standard digital library with external voltage control. The circuit was fabricated in a 0.5-micron CMOS technology and silicon tests showed a good correspondence with the simulations. The circuit is stable for ±10% variations of Vdd and for temperatures in the range from –20°C to 200°C.

Table 7.1. Examples of existing polymorphic gates and their implementation cost

Gate	control values	control method	transistors	ref.
AND/OR	27/125°C	temperature	6	[117]
AND/OR/XOR	3.3/0.0/1.5V	external voltage	10	[117]
AND/OR	3.3/0.0V	external voltage	6	[117]
AND/OR	1.2/3.3V	Vdd	8	[116]
NAND/NOR	3.3/1.8V	Vdd	6	[114]

Potential applications would involve special circuits that are able to decrease resolution of digital/analog converters or speed/resolution of a data transmission when a battery voltage decreases, circuits with a hidden/secret function that can be used for security purposes, intelligent sensors, novel solutions for reconfigurable cells and function generators in reconfigurable devices (such as FPGA and CPLD), circuits of random number generators changing distributions of the probability according to the external environment and some others circuits as discussed in [116].

7.2 Problem Formulation

Let P be a set of polymorphic gates. Each of them is able to implement up to K functions (K is specified beforehand) according to a control signal which holds up to K different values. A gate is in *mode j* (and so performing the j-th function) in the case that j-th value of the control signal is activated. For purposes of this study, we will denote a polymorphic gate as $X_1/X_2/\dots/X_K$, where X_i is its i-th logic function. For example, NAND/NOR denotes the gate operating as NAND in *mode 1* and as NOR in *mode 2*. Note that some gates can perform less than K different functions; however, their function must be fully defined for each mode. For example, the conventional NAND gate considered for polymorphic circuits will perform the same function in all modes (denoted as NAND/NAND in the rest of this thesis).

A polymorphic circuit can formally be represented by the graph $G = (V, E, \varphi)$, where V is a set of vertices, and E is a set of edges between the vertices $E = \{(a, b)|a, b \in V\}$ and φ is a mapping assigning a function (polymorphic gate) to each vertex, $\varphi : V \to P$. As usually, V models the gates and E models the connections of the gates. A circuit (and also its graph) is in the *mode j* in the case that all gates are in the *mode j*.

Given P and logic functions $f_1 \dots f_K$ required in different modes, the problem of the polymorphic circuit design at the gate level is formulated as follows: Find a graph G representing the digital circuit which performs functions $f_1 \dots f_K$ in its modes $1 \dots K$. Additional requirements can be specified, e.g. to minimize delay, area, power consumption etc.

7.3 Evolutionary Approach Based on CGP

In our case we are looking for a circuit composed of polymorphic gates that performs f_1 in the first environment and f_2 in the second environment. The structure (topology) of the circuit is invariable; only functions of gates are modified when the environment is changed.

7.3.1 Experimental Setup

CGP is employed to solve this problem. Our algorithm operates with the population of 128 individuals; every new population consists of mutants of the best four individuals. Only the mutation operator has been utilized that modifies one randomly selected gene of an individual. In case that evolution has found a solution which produces the correct outputs for all possible input combinations, the number of gates is getting to minimize. Delay is not optimized. The evolution is terminated in case that no improvement of the best fitness value has been reported in a given number of generations (typically after 50000 generations). As these experiments deal with bi-functional gates, i.e. with bi-functional polymorphic circuits, the fitness value is obtained as follows:

1. Set all gates of a candidate circuit into *mode 1.*
2. Apply all possible input combinations at the circuit inputs and calculate the number of correct output bits B_1 obtained as response for these input values for the required function f_1.
3. Set all gates into *mode 2.*
4. Apply all possible input combinations at the circuit inputs and calculate the number of correct output bits B_2 obtained as response for these input values for the required function f_2.
5. Calculate $\Phi_1 = B_1 + B_2$.

After achieving the required behavior for f_1 and f_2, the number of gates is being minimized; the fitness value is defined as:

$$\Phi_2 = \Phi_1 + u - g \tag{7.1}$$

where g denotes the number of gates utilized in a particular candidate circuit and u is the total number of gates available.

7.3.2 Results

This section presents some interesting examples of polymorphic combinational modules that we evolved. We dealt with small combinational circuits (up to 5 inputs and 4 outputs) with various types of bi-functional polymorphic gates. In particular, the following circuits will be presented:

- 5-bit parity circuit vs 5-bit median circuit (denoted as 5b-parity/median)
- 2-bit multiplier vs 4-input sorting network (mult2b/sn4b)
- 2-bit multiplier vs 2-bit adder (2b-mult/add)

Table 7.2 summarizes the circuits obtained using the algorithm proposed in Chapter 4.2.1. The symbol u denotes the number of gates used in CGP. The column *corr. circuits* gives the number of correct circuits (i.e., those circuits perfectly working in both modes). The minimal number of gates we obtained for a particular circuit is given in the column denoted as *min. gates*. *Gate1* and *gate2* are the polymorphic gates utilized in the design process. On average, tens of thousands of generations are needed to find a solution. We learned that the correct circuits represent only a very small fraction of all circuits generated by evolution. In Figures 7.1, 7.2, 7.3 and 7.4, the label **0** denotes the first polymorphic gate and label **1** is the second gate of the two used. The bit 0 is LSB.

Table 7.2. Evolved polymorphic combinational circuits

Circuit	u	# of runs	corr. circuits	min. gates	gate1	gate2
5b-parity/median	24	200	16	14	NAND/NOR	XOR/XOR
5b-parity/median	20	900	48	13	NAND/XOR	XOR/NOR
Mult2b/sn4b	40	1000	8	25	NAND/NOR	AND/AND
Mult2b/sn4b	40	50	1	27	$(a \vee \overline{b})$/XOR	XOR/$(a \wedge \overline{b})$
2b-mult/add	40	200	11	20	NAND/NOR	OR/XOR
2b-mult/add	40	200	5	23	NAND/NOR	AND/AND

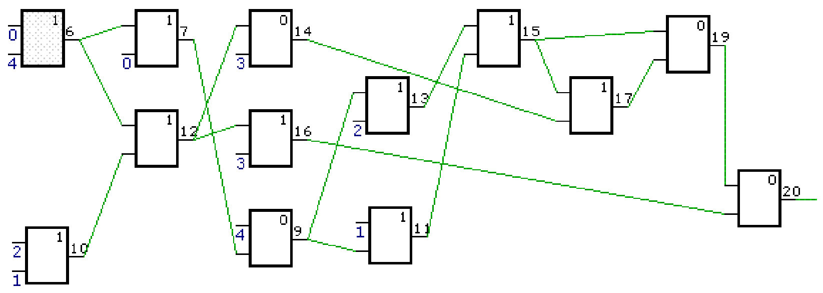


Fig. 7.1. Polymorphic 5b-median/parity circuit (the inputs: 0–4; gates: 0 – NAND/XOR, 1 – XOR/NOR)

7.3.3 Analysis of Evolved Circuits

No useful circuits utilizing only a single type of a polymorphic gate were obtained. As Table 7.2 indicates, it seems useful to combine polymorphic gates with conventional gates. The selection of suitable gates is also important for a particular polymorphic circuit. After our experimental work, we have identified suitable polymorphic gates for the presented circuits (see Table 7.2 for concrete circuits). We performed the selection manually; next research will be conducted to use the evolution to accomplish this task. We recognized that only a few combinations of gates are suitable for a given problem; most combinations do not lead to a solution. As an example, we can mention the Mult2b/sn4b problem. We performed 100 independent runs for each combination of polymorphic gates taken from the following list:

1. NAND/NOR and XOR/XOR
2. NAND/NOR and OR/OR
3. NAND/NOR and $(a \vee \overline{b})/(a \vee \overline{b})$

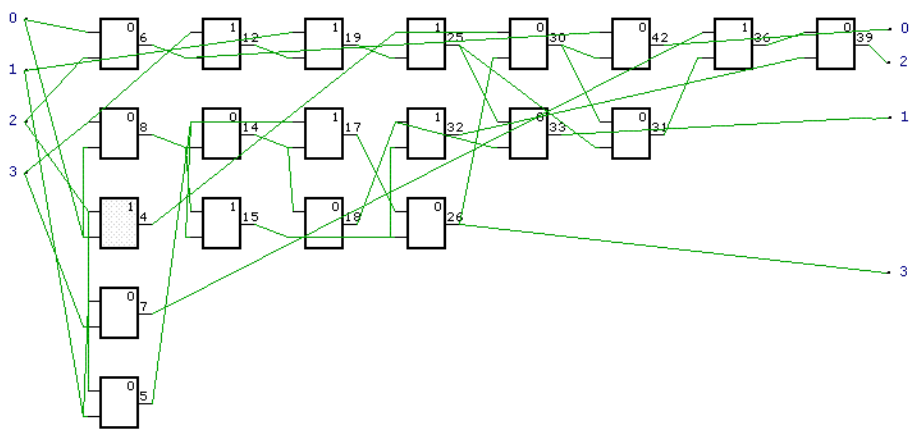


Fig. 7.2. Polymorphic multiplier/adder circuit (the inputs: A(0–1), B(2–3); the outputs: 0–3 (0–2 in case of adder); gates: 0 – NAND/NOR, 1 – OR/XOR)

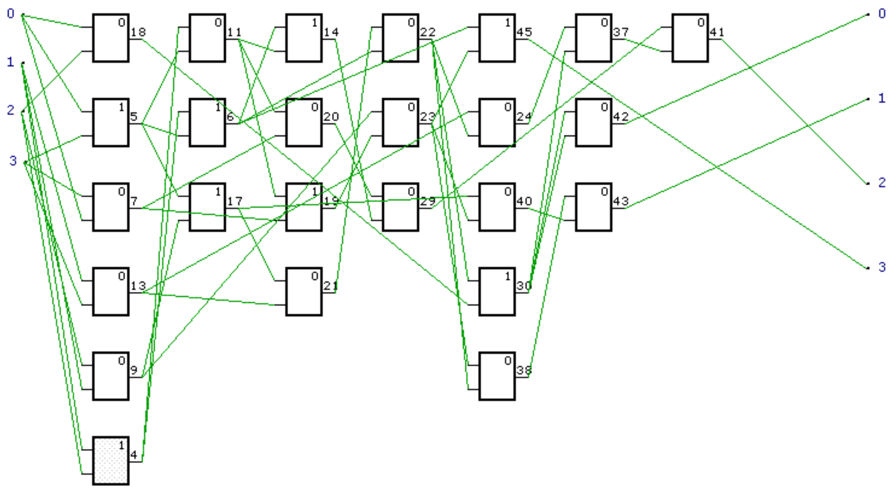


Fig. 7.3. Polymorphic multiplier/sorting network circuit (the inputs: A(0–1), B(2–3) in case of multiplier; the outputs: 0–3; gates: 0 – NAND/NOR, 1 – AND/AND)

4. 4: NAND/NOR and AND/AND
5. NAND/NOR and $(a \wedge \overline{b})/(a \wedge \overline{b})$
6. NAND/NOR and OR/XOR
7. NAND/NOR and XOR/AND
8. $(a \vee \overline{b})$/XOR and XOR/$(a \wedge \overline{b})$
9. $(a \vee \overline{b})$/OR and XOR/$(a \wedge \overline{b})$
10. $(a \vee \overline{b})$/XOR and NXOR/$(a \wedge \overline{b})$

and obtained the perfect solution only for combinations (4) and (8). An open theoretical issue remains which combinations of polymorphic gates are sufficient to implement the required multifunctional behavior.

In most cases we are not able to understand the topology of the evolved circuits since implementations of required behaviors are "entangled". Fig 7.4 shows the analysis performed on the first circuit listed in Table 7.2. In the mode 1 (Fig. 7.4A), the realization is based on a classical implementation of parity circuits. In mode 2 (Fig. 7.4B) an inefficient implementation of the 5-input median circuit was evolved. For comparison, the implementation of the same behavior depicted in Fig. 7.1 is much more compact and difficult to decode. We evolved some other polymorphic circuits with up to five inputs and outputs (see [103]). Those experiments have shown that more compact circuits can be obtained in case that a circuit has more outputs.

The CGP is not scalable in its basic form, which means that neither our approach can be scaled. In order to illustrate the computational effort of the proposed method, we measured the average time of evolution for 500 runs of the multiplier/sorting network problem. In average 118,985 generations were produced in a single run, which corresponds to 143 seconds of the computational time at Pentium IV (2.6 GHz, 512MB RAM).

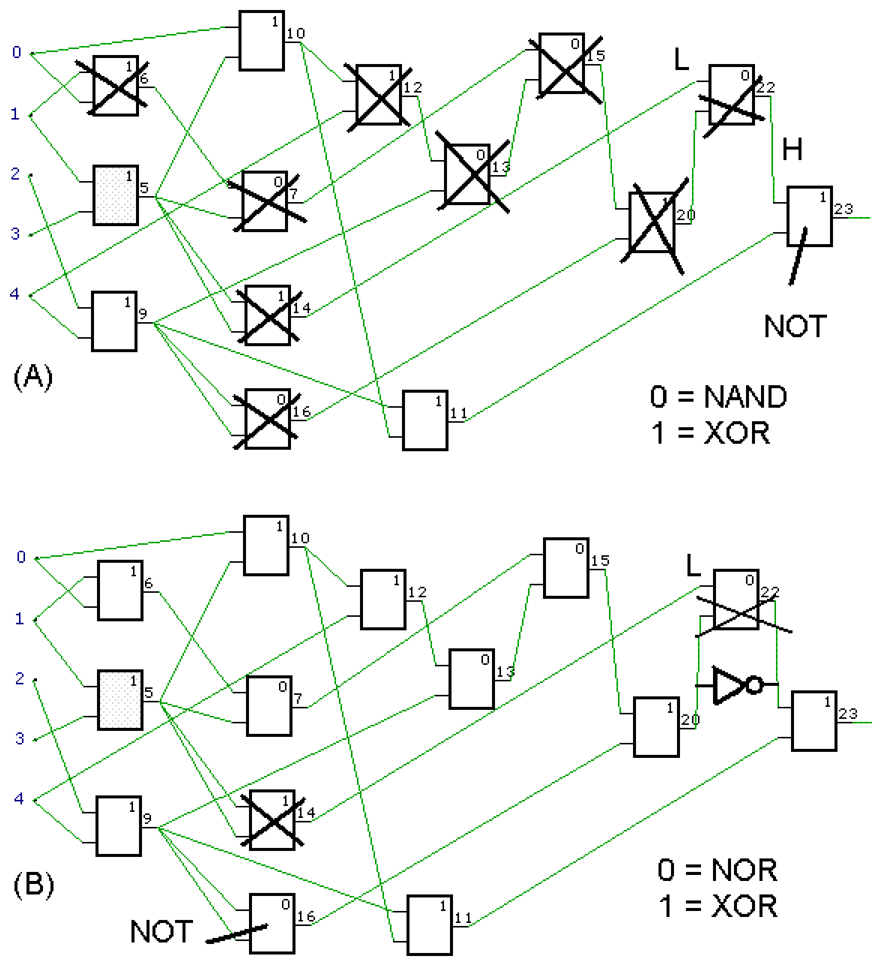


Fig. 7.4. Analysis of the polymorphic 5b-median/parity circuit (gates: 0 – NAND/NOR, 1 – XOR/XOR)

7.4 Development for Polymorphic Circuits

During *development*, a multicellular organism is formed from the zygote in a process of cellular division and cellular differentiation. Pattern formation depends on *positional information*, which instructs competent cells to differentiate in ways characteristic of their position in the structure. Positional information is not only provided by the environmental trigger but also is usually understood to be imparted by the concentration of one or more morphogens emitted from spatially distinct organizers [2]. The main feature of all the electronic developmental systems utilizing some information from the environment (e.g. positional information) is that this information is considered as one of many *digital* inputs to the system, typically obtained through a couple of wires [61].

The following experiments will show that there are other options for providing information for growing digital circuits. The positional information will not be provided in the digital form; rather it will influence the system in an analogue (and perhaps non-electrical) form, for example, as temperature, light, radiation, specific voltage etc. The motivation for this approach is that every electronic circuit operates in a real physical environment and "good" physical environment is crucial for correct behavior of the resulting system (similarly, cells survive and divide only in a "good" environment). The proposed approach should allow the developing system to interact with the environment much closely and to differentiate according to specific real-world situations. For example, one can imagine that a growing circuit could generate some heat, which, in turn, could automatically be reflected by the circuit to regulate its developmental process and so its behavior.

Of course, we do not want to build special sensors. The growing circuits should inherently interact with the physical environment, in our case, through polymorphic gates. If some gates of a growing circuit are polymorphic then the process of development can be controlled by the mentioned external signals (in addition to controls derived from expressed genetic information).

In particular, the use of polymorphic gates enables us to implement more complex developmental schemes. Therefore, the growing circuit and environment can interact in more complex ways which, in result, should lead to more complex target circuits. There are two scenarios:

- Polymorphic gates influence the physical structure of the circuit. For instance, for some environments (e.g., for low temperatures), a functional unit, say U, is created, for other environments (e.g. for high temperatures) functional unit U is not created. The unit U is created because a special mode of polymorphic gate is activated in which a corresponding control digital signal is enabled. Otherwise, the control signal is not enabled and unit U is not created.
- Polymorphic gates do not influence the physical structure of the growing circuit, i.e. circuit topology is independent of the environment. However, because the circuit contains polymorphic gates, its behavior depends on the environment. For instance, the growing circuit operates as an adder in low temperatures and as a sorter in high temperatures.

7.4.1 The Method

We will deal with the second scenario. In particular, the approach is based on the method introduced in Chapter 4.4. Starting with an embryo (a small instance of the desired solution), genetic algorithm is to find such a constructor which is able to create larger instance of the circuit from a smaller one. We will use the gates given in Table 7.3). Figure 7.5 shows the basic building block and its symbolic notation.

We are using an "empty" embryo at the beginning of the developmental process, i.e. no particular information is known about the structure of the initial circuit, which is, therefore,

ID	Gate	ID	Gate	ID	Gate	ID	Gate	ID	Gate
0	AND/I	4	I/AND	8	OR/XOR	12	XOR/OR	16	NOT/NOT
1	AND/AND	5	OR/I	9	I/OR	13	XOR/XOR	17	I/NOT
2	AND/OR	6	OR/AND	10	XOR/I	14	I/XOR	18	I/I
3	AND/XOR	7	OR/OR	11	XOR/AND	15	NOT/I		

Table 7.3. List of gates. Some of them are polymorphic. "I" denotes the identity function (buffer).

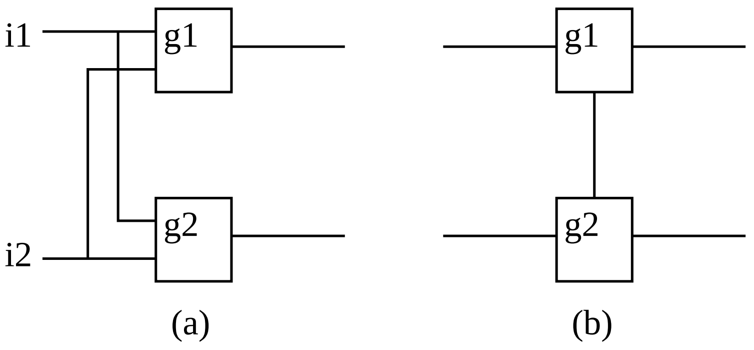


Fig. 7.5. The basic building block: (a) logic structure, (b) symbolic notation. $g1$ and $g2$ represent logic functions according to Table 7.3. $i1$ and $i2$ denote the input indices. This is encoded as $(i1, i2, g1, g2)$

considered as a 4-tuple $(0, 0, I/I, I/I)$. Considering that, there must be instructions in the instruction set that are able to set the input signals of the polymorphic gates and their functions (e.g. instruction MODIS and MODFS). These so-called *modify*-instructions only modify the gates denoted by the embryo pointer (*ep*) without copying them and hence the next-position pointer (*np*) remains unchanged after their execution. The instructions which are responsible for growth of the circuit are of the types *copy* or *copy-and-modify* (e.g. instructions CPOS or CPMIS). Every instruction is used in two variants (e.g. CPMIS and CPMIN), whose difference lies in updating strategy of the embryo pointer (*ep*) after execution of the instruction. Table 7.4 lists all the utilized instructions. For example, CPOS copies a pair of gates (i.e. one building block) from position given by *ep* to the position given by *np*; the embryo pointer remains unchanged in this case.

7.4.2 Experimental Setup

A simple genetic algorithm is utilized to find a constructor whose repeated application on an existing circuit will create a larger circuit. All the experiments were performed with the following settings: standard crossover with the probability 0.55, the probability of mutation 0.04, population size 40, tournament selection mechanism with the base 3, the maximal number of generations 20,000.

The size of the chromosome remains constant during an experiment. Note, that all the evolved constructors work only with either even or odd number of inputs.

The objective is to develop arbitrarily large circuits; however, only four developmental steps are considered in the fitness calculation in order to make the time of evolution reasonable. In the first environment, the fitness value is determined as $\Phi = f(2) + f(4) + f(6) + f(8)$ for even-input circuits and $\Phi = f(3) + f(5) + f(7) + f(9)$ for odd-input circuits, where $f(i)$ is the number of test sequences correctly processed by the circuit with i inputs. Therefore, the maximum fitness value that we can reach in one environment is $\Phi_{max} = 2^2 + 2^4 + 2^6 + 2^8 = 340$ for even-input circuits and $\Phi_{max} = 2^3 + 2^5 + 2^7 + 2^9 = 680$ for odd-input circuits. For the

Id	Instr.	A1	A2	Meaning
0	CPOS	–	–	copy the pair of gates from ep to np; $cp = cp + 1, np = np + 1$
1	CPON	–	–	copy the pair of gates from ep to np; $cp = cp + 1, np = np + 1, ep = ep + 1$
2	CPNS	p	–	copy $w - p$ pairs of gates; $cp = cp + 1, np = np + w - p$
3	CPNN	p	–	copy $w - p$ pairs of gates; $cp = cp + 1, np = np + w - p, ep = ep + w - p$
4	CPMIS	p	q	copy the pair of gates from ep to np and do $i_1 = (i_1 + p)\ mod\ w, i_2 = (i_2 + q)\ mod\ w, cp = cp + 1, np = np + 1$
5	CPMIN	p	q	copy the pair of gates from ep to np and do $i_1 = (i_1 + p)\ mod\ w, i_2 = (i_2 + q)\ mod\ w, cp = cp + 1, np = np + 1, ep = ep + 1$
6	CPMFS	p	q	copy the gates from ep to np and do $f_1 = p, f_2 = q, cp = cp + 1, np = np + 1$
7	CPMFN	p	q	copy the gates from ep to np and do $f_1 = p, f_2 = q, cp = cp + 1, np = np + 1, ep = ep + 1$
8	MODIS	p	q	modify inputs of the gates at ep as follows: $i_1 = (i_1 + p)\ mod\ w, i_2 = (i_2 + q)\ mod\ w, cp = cp + 1$
9	MODIN	p	q	modify inputs of the gates at ep as follows: $i_1 = (i_1 + p)\ mod\ w, i_2 = (i_2 + q)\ mod\ w, cp = cp + 1, ep = ep + 1$
10	MODFS	p	q	modify functions of the gates at ep as follows: $f_1 = p, f_2 = q; cp = cp + 1$
11	MODFN	p	q	modify functions of the gates at ep as follows: $f_1 = p, f_2 = q; cp = cp + 1, ep = ep + 1$
12	NOP	–	–	an empty instruction: $cp = cp + 1$

Table 7.4. The instruction set. p and q represent the arguments of the instruction; i_1 and i_2 denote the indices of inputs of the polymorphic gates; f_1 and f_2 are functions of the gates and w is the number of inputs of the circuit being created.

second environment the fitness value is calculated in the same way. At the end of evolution we have to verify whether the resulting constructors are general, i.e. able to produce arbitrarily large circuits (typically we verify the functionality of circuits up to 28 inputs).

7.4.3 Results

Some constructors were successfully evolved for the polymorphic even/odd parity circuits. According to the environment, the circuits calculate either even or odd parity. The solution is usually based on the conventional XOR gate (no. 13 in Table 7.3) and a polymorphic NOT switch (no. 17 in Table 7.3). As Fig. 7.6 shows, its structure is not surprising; however, the structure was designed fully automatically without any supporting domain knowledge (we started with the empty embryo $(0, 0, I/I, I/I)$). We gained 35 general constructors out of 200 independent runs of the evolutionary design process for a five-instruction chromosome. 42 constructors consisting of six instructions were evolved out of 200 independent runs of the evolutionary process from which 36 were recognized as general.

Figure 7.7 shows a polymorphic sorting network created by means of an eight-instruction constructor. The constructor is general. We gained 45 constructors out of 200 independent runs of the evolutionary process (five of them are general).

The evolved circuits use gate 2 (AND/OR) and 6 (OR/AND) which means that they sort the input sequences in increasing order in the first environment and in decreasing order in the second environment.

7.4.4 Discussion

In case of sorting networks, evolution has discovered that by exchanging AND gates with OR gates, the ordering of sorted sequence can be changed. There is no innovation; human

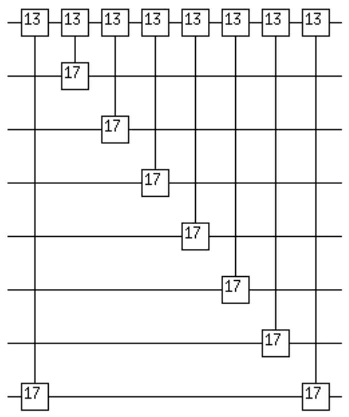


Fig. 7.6. Polymorphic even/odd parity circuit created by means of constructor [MODIS 4 3] [MODFS 13 17] [CPMIS 0 1] [MODIS 0 2] [CPOS 3 1]

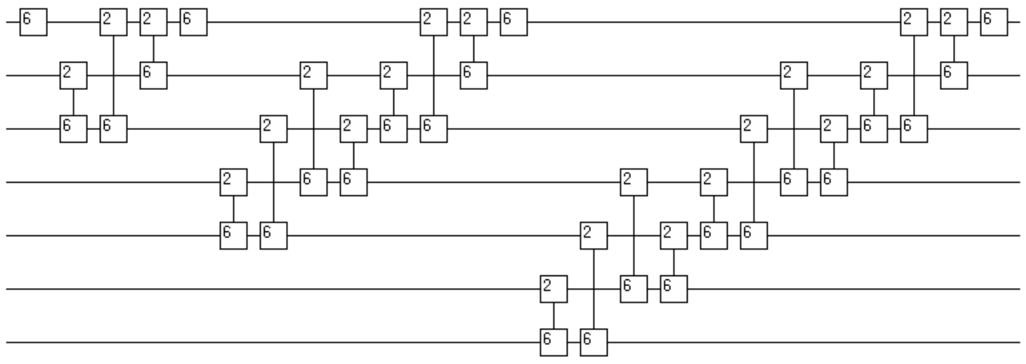


Fig. 7.7. Polymorphic sorting network created by means of the constructor [CPMIS 2 2] [MODFS 2 6] [CPMIS 1 2] [CPMIS 3 2] [CPMIS 0 1] [CPMIS 1 1] [CPNN 2 4] [CPNN 4 4]

designer would construct the circuit in the same way. Although the evolution could use many types of gates, it has utilized the same gates as a human designer uses. The implementation of AND as well as OR gate costs 6 transistors in the standard CMOS technology. Surprisingly, the cost of polymorphic AND/OR gate controlled by temperature is also 6 transistors [117]. If one were able to build OR/AND gate with the same cost, the resulting polymorphic sorting network should consist of the same number of transistors as the original one whose behavior cannot be changed!

Despite we put effort into evolution of other types of large polymorphic circuits (such as adder/sorting network and parity/Boolean symmetry circuits etc.) we have not obtained any functional result yet. The explanation could be as follows: The number of correct suitable topologies which perform the required behavior is very limited with the proposed encoding. Therefore, the probability is very low that a single topology can represent two different behaviors (e.g., n-bit adder and n-bit multiplier) in two different environments.

Of course, because of the utilized representation, it is always possible to manually merge two different circuits into a single working polymorphic circuit. The method is as follows: Construct the resulting circuit from left to right. If the first circuit requires logic function L_1, create polymorphic gate L_1/I. If the second circuit requires logic function L_2, create polymorphic gate I/L_2. Then the resulting circuit (consisting of polymorphic gates) will perform the first function in the first environment and the second function in the second environment. However, we are not interested in this type of solution, since there is no innovation visible.

7.5 Summary

We do not know how to design polymorphic circuits by means of conventional techniques. Hence we have introduced a CGP-based method and developmental scheme to create polymorphic circuits.

Although the CGP-based approach is not scalable, we obtained promising results; the evolved circuits exhibit compact topologies unknown in conventional circuit design. The results of experiments have indicated that the approach could be useful for the design of real-world applications of polymorphic electronics. However, we have utilized hypothetic polymorphic gates. Hence more research is needed in the basic polymorphic gate design. The evolved polymorphic circuits confirm that hypothesis H5 is valid.

Using the approach based on development we have discovered only two scalable polymorphic circuits—the parity detector and sorting network. In real biological systems as well as in some artificial developmental systems (e.g. in [61]) the interplay between a growing solution and its environment is very complex. In our system the interplay practically does not exist. It is necessary to develop more complex models of development and combine them with polymorphic gates to create complex and simultaneously innovative polymorphic circuits.

8. Characterization of Evolved Computational Systems

In this chapter, we will characterize the class of computational systems that are designated and implemented by means of evolutionary techniques. First, we will perform analysis of the experimental results obtained in previous chapters. Then we will investigate properties of the evolved systems from the point of view of theoretical computer science.

8.1 Analysis of the Evolved Circuits

8.1.1 Complexity and Innovation in the Evolved Circuits

We mentioned that the problem of scale is critical for evolvable hardware. Researchers have tried to evolve as complex circuits as possible. Simultaneously, they are interested in discovery of innovative solutions.

For purposes of this thesis, we will measure the *complexity* of an evolved circuit as the number of gates utilized in the evolved circuit. This is not a perfect measure because we have not optimized the size of the evolved circuits for some applications (for example, for sorting networks evolved in FPGA). Furthermore, the gates were not used as building blocks in some experiments (e.g. for the evolution at the transistor and functional levels). Therefore, in order to express the complexity of circuits evolved at these levels we have to perform some conversions. Nevertheless, this measure is accurate enough to distinguish between complexities of different classes of circuits that were evolved using different approaches.

Domain knowledge is the second parameter considered to characterize the quality of our results. In general, fitness function, genetic operators and problem representation influence the quality of search [77]. In all these three factors some domain knowledge is included. In most experiments described in this thesis, the fitness function calculates the difference between the required and obtained output signals for all or some selected input combinations (except, for instance, benchmark circuits for which a structural analysis has been performed). For this reason, we do not consider the fitness function as a critical factor for our comparisons. In most experiments, we have used a very simple variant of evolutionary algorithm, utilizing a simple mutation and one-point crossover. In many cases we recognized that this approach produces the best results (e.g., in CGP); in other cases we have not explored more sophisticated operators yet. Nevertheless, we have always aimed to maximize the performance and efficiency of the evolutionary algorithm. For these reasons, similarly to the fitness function, we do not consider the genetic operators utilized as critical for our comparisons. Most domain knowledge was included into the problem representation in which we define the elementary blocks the circuits are built from and the possibilities for interconnecting these blocks. Hence we consider the size of chromosome as a measure of knowledge included into the design process. In particular, we measure the number of bits. Note that the size of chromosome is not usually directly related to the amount of domain knowledge included. A short chromosome means that we assume a lot about the target system (i.e. we know how it should "look like"). A long chromosome means that we do not assume a lot about the target system and more freedom is given to the evolution to build the system. Again, the proposed measure is not perfect; however, it is sufficient for our purposes. Note that in all experiments we have started from a randomly generated population, i.e. no specific knowledge has been utilized at the beginning of evolution.

Figure 8.1 shows the relation between the size of chromosome and the complexity of circuits we presented in the previous chapters. In particular, the following classes are considered[1]:

- The **CGP** class: The best obtained implementations of 4×3-bit multiplier, 7-input sorting networks and 7-input median networks (see Table 4.2) belong to this class. We can also include into this class the 4×4-bit multiplier reported in [131] (the circuit consists of 57 gates). This is probably the largest circuit evolved at the gate level using CGP (i.e. with a direct mapping between the genotype and phenotype). Larger multipliers were evolved using more advanced techniques such as incremental evolution.
- **CGP-median** class consists of median circuits that have from 3 to 11 inputs (see Table 4.3). In contrast to the previous class, median circuits have only one output. Hence it is easier to evolve circuits with more inputs.
- Median circuits evolved using compare&swap encoding (**C&S-median**): This class consists of median circuits that have from 3 to 25 inputs (see Table 4.4). A single C&S component is counted as two gates.
- Polymorphic circuits evolved using CGP (**polymorphic**): In this class, four-input and five-input polymorphic modules are considered. As it is necessary to evaluate two modes of operation in the fitness function, the obtained circuits are of lower complexity in comparison to the CGP class.
- Image filters evolved in FPGA (**image filters**): These image filters were evolved on VRCs of different size. Their implementation cost was determined as the cost of the filter which is extracted from the FPGA and synthesized as an independent circuit. The average number of gates is around 2000 [94].
- Benchmark circuits (**benchmarks**): This class consists of benchmark circuits according to Table 4.11.
- Sorting networks constructed using development (**development**): Using this approach, arbitrarily large sorting networks can be constructed by means of the chromosome of constant size.
- Sorting networks evolved in FPGA (**sort_nets in FPGA**): Sorting networks of 6–28 inputs were considered in Fig. 8.1. Although programmable elements of VRC often operate as pure wires (i.e. they do not influence the complexity of the circuit), we are counting every programmable element utilized as the equivalent to two gates.
- Circuits evolved in the FPTA (**FPTA**): For purposes of this comparison, we consider that the two-input gate evolved in the FPTA is equivalent to a single gate, the RS circuit is equivalent to two gates and the D-latch is equivalent to 5 gates.

The curves in 8.1 can easily be extrapolated toward circuits of lower complexity (and shorter chromosomes). However, we believe that only a very small extrapolation towards more complex circuits is possible when one utilizes the proposed methods. The following observations have come from analyzing Figure 8.1.

[1] An abbreviations is used to label every class.

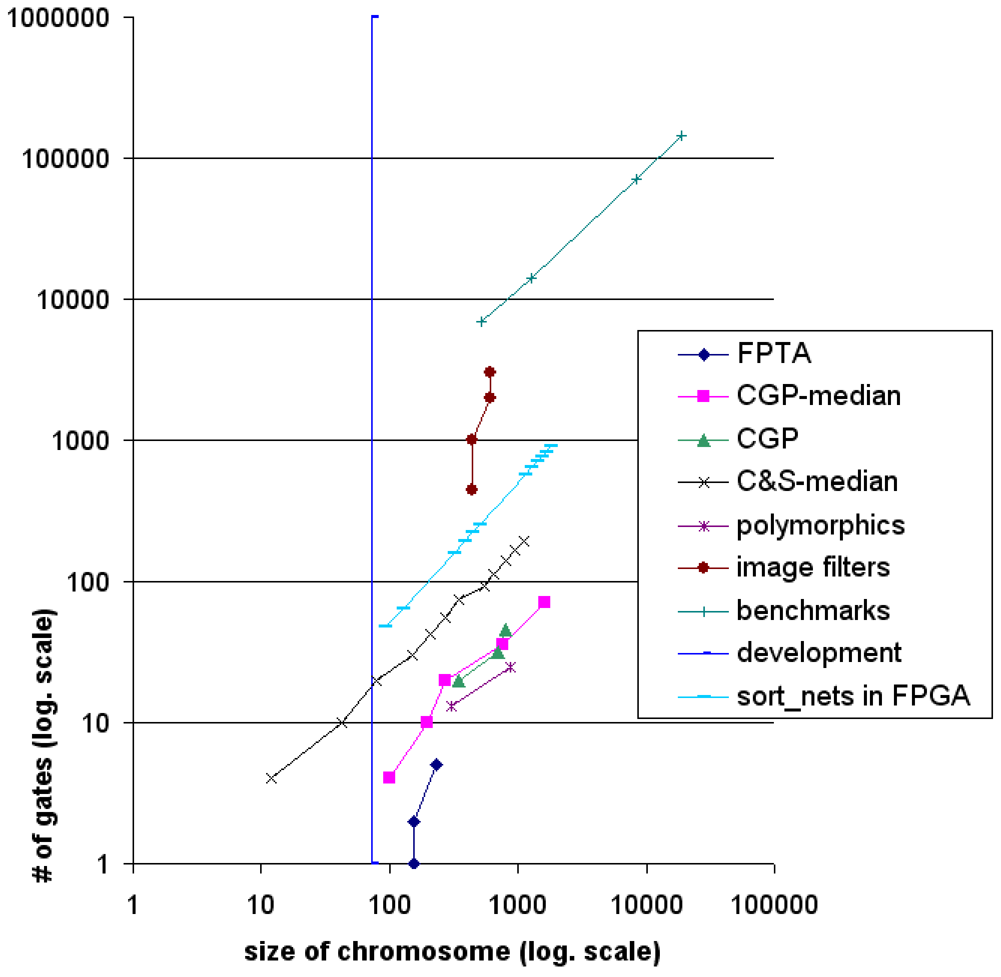


Fig. 8.1. The complexity of the evolved circuits (# of gates) vs the size of chromosome (# of bits). The curves may easily be extended toward left-down corner.

- The proposed techniques have a limitation in the size of the search space they are able to effectively explore. The limitation is somewhere around 1000 bits in the chromosome. The main reason for this limit is that the corresponding search spaces are extremely rugged and so difficult to search.
- The *gate-level* evolution can produce circuits of complexity up to approx. 100 gates. This class is characterized by the fact that all possible input/output combinations are considered in the fitness calculation process. As the complexity of the circuits is low and simultaneously the chromosome is long enough, the chromosome may encode many candidate variants. It means that novel solutions can be discovered easily.
- By using an *application-specific encoding* (as in case of C&S-medians or sorting networks in FPGA), more complex circuits can be obtained in comparison to the simple CGP. Similarly to CGP, all possible input/output combinations are evaluated in the fitness function. Hence a very time-consuming fitness evaluation represents the main obstacle in growing complexity.
- The *functional-level* approaches have produced complex and innovative solutions. The complexity of circuits is in fact unlimited and depends on a complexity of components used as

building blocks. It seems that only this approach is able to generate solutions to real-world problems. However, because the complexity of evolved circuits is relative high, it is impossible to test all possible input/output combinations and, therefore, only a subset of these combinations can be evaluated in the fitness function (alternatively, some structural properties of candidate circuits can be evaluated). That immediately implies that (1) usually only a suboptimal solution is obtained and (2) the application class is restricted. The level of novelty obtained strongly depends on the functional blocks utilized, i.e. on the amount of domain knowledge introduced.
- *Developmental approaches* are able to produce arbitrarily complex solutions using relatively short chromosomes. EA is in fact used to derive a program that generates solutions to all instances of a given problem. The problem is that the obtained solutions exhibit regularity which causes redundancy.
- As the *transistor-level* evolution produces only small digital circuits, it leaves a space opened for unconventional implementations of these elementary digital circuits. Note that the FPTA used in our experiments is not a platform primarily designated to implement digital circuits.

We can summarize: The complexity of evolved circuits only partially depends on the size of chromosome. It mainly depends on the size of objects encoded in the chromosome or manipulated by instructions encoded in the chromosome (in case of development). That is the reason why we can evolve relatively complex circuits although we can effectively search only in the search space of size approx. 2^{1000}. Almost at all levels, we have obtained results that could be classified as human-competitive. It confirms hypothesis H1 that there is no significant relation between the size of chromosome, complexity of evolved circuits and level of innovation obtained. A lesson learned is that it is extremely important how the information stored in chromosomes is interpreted.

8.1.2 Computational Effort

Let us consider those circuits that have similar size of the configuration bitstream. In other words, we are interested in circuits designed by methods that utilized the same amount of information to find a solution. Table 8.1 summarizes properties of circuits described by chromosome of the size between 512–868 bits. The columns have the following meaning: *in* is the number of inputs, *out* is the number of outputs, *chrom-size* is the size of chromosome, *gates* is the number of gates the circuit is composed of, *test-vect* is the number of test vectors used in the fitness function to evaluate the circuit[2], *pop-size* is the population size and *gnrs* is the average number of generations to evolve the circuit.

Figure 8.2 shows some of these values in a graphical form. The circuits are sorted according their complexity. In particular, we can observe the computational effort required to evolve a given circuit. Let us assume that the evaluation of a circuit requires a unit time for a single training vector. We can see that the evaluation of a 21-input median circuit is the most time-consuming. It is because the circuit has many inputs and all possible combinations are considered. Recall that for image filters and benchmark circuits not all possible input combinations are considered. The *pxg* is the product of population size and the number of generations (pxg = popsize × gnrs). It informs us how many candidate circuits must be

[2] Benchmark circuits are not evaluated using test vectors; instead, the structural analysis of the circuit is performed. As the analysis has a quadratic complexity, we can estimate the number of steps required to evaluate a benchmark circuits as K^2 where K denotes the number of components in the circuit.

Table 8.1. Properties of circuits whose chromosomes have similar length

circuit	in	out	chrom-size	gates	test-vect	pop-size	gnrs
polymorphic mult./adder	4	4	868	20	2×2^4	128	120k
9-input median (CGP)	9	1	755	36	2^9	128	11k
4×3-bit multiplier	7	7	700	46	2^7	5	10M
21-input median (C&S)	21	1	800	140	2^{21}	200	270
sorting network in FPGA	16	16	512	256	2^{16}	4	20k
image filter	72	8	616	2000	254^2	4	30k
benchmark circuit	40	40	525	7003	50^2	25	200

evaluated in order to find the required solution; in other words how it is *difficult* for the evolutionary algorithm to find a solution. It can be seen that this parameter is maximal for CGP although the complexity of these gate-level circuits is not very high. We have to mention that in CGP we have optimized the size of the evolved circuits, i.e. a lot of computational time was spent to improve an already discovered solution. Surprisingly, *pxg* decreases with the complexity of evolved circuits. If we multiply *pxg* by the number of test vectors, we can obtain the number of test vectors that must be evaluated in the design process. This value corresponds to the *"time of evolution"*; however, only in case that all the experiments are performed on the same machine. In reality, we have utilized different machines (different PCs and FPGAs). We can observe that the "time of evolution" is very similar for almost all circuits. This parameter indicates how much time/resources we are able/willing to invest to the evolutionary design process.

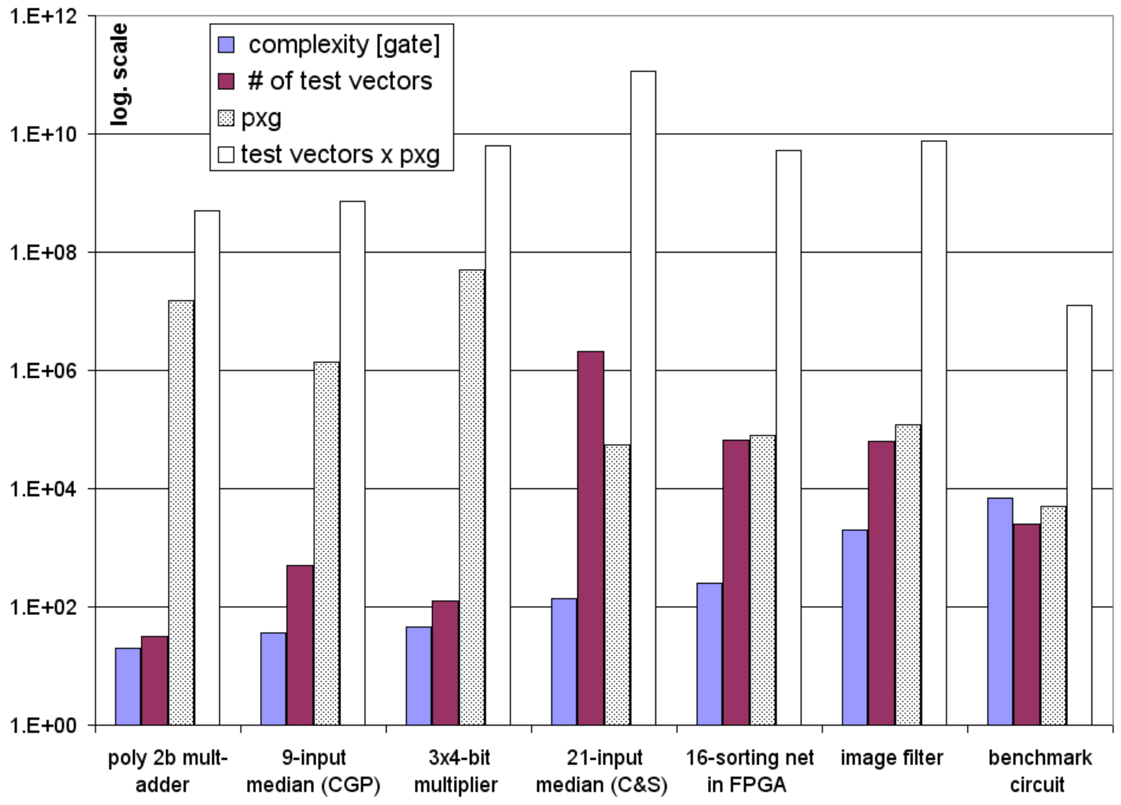


Fig. 8.2. Comparison of circuits whose chromosomes have similar length

8.2 Consequence for Theoretical Computer Science

8.2.1 Basic Characteristics

The traditional process of implementation of a computing device which is supposed to work according to a given specification can be understood as a process of manipulating/fabricating a suitable physical system whose behavior can easily be interpreted in the required way. In this implementation process, the specification is transformed onto implementation in well-defined steps using well-defined methods, tools and components. The engineer usually knows the abstract model of the target system at any desired level. The engineer is primarily guided by knowledge of methodology established for solving the given type of problem. For that process, it is crucial that the engineer understands the principles that the resulting system is based on. The design process is completely under engineer's control. There exists a well-established consensus (e.g. voltage levels for logic '0' and '1') allowing us to interpret the input/output behavior of the system. Furthermore, it is usually required to have a sufficient interpretation of internal physical processes of that system in terms of computation ("two pn-junctions form a transistor and two transistors make up an inverter").

With the development of natural computing, we have started to understand various physical, chemical and biological processes as computational processes. For example, a cell is able to construct three-dimensional shapes of proteins according to one-dimensional genetic information in an extremely efficient way [19]. Interpreting these observations as computation could help us in building more powerful, adaptive and fault-tolerant computers.

Similarly to the conventional design process, we have to declare our required interpretation of input/output behavior in advance when the evolutionary design approach is utilized to find a physical implementation of a computational process. This interpretation is used to define the fitness function. When not constrained, the evolution can utilize all the resources available, including normally unused characteristics of the reconfigurable platform and environment. Similarly to biological evolution, artificial evolution promotes those candidates ("organisms") that are better adapted to the given environment. Although the evolution is often able to find an implementation perfectly satisfying the specification, we have problems to understand how and why the solution works. In other words, we are not able to interpret the internal behavior as a computational process which produces the required responses.

On the basis of experiments presented in this thesis and reported in literature and on the basis of bibliographic search performed in the area of computer science, we can assemble Table 8.2 that compares ordinary computers, evolved computational systems and the brain (the brain represents here all the physical and biological systems that are often studied as computational devices). This table shows that the evolved computational devices represent a distinctive class of devices that exhibits specific combination of properties, not studied in the scope of all computational devices up to now. The main properties of this class of devices are as follows:

1. We can specify behavior of these devices beforehand.
2. We can usually obtain their implementations.
3. In general, we are not able to understand why their internal implementation causes the computation.

Properties (1) and (2) mean that these devices are useful for practice. Property (3) is sometimes problematic from a practical point of view.

Table 8.2. Comparison of different types of computational devices

Property	Ordinary computer	The brain	Evolved device
I/O behavior can be interpreted as computing	Yes	Yes	Yes
Device is a computing mechanism	Yes	Unsure	Unsure
An abstract model exists before implementation	Yes	No	No
The required behavior is specified beforehand	Yes	No	Yes
Engineers can design&build	Yes	No	Yes

8.2.2 The Embodiment Phenomenon

Figures 4.1 and 6.1 have shown that the evolutionary approach is able to find a solution outside the space of conventional designs. Every physical object that we can design by means of the evolutionary algorithm is of finite size. Assume that we have a reconfigurable digital circuit consisting of K programmable elements whose function is defined by the configuration bitstream stored in the B-bit configuration memory. There are 2^B different configurations, i.e. up to 2^B behaviors that can be realized in the reconfigurable device. However, the number of *distinguishable* behaviors is usually much smaller. Although every two different configurations determine two physically different electronic circuits, the two configurations can produce the same (digital) behavior because of inherent redundancy of reconfigurable circuits.

It was shown that under some conditions, some configurations could exhibit *useful* digital behavior that *none* of 2^B configurations can do under normal conditions (see Thompson's experiment [122]). Note that the fitness function has remained formally unchanged. The evolutionary analog circuit design or evolutionary robotics show that it is impossible to build a simulator of a given physical entity in order to completely avoid doing experiments with physical devices. In many cases, simulators predict different values than physical devices generate simply because computer models cannot cover everything about the reality. Physical embodiment is a crucial feature of these systems (see also [13]).

We can explain this "embodiment phenomenon" by means of mappings g and f shown in Fig. 8.3. Mapping g represents the transformation from the genotype space $\mathcal{C}$ to phenotypes $\mathcal{M}$ (i.e. from configuration bitstreams to physical circuits, i.e. computational machines). Mapping f is the fitness function assigning fitness values (e.g., real numbers, R) to phenotypes. We are offering two explanations of the "embodiment phenomenon":

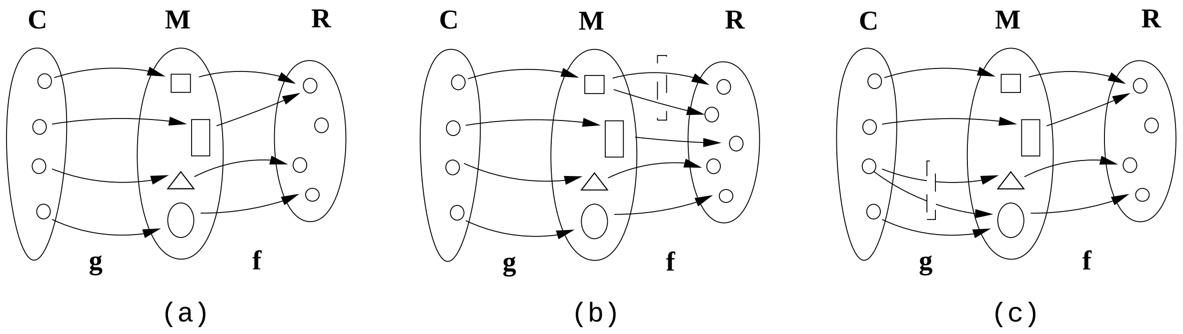


Fig. 8.3. Properties of g and f when implemented physically: (a) a physical implementation corresponds to an abstract model, (b) the fitness function is influenced by environment, (c) the genotype-phenotype mapping is influenced by environment

In the first viewpoint, f is important. In fact, f assigns two different fitness values to a single circuit (machine) $m \in \mathcal{M}$ (see Fig. 8.3b). Note again that the fitness function has remained formally unchanged. In reality, m is represented by an electronic circuit (configuration) whose behavior depends on something missing in the fitness calculation. In this case, f can not be seen as a function, rather f is a *relation*.

In the second viewpoint, g is important. Fitness function has never assigned different fitness values to a single machine (assuming invariable fitness function). All phenotypes are seen as different (but they can obtain the same fitness value). However, a single genotype can generate two or more different phenotypes (circuits), because it is assumed that environment influences the genotype-phenotype mapping. The situation is known from nature where phenotypes depend on the environment where they are constructed (developed). For instance, the development depends on gradient concentrations nearby a dividing cell [2].

Low temperature experiments (see Chapter 6.4) have demonstrated the "embodiment phenomenon". Although we have left the fitness function unchanged, the behavior as well as a corresponding fitness value of circuit was changed with temperature. The reason is simple: the effect of temperature was not considered by the fitness function.

The idea of polymorphic electronics should be understood as a surprising and novel outcome of research in the area of intrinsic evolutionary design. In fact, the polymorphic gates utilize the "embodiment phenomenon".

Evolution is able to effectively exploit physical properties of a given *materio* in order to build a device, as demonstrated by Thompson [122], Harding [39], Stoica [114] and others. Probably the best example is the brain evolved by nature. However, there are physical limits (summarized in Chapter 2.6) that define what can ultimately be done by an arbitrary physical system when it is considered as a computational device, independently of the approach utilized to build that system. As the conventional implementation approach is constrained by practical limits that are determined by contemporary technological development, there are, in principle, no limits of this kind in case of the evolutionary approach. The main problem of the current (artificial) evolutionary design approach is that it is not able to produce complex and simultaneously innovative designs.

8.2.3 The Implementation Problem and Evolutionary Design

Various versions of the Church-Turing thesis are often used to analyze computational capabilities of physical and biological systems (such as the brain). The Church Turing thesis does entail that if the brain follows an effective procedure, then that procedure is Turing–computable. The Modest Physical Church-Turing thesis does entail that if the brain performs computations then those computations are Turing-computable. But neither the Church-Turing thesis nor Modest Physical Church–Turing thesis do say whether the brain follows effective procedure or performs computations [84]. In the brains we can observe elementary cells and higher-level units composed of many cells; however, we are not able to exactly determine which properties are responsible for the behavior that we can interpret as computation. Also in case of artificially evolved computational devices we can observe elementary components but, in general, we do not know which properties of these components the evolution has used to ensure the behavior which can be interpreted as computation. As the brains and computational devices are very similar in this point neither the Church-Turing thesis nor Modest Physical Church-Turing thesis does say whether the evolved computational devices follow effective procedure, i.e. whether they perform computations.

A given computing can be implemented in different ways in different physical systems (this property is called *multiple realizability*). Searle pointed out that “the multiple realizability of computationally equivalent processes in different physical media is not just a sign that the processes are abstract, but that they are not intrinsic to the system at all. They depend on an interpretation from outside ... computation is not discovered in the physics, it is assigned to the physics” [89]. We asked whether the evolved physical entities that perform some useful computations are computers. The previous argument clearly shows that the question does not have a clear sense. The computation is not discovered in the physics; however, we (humans) are able to interpret this behavior as computation.

Assume that a simple sequential circuit has been evolved in the FPTA or liquid crystals. Moreover, assume that the environment is stable (i.e. temperature, radiation and all the phenomena that could influence its behavior are stable). There are digital inputs and outputs. In principle, we are able to interpret the signals at these ports as “zeroes” and “ones” because we had to declare our interpretation in the fitness evaluation process (i.e. in advance, before the circuit was evolved) how we are going to interpret these signals. However, we don’t exactly understand how the platform performs the computation. We have specified only the required input/output relation in the fitness function, i.e. we have not introduced any abstract model indicating that some internal states must exist in order to implement that behavior (in general, the behavior is not combinational). It is the role of evolution to introduce and implement the internal states in some way. Therefore, we are not able to say anything about the relation between the abstract model (because it does not exist) and the resulting implementation created using the evolutionary algorithm.

Most scientists cited in Chapter 2.4 would probably agree that the evolved systems really perform computation because we can interpret their input/output behavior as computation. However, there are problems with the interpretation of their internal behavior. Copeland, Johnson and others would like to see that symbols within the system have a consistent interpretation throughout the computation and that interpretation is specified beforehand. It makes no sense to establish the mapping between computational states and physical states before the evolution is performed. The evolution can use totally different physical properties of the physical device to implement the required behavior. On the other hand, establishing the mapping at the end of evolution could solve the problem because (under some assumptions, such as that the environment does not influence the physical device at all) it could be possible to identify physical states that correspond to abstract computational states. Here, it is reasonable to follow Scheutz approach (“starting in physical” [88]). Unfortunately, the discovery of computational states within physical system is not quite sure. Although we can apply all the physical theories we currently know to analyze and explain the evolved system, we could obtain no satisfactory answer. As Bartles et al. have shown [6], evolution is able to utilize the physical behaviors that are physically impossible from the point of view of current physical theories. Finally, we have to consider again that no abstract model is available because our specification is defined in terms of input/output relations. Hence we are not able to establish the relation between physical and abstract states.

If we agree that (1) it does not matter how a system realizes a function as long as the system is digital from the outside (a digital system can be treated as a “black box” with digital inputs and outputs), (2) we are able to interpret its input/output behavior as computation and (3) this interpretation is defined in advance then *the evolved systems are computational systems* (computing mechanisms). On the other hand, if the correspondence between the abstract states and physical states is crucial to qualify a system as computational one then *the evolved systems are not computing mechanism.*

8.2.4 The Computational Power

In contrast to the previous subchapter, in which we have investigated the *result* of evolutionary process, the following paragraphs deal with evolvable computational machines. As the goal of this thesis is to characterize the class of computational devices designed and implemented using the evolutionary approach, we have to ask what is their *computational power.* It was proven that evolvable computational machines exhibit a super-Turing computational power [94]. We are not going to repeat here the formal model and argumentation used in [94] to prove this claim. This subchapter is devoted to recall basic ideas.

In case of evolvable computational machines in which the evolutionary algorithm is responsible for adaptation, we try in fact to find an algorithm (machine) in a given set of algorithms (e.g. in the set of all configurations of an FPGA) that satisfies the objectives (formulated via the fitness function) by means of an evolutionary algorithm in which the problem encoding, genotype–phenotype mapping and genetic operators remain unchanged during the run. We assume infinite and interactive computation. As an example, we can mention an evolvable image filter for a space mission. The filter should dynamically change its functionality and thus to suppress variable noise and ensure functional recovery in case of malfunctions.

We can observe that evolvable computational machines operating in a dynamic environment show simultaneous non-uniformity of computation, interaction with an environment, and infinity of operations. Furthermore, the point at time in which the fitness function (specification) is changed is in general *uncomputable* (see the *Driving Home from Work* problem presented in Chapter 2.3 that is uncomputable on a standard Turing machine, but computable in reality).

At each time point, these evolvable devices have a *finite* description. However, when one observes their computation in time, they represent *infinite* sequences of reactive devices computing non-uniformly. The "evolution" of machine's behavior is supposed to be endless. In fact, it means that they offer an example of real devices (physical implementations) that can perform computation that no single Turing machine (without oracle) can. If the lifespan of the physical implementations of these machines (e.g. evolvable hardware) can be considered as infinite, then these computers cannot be simulated by means of a standard Turing machine. In [94] we have shown how to simulate an evolvable machine on an interactive Turing machine with advice.

8.3 Summary

In this chapter we experimentally confirmed hypothesis H1: By showing human-competitive results at various levels of description, we demonstrated that there is no significant relation between the complexity of the evolved circuits, chromosome size and novelty obtained.

Because the evolutionary design does not work with a specification based on abstract computational states and because we are not able (in advance) to establish a mapping between computational states and physical states, the evolved systems cannot be considered as computing mechanisms in terms of theoretical computer science (disproving so hypothesis H6).

9. Conclusions

Evolutionary circuit design and evolvable hardware traditionally belong to the area of electrical engineering. In this habilitation thesis, we have interpreted the evolutionary design of computational systems from the perspective of computer science. In this view, the evolutionary circuit design represents an approach allowing engineers to implement computational devices. The evolved computational devices represent a distinctive class of devices that exhibits specific combination of properties, not visible and studied in the scope of all computational devices up to now. In particular, the evolution is able to produce a computing system satisfying the given specification; however, in general, we do not understand how and why the system performs a computation. It means that we cannot classify the evolved systems as computing mechanisms (disproving thus hypothesis H6). Consequently, the evolved devices are irrelevant for the Church-Turing thesis. On the basis of experimental results, we have shown that:

- There is no significant relation between the complexity of the evolved circuits, chromosome size and novelty obtained. Human-competitive results can be evolved at any level of interest if the problem is formulated in a clever way (H1).
- It is possible to speed up digital circuit evolution using a common FPGA for a reasonable class of applications (H2).
- It is possible to evolve sequential circuits at reconfigurable platforms operating at sub-gate levels. In particular, the RS circuit was (re)discovered at the transistor level (H3).
- Evolution is able to form computational elements in extreme low temperatures (H4).
- The embodiment can be utilized to introduce additional functions to digital modules in a manageable way; in particular, polymorphic gate-level modules were discovered (H5).

This habilitation thesis have opened several directions for future research, including:

- A link between the evolved computational systems and cognitive sciences have been established. A possible research should investigate the evolved computational systems and evolvable computational machines in terms of *computationalism* (where the brain is considered as computer).
- In order to improve the effectiveness of the evolutionary circuit design, novel representations and genetic operators should be proposed and evaluated.
- In order to evolve more complex circuits, new nature-inspired developmental schemes should be proposed and evaluated. In particular, the growing circuits should more closely interact with the environment.
- The evolutionary algorithms should be applied to exploit properties of materio and environment to discover new approaches to the implementation of engineering artefacts (especially in the area of nanotechnology).
- Real-world applications of evolvable hardware implemented as ASICs are surprisingly simple from the point of view of evolutionary algorithms. The evolutionary algorithm is usually

utilized to set up a few hundreds of configuration bits (see, for example, [53]). Only a very small part of the product is autonomously adapted (tuned) for a given user and environment. We could call this approach as the *design for evolvability.* We should identify real-world applications suitable for this approach and systematically re-design them (so replacing fixed components by reconfigurable components) in order to adapt them for a given customer and in-situ.
- More research is needed in the area of polymorphic electronics: both theoretical and practical.

References

1. Adleman, L. M.: Molecular computation of solutions to combinatorial problems. Science **266**(11), 1021–1024 (1994)
2. Alberts, B. et al.: *Essential Cell Biology – An Introduction to the Molecular Biology of the Cell* (Garland Publishing, New York 1998)
3. Ali, B., Almaini, A., Kalganova, T.: Evolutionary algorithms and their use in the design of sequential logic circuits. Genetic Programming and Evolvable Machines. **5**(1) 11–29 (2004)
4. Bäck, T.: *Evolutionary Algorithms in Theory and Practice* (Oxford University Press, New York Oxford 1996)
5. Banzhaf, W., Nordin, P., Keller, R. E., Francone, F. D.: *Genetic Programming – An Introduction* (Morgan Kaufmann Publishers, San Francisco CA 1998)
6. Bartels, R. et al.: Learning from Learning Algorithms: Applications to attosecond dynamics of high-harmonic generation. Physical Review A **70**(1):1-5 (2004)
7. Bentley, P. (ed): *Evolutionary Design by Computers* (Morgan Kaufmann Publishers, San Francisco CA 1999)
8. Bidlo, M., Sekanina, L.: Providing Information from the Environment for Growing Electronic Circuits Through Polymorphic Gates. In: GECCO-SEEDs'05: Proc. of Genetic and Evolutionary Computation Conference - Workshops 2005, Washington D.C. (ACM 2005) pp 242–248
9. Bidlo, M., Bidlo, R.: An Evolved General Construction Method for the Sorting Networks. In: MEMICS'05: *Proc. of the 1st Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, Znojmo (MU Brno 2005)
10. Blodget, B., Roxby, J., Keller E., McMillan, S., Sundararajan, P.: A Self-reconfiguring Platform. In: FPL'03: *Proc. of the 13th International Conference on Field-Programmable Logic and Applications*, Lisbon, Portugal, 2003. Lecture Notes in Computer Science, vol 2778 (Springer, Berlin Heidelberg New York 2003) pp 565–574
11. Bourianoff, G.: The Future of Nanocomputing. IEEE Computer. **36**(8), 44–53 (2003)
12. Britanica Encyclopedia Online (October 2005)
13. Brooks, R.: The relationship between matter and life. Nature. **409**(6818), 409-411 (2001)
14. Choi, S. S., Moon, B. R.: More Effective Genetic Search for the Sorting Network Problem. In: GECCO'02: *Proc. of the Genetic and Evolutionary Computation Conference* GECCO'02 (Morgan Kaufmann 2002) pp 335–342
15. Compton, K., Hauck, S.: Reconfigurable computing: A survey of systems and software. ACM Comput. Surv. **34**(2), 171–210 (2002)
16. Copeland B. J.: What is Computation? Synthese. **108**, 335–359 (1996)
17. Copeland, B. J.: Super-Turing machines. Complexity. **4**(1), 30–32 (1998)
18. Copeland, B. J., Sylvan, R.: Beyond the universal Turing machine. Australasian J. of Philosophy. **77**(1), 46–66 (1999)
19. Crescenzi, P. et al.: On the Complexity of Protein Folding. Journal of Computational Biology. **5**(3) 423–466 (1998)
20. Dawkins, R.: *The Blind Watchmaker* (Penguin Books, London 1991)
21. Deutsch, D.: Quantum theory, the Church-Turing principle and the universal quantum computer. Proceedings of the Royal Society London A, A400:97–117 (1985)
22. Devillard, N.: Fast Median Search: An ANSI C Implementation. 1998 `http://ndevilla.free.fr/median/median/index.html`
23. Durbeck, L., Macias, L.: The Cell Matrix: An architecture for nanocomputing. Nanotechnology. **12**(3), 217–230 (2001)
24. Eberbach, E., Goldin, D., Wegner, P.: Turing's Ideas and Models of Computation. In: Alang Turing: Life and Legacy of a Great Thinker, ed. Teuscher Ch.(Springer, Berlin Heidelberg New York 2004) pp 166–173
25. Ehrenfeucht, A., Harju, T., Petre, I., Prescott, D.M., Rozenberg, G.: Computation in Living Cells – Gene Assembly in Ciliates. Natural Computing Series (Springer, Berlin Heidelberg New York 2004)
26. Elowitz, M. B., Leibler, S.: A Synthetic Oscillatory Network of Transcriptional Regulators. Nature (London) **403**(6767), 335–338 (2000)

27. Extreme Temperature Electronics – Tutorial (2005) http://www.extremetemperatureelectronics.com
28. Flake, G. W.: *The Computational Beauty of Nature* (The MIT Press 1999)
29. Flockton, S. J., Sheehan, K.: Intrinsic Circuit Evolution Using Programmable Analogue Arrays. In: ICES'98: *Proc. of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, ed by Sipper, M. et al., Lausanne, Switzerland, 1998. Lecture Notes in Computer Science, vol 1478 (Springer, Berlin Heidelberg New York 1998) pp 144–153
30. Frank, M. P.: Reversibility for Efficient Computing. PhD thesis, Massachusetts Institute of Technology (1999) pp 430
31. de Garis, H.: Evolvable Hardware – Genetic Programming of a Darwin Machine. In: ICANNGA'93: *International Conference on Artificial Neural Networks and Genetic Algorithms*, Innsbruck, Austria (Springer, Berlin Heidelberg New York 1993) http://www.cs.usu.edu/~degaris/papers/index.html
32. Garvie, M., Thompson, A.: Evolution of Combinational and Sequential On-line Self-Diagnosing Hardware. In: ICES'03: *Proc. of the 5th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Tyrrell, A. et al., Trondheim, Norway, 2003. Lecture Notes in Computer Science, vol 2606 (Springer, Berlin Heidelberg New York 2003) pp 167-173
33. Goldberg, D.: *Genetic Algorithms in Search, Optimization, and Machine Learning* (Addision-Wesley 1989)
34. Gordon, T., Bentley, P.: On Evolvable Hardware. Soft Computing in Industrial Electronics, ed by Ovaska, S., Sztandera, L. (Physica-Verlag, Heidelberg 2001) pp 279–323
35. Gordon, T., Bentley, P: Towards Development in Evolvable Hardware. In: EH'02: *Proc. of the 4th NASA/DoD Conference on Evolvable Hardware*, ed by Stoica, A. et al., Alexandria, Virginia, USA, 2002 (IEEE Computer Society, Los Alamitos 2002) pp 241–250
36. Gruska, J.: *Foundations of Computing* (Int. Thomson Publishing Computer Press 1997)
37. Gruau, F.: Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm. PhD thesis, l'Universite Claude Bernard Lyon I, 1994, pp 159
38. Haddow, P., Tufte, G., van Remortel, P.: Shrinking the Genotype: L-systems for EHW? In: ICES'01: *Proc. of the 4th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Liu, Y. et al., Tokyo, Japan, 2001 Lecture Notes in Computer Science, vol 2210 (Springer, Berlin Heidelberg New York 2001) pp 128–139
39. Harding, S., Miller, J.: Evolution in materio: Initial experiments with liquid crystal. In: EH'04: *Proc. of 2004 NASA/DoD Conference on Evolvable Hardware*, ed by Zebulum, R. et al., Seattle, USA, 2004 (IEEE Computer Society, Los Alamitos 2004) pp 298–305
40. Harding, S., Miller, J.: Evolution in materio: Evolving Logic Gates in Liquid Crystal. In: Proc. of the Workshop on Unconventional Computing at ECAL 2005 VIIIth European Conference on Artificial Life. To appear in International Journal of Unconventional Computing, pp 12
41. Hennessy, J. L., Patterson, D. A.: *Computer Architecture – A Quantitative Approach* (Morgan Kaufman Publishers, San Francisco 1996)
42. Higuchi, T. et al.: Evolving Hardware with Genetic Learning: A First Step Towards Building a Darwin Machine. In: SAB'92: *Proc. of the 2nd International Conference on Simulated Adaptive Behaviour* (MIT Press, Cambridge MA 1993) pp 417–424
43. Higuchi, T. et al.: Real-world applications of analog and digital evolvable hardware. IEEE Trans. on Evolutionary Computation. **3**(3), 220–235 (1999)
44. Hillis, W. D.: Co-evolving parasites improve simulated evolution as an optimization procedure. Physica D 42, 228–234 (1990)
45. deHon, A.: Comparing Computing Machines. In: *Configurable Computing: Technology and Applications*, Bellingham, WA, Proc. SPIE 3526 (1998) pp 124–133
46. Hopcroft, J. E., Ullman, J. D.: *Introduction to Automata Theory, Languages, and Computation* (Addison–Wesley, Reading MA 1979)
47. Hornby, G. S., Pollack, J. B.: The Advantages of Generative Grammatical Encodings for Physical Design. In: CEC'01: *Proc. of the 2001 Congress on Evolutionary Computation* CEC2001 (IEEE Computer Society Press 2001) pp 600–607
48. Imamura, K., Foster, J. A., Krings, A. W.: The Test Vector Problem and Limitations to Evolving Digital Circuits. In: EH'00: *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, ed by Lohn, J. et al., Palo Alto, CA, USA, 2000 (IEEE Computer Society, Los Alamitos 2000) pp 75–79
49. Israel, D.: Reflections on Gödel's and Gandy's Reflections on Turing's Thesis. Minds and Machines. **12**(2), 181–201 (2002)
50. International Technology Roadmap for Semiconductors 2004 update http://public.itrs.net
51. Johnson, C. G.: What Kinds of Natural Processes Can be Regarded as Computations? In: Computation in Cells and Tissues: Perspectives and Tools of Thought. Paton T. (ed) (Springer, Berlin Heidelberg New York 2004)
52. Juillé, H.: Evolution of Non–Deterministic Incremental Algorithms as a New Approach for Search in State Spaces. In: *Proc. of the 6th International Conference on Genetic Algorithms*, ed by Eshelman, L. J., San Francisco, 1995 (Morgan Kaufmann Publishers, San Francisco 1995) pp 351–358

53. Kasai, Y. et al.: Adaptive Waveform Control in a Data Transceiver for Multi-speed IEEE1394 and USB Communication. In: ICES'05: *Proc. of the 6th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Moreno M. et al., Barcelona, Spain, 2005. Lecture Notes in Computer Science, vol 3637 (Springer, Berlin Heidelberg New York 2005) pp 198–204
54. Keymeulen, D., Stoica, A., Zebulum, R.: Fault-Tolerant Evolvable Hardware using Field Programmable Transistor Arrays. IEEE Transactions on Reliability. Special Issue on Fault-Tolerant VLSI Systems. **49**(3) 305–316 (2000)
55. Knuth, D. E.: *The Art of Computer Programming*, 2nd edn (Addison–Wesley, 1998)
56. Kolte, P., Smith, R., Su, W.: A Fast Median Filter Using AltiVec. In Proc. of the IEEE Conf. on Computer Design, Austin, Texas, IEEE CS Press, 1999, pp 384–391
57. Kořenek, J., Sekanina, L.: Intrinsic Evolution of Sorting Networks: A Novel Complete Hardware Implementation for FPGAs In: ICES'05: *Proc. of the 6th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Moreno M. et al., Barcelona, Spain, 2005. Lecture Notes in Computer Science, vol 3637 (Springer, Berlin Heidelberg New York 2005) pp 46–55
58. Koza, J. R.: *Genetic Programming II: Automatic Discovery of Reusable Programs* (MIT Press, Cambridge MA 1994)
59. Koza, J. R. et al.: *Genetic Programming III: Darwinian Invention and Problem Solving* (Morgan Kaufmann Publishers, San Francisco CA 1999)
60. Koza, J. R. et al.: *Genetic Programming IV: Routine Human-Competitive Machine Intelligence* (Springer New York 2003)
61. Kumar, S.: Investigating Computational Models of Development for the Construction of Shape and Form. PhD thesis, University of London, UK, 2004
62. Kurdahi, F. et al. (eds): Configurable computing. IEEE Des. Test Comput. **17**(1), 17–19 (2000)
63. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development **5**, 183–191 (1961)
64. Langeheine, J., Meier, K., Schemmel, J.: Intrinsic Evolution of Quasi DC Solutions for Transistor Level Analog Electronic Circuits Using a CMOS FPTA Chip. In: EH'02: *Proc. of the 4th NASA/DoD Conference on Evolvable Hardware*, ed by Stoica, A. et al., Alexandria, Virginia, USA, 2002 (IEEE Computer Society, Los Alamitos 2002) pp 75–84
65. Layzell, P.: A New Research Tool for Intrinsic Hardware Evolution. In: ICES'98: *Proc. of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, ed by Sipper, M. et al., Lausanne, Switzerland, 1998. Lecture Notes in Computer Science, vol 1478 (Springer, Berlin Heidelberg New York 1998) pp 47–56
66. van Leeuwen, J., Wiedermann, J.: The Turing Machine Paradigm in Contemporary Computing. In: *Mathematics Unlimited – 2001 and Beyond* (Springer, Berlin Heidelberg New York 2001) pp 1139–1155
67. Liberouter web site (2005) `www.liberouter.org`
68. Linden, D. S.: Optimizing Signal Strength In-Situ Using an Evolvable Antenna System. In: EH'02: *Proc. of the 4th NASA/DoD Conference on Evolvable Hardware*, ed by Stoica, A. et al., Alexandria, Virginia, USA, 2002 (IEEE Computer Society, Los Alamitos 2002) pp 147–151
69. Lloyd, S.: Ultimate Physical Limits to Computation. Nature **406** 1047–1054 (2000)
70. Lohn, L., Larchev, G., DeMara, R.: A Genetic Representation for Evolutionary Fault Recovery in Virtex FPGAs. In: ICES'03: *Proc. of the 5th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Tyrrell, A. et al., Trondheim, Norway, 2003. Lecture Notes in Computer Science, vol 2606 (Springer, Berlin Heidelberg New York 2003) pp 47–56
71. Lohn J. D. et al.: Evolutionary Design of an X-Band Antenna for NASA's Space Technology 5 Mission. In: *Proc. 2004 IEEE Antenna and Propagation Society International Symposium and USNC/URSI National Radio Science Meeting*, Vol. 3 (IEEE Press 2004) pp 2313–2316
72. Maclennan, B. J.: Transcending Turing Computability. Minds and Machines. **13**(1), 3–22 (2003)
73. Mange, D. et al.: Towards robust integrated circuits: The embryonics approach. Proc. of IEEE. **88**(4), 516–541 (2000)
74. Manovit, C., Aporntewan, C., Chongstitvatana, P.: Synthesis of Synchronous Sequential Logic Circuits from Parital Input/Output Sequences. In: ICES'98: *Proc. of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, ed by Sipper, M. et al., Lausanne, Switzerland, 1998. Lecture Notes in Computer Science, vol 1478 (Springer, Berlin Heidelberg New York 1998) pp 98-105
75. Martínek, T., Sekanina, L.: An Evolvable Image Filter: Experimental Evaluation of a Complete Hardware Implementation in FPGA. In: ICES'05: *Proc. of the 6th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Moreno M. et al., Barcelona, Spain, 2005. Lecture Notes in Computer Science, vol 3637 (Springer, Berlin Heidelberg New York 2005) pp 76–85
76. Meindl, J.: Low Power Microelectronics: Retrospect and Prospect. Proceedings of the IEEE. **83**(4), 619–635 (1995)
77. Michalewicz, Z., Fogel, D. B.: *How to Solve It – Modern Heuristics* (Springer, Berlin Heidelberg New York 2000)

78. Miller, J., Job, D., Vassilev, V.: Principles in the evolutionary design of digital circuits – Part I. Genetic Programming and Evolvable Machines. **1**(1), 8–35 (2000)
79. Miller, J., Thomson, P.: Cartesian Genetic Programming. In: EuroGP'00: *Proc. of the 3rd European Conference on Genetic Programming*, Lecture Notes in Computer Science, vol 1802 (Springer, Berlin Heidelberg New York 2000) pp 121–132
80. Miller, J., Downing, K.: Evolution in Materio: Looking beyond the Silicon Box. In: EH'02: *Proc. of the 4th NASA/DoD Conference on Evolvable Hardware*, ed by Stoica, A. et al., Alexandria, Virginia, USA, 2002 (IEEE Computer Society, Los Alamitos 2002) pp 167–176
81. Miller, J., Thomson, P.: A Developmental Method for Growing Graphs and Circuits. In: ICES'03: *Proc. of the 5th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Tyrrell, A. et al., Trondheim, Norway, 2003. Lecture Notes in Computer Science, vol 2606 (Springer, Berlin Heidelberg New York 2003) pp 93–104
82. Murakawa, M. et al.: Evolvable Hardware at Function Level. In: PPSN IV: *Proc. of the Parallel Problem Solving from Nature Conference*,Lecture Notes in Computer Science, vol 1141 (Springer, Berlin Heidelberg New York 1996) pp 62–71
83. Pečenka, T., Kotásek, Z., Sekanina, L., Strnadel, J.: Automatic Discovery of RTL Benchmark Circuits with Predefined Testability Properties. In: EH'05: *Proc. of the 2005 NASA/DoD Workshop on Evolvable Hardware*, ed by Lohn, j. et al., Washington DC, USA, 2005 (IEEE Computer Society, Los Alamitos 2005) pp 51–58
84. Piccinini, G.: Computations and Computers in the Sciences of Mind and Brain. PhD thesis, University of Pittsburgh (2003) pp 323
85. PicoChip home page `http://www.picochip.com`
86. Pradhan, D. K.: *Fault-Tolerant Computer System Design* (Prentice Hall 1996)
87. Preble, S., Lipson, H., Lipson, M.: Two-dimensional photonic crystals designed by evolutionary algorithms. Applied Physics Letters. **86** 061111 (2005)
88. Scheutz, M.: When Physical Systems Realize Functions ... Minds and Machines. **9**(2), 161–196 (1999)
89. Searle, J. Is the Brain a Digital Computer? Proceedings and Addresses of the American Philosophical Association **64**, 21–37 (1990)
90. Sekanina, L.: Image Filter Design with Evolvable Hardware. In: EvoIASP'02: *Applications of Evolutionary Computing*, ed by Cagnoni, S. et al., 4th Workshop on Evolutionary Computation in Image Analysis and Signal Processing, Kinsale, Ireland, 2002. Lecture Notes in Computer Science, vol 2279 (Springer, Berlin Heidelberg New York 1997) pp 255–266
91. Sekanina, L.: Virtual Reconfigurable Circuits for Real-World Applications of Evolvable Hardware. In: ICES'03: *Proc. of the 5th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Tyrrell, A. et al., Trondheim, Norway, 2003. Lecture Notes in Computer Science, vol 2606 (Springer, Berlin Heidelberg New York 2003) pp 186–197
92. Sekanina, L.: Towards Evolvable IP Cores for FPGAs. In: EH'04: *Proc. of the 2004 NASA/DoD Workshop on Evolvable Hardware*, ed by Zebulum, R. et al., Seattle USA, 2004 (IEEE Computer Society, Los Alamitos 2004) pp 145–154
93. Sekanina, L., Růžička, R.: Easily Testable Image Operators: The Class of Circuits Where Evolution Beats Engineers. In: EH'04: *Proc. of the 2004 NASA/DoD Workshop on Evolvable Hardware*, ed by Zebulum, R. et al., Seattle USA, 2004 (IEEE Computer Society, Los Alamitos 2004) pp 135–144
94. Sekanina, L.: *Evolvable Components: From Theory to Hardware Implementations.* Natural Computing Series (Springer, Berlin Heidelberg New York 2004)
95. Sekanina, L.: Evolutionary Design Space Exploration for Median Circuits. In: EvoHOT'04: *Applications of Evolutionary Computing*, Coimbra, Portugal, 2004, Lecture Notes in Computer Science, vol 3005 (Springer, Berlin Heidelberg New York 2004) pp 240–249
96. Sekanina, L., Friedl, S.: An Evolvable Combinational Unit for FPGAs. Computing and Informatics. **23**(5–6) 461–486 (2004)
97. Sekanina, L.: Evolvable computing by means of evolvable components. Natural Computing. **3**(3) 323–355 (2004)
98. Sekanina, L.: Evolving Constructors for Infinitely Growing Sorting Networks and Medians. In: Sofsem'04: *SOFSEM: Theory and Practice of Computer Science*, Měřín, Czech Rep., 2004, Lecture Notes in Computer Science, vol 2932 (Springer, Berlin Heidelberg New York 2004) pp 314–323
99. Sekanina, L., Bidlo, M.: Evolutionary Design of Arbitrarily Large Sorting Networks Using Development. Genetic Programming and Evolvable Machines. **6**(3) 29 pp (2005)
100. Sekanina, L., Zebulum, R. S.: Intrinsic Evolution of Controllable Oscillators in FPTA-2. In: ICES'05: *Proc. of the 6th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Moreno M. et al., Barcelona, Spain, 2005. Lecture Notes in Computer Science, vol 3637 (Springer, Berlin Heidelberg New York 2005) pp 98-107
101. Sekanina, L., Zebulum, R.: Evolutionary discovering of the concept of the discrete state at the transistor level. In: EH'05: *Proc. of the 2005 NASA/DoD Workshop on Evolvable Hardware*, ed by Lohn, J. et al., Washington DC, USA, 2005 (IEEE Computer Society, Los Alamitos 2005) pp 73–78

102. Sekanina, L.: Evolutionary Design of Gate-Level Polymorphic Digital Circuits. In: EvoHOT'05: *Applications of Evolutionary Computing*, Lausanne, Switzerland, 2005, Lecture Notes in Computer Science, vol 3449 (Springer, Berlin Heidelberg New York 2004) pp 185-194
103. Sekanina, L.: Design Methods for Polymorphic Digital Circuits. In: DDECS'05: *Proc. of the IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, Sopron, Hungary (UWH Sopron 2005) pp 145–150
104. Shackleford, B. et al.: A high-performance, pipelined, FPGA-based genetic algorithm machine. Genetic Programming and Evolvable Machines. **2**(1), 33–60 (2001)
105. Sipper, M. et al: A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. IEEE Trans. on Evolutionary Computation. **1**(1), 83–93 (1997)
106. Šonka, M., Hlaváč, V., Boyle, R.: *Image Processing: Analysis and Machine Vision.* (Thomson-Engineering 1999)
107. Stannett, M.: Computation and hypercomputation. Minds and Machines. **13**(1), 115–153 (2003)
108. Stepney, S. et al.: Journeys in Non-Classical Computation – A Grand Challenge for Computing Research. pp 30 (2004) `http://www.cs.york.ac.uk/nature/gc7/`
109. Stoica, A., Zebulum, R., Keymeulen, D.: Mixtrinsic Evolution. In: ICES'00: *Proc. of the 3rd International Conference on Evolvable Systems: From Biology to Hardware*, ed by Miller, J. et al., Edinburgh, Scotland, UK, 2000. Lecture Notes in Computer Science, vol 1801 (Springer, Berlin Heidelberg New York 2000) pp 208–217
110. Stoica, A. et al.: Evolution of Analog Circuits on Field Programmable Transistor Arrays. In: EH'00: *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, ed by Lohn, J. et al., Palo Alto, CA, USA, 2000 (IEEE Computer Society, Los Alamitos 2000) pp 99–108
111. Stoica, A. et al.: Evolving Circuits in Seconds: Experiments with a Stand-Alone Board Level Evolvable System. In: EH'02: *Proc. of the 4th NASA/DoD Conference on Evolvable Hardware*, ed by Stoica, A. et al., Alexandria, Virginia, USA, 2002 (IEEE Computer Society, Los Alamitos 2002) pp 67–74
112. Stoica, A., Zebulum, R., Keymeulen, D., Ferguson I., Duong, V., Guo, X.: Evolvable hardware techniques for on-chip automated reconfiguration of programmable devices. Soft Computing - Spec. Issue on Evolvable Hardware. **8**(5) 354–365 (2004)
113. Stoica, A., Keymeulen, D., Arslan, T., Duong, V., Zebulum, R., Ferguson I., Guo, X.: Circuit Self-Recovery Experiments in Extreme Environments. In: EH'04: *Proc. of the 2004 NASA/DoD Workshop on Evolvable Hardware*, ed by Zebulum, R. et al., Seattle USA, 2004 (IEEE Computer Society, Los Alamitos 2004) pp 142–145
114. Stoica, A., Zebulum, R. S., Guo X., Keymeulen, D., Ferguson, I., Duong, V.: Taking Evolutionary Circuit Design From Experimentation to Implementation: Some Useful Techniques and a Silicon Demonstration. IEE Proc.-Comp. Digit. Tech. **151**(4) 295–300 (2004)
115. Stoica, A.: Evolvable Hardware for Autonomous Systems. Tutorial CEC, GECCO (2004) `http://ehw.jpl.nasa.gov/Content/Public/TutorialCEC2004/TutorialCEC2004.pdf`
116. Stoica, A., Zebulum, R. S., Keymeulen, D., Lohn, J.: On Polymorphic Circuits and Their Design Using Evolutionary Algorithms. In: AI2002: Proc. of IASTED International Conference on Applied Informatics, Innsbruck, Austria (Acta Press 2002)
117. Stoica, A., Zebulum, R., Keymeulen, D.: Polymorphic Electronics. In: ICES'01: *Proc. of the 4th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Liu, Y. et al., Tokyo, Japan, 2001 Lecture Notes in Computer Science, vol 2210 (Springer, Berlin Heidelberg New York 2001) pp 291–302
118. Strnadel, J.: Testability Analysis and Improvements of Register-Transfer Level Digital Circuits. PhD thesis, Brno University of Technology, Faculty of Information Technology (2004) pp 150
119. Sulaiman, N., Arslan, T.: A Multi-objective Genetic Algorithm for On-chip Real-time Optimisation of Word Length and Power Consumption in a Pipelined FFT Processor targeting on MC-CDMA Receiver. In: EH'05: *Proc. of the 2005 NASA/DoD Workshop on Evolvable Hardware*, ed by Lohn, J. et al., Washington DC, USA, 2005 (IEEE Computer Society, Los Alamitos 2005) pp 154-159
120. Tempesti, G. et al.: Ontogenetic Development and Fault Tolerance in the POEtic Tissue. In: ICES'03: *Proc. of the 5th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Tyrrell, A. et al., Trondheim, Norway, 2003. Lecture Notes in Computer Science, vol 2606 (Springer, Berlin Heidelberg New York 2003) pp 141–152
121. Thompson, A.: *Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution.* Distinguished Dissertation Series (Springer, London 1998)
122. Thompson, A., Layzell, P., Zebulum, R. S.: Explorations in design space: unconventional electronics design through artificial evolution. IEEE Trans. on Evolutionary Computation. **3**(3), 167–196 (1999)
123. Thomson, R., Arslan, T.: Evolvable Hardware for the Generation of Sequential Filter Circuits. In: EH'02: *Proc. of the 4th NASA/DoD Conference on Evolvable Hardware*, ed by Stoica, A. et al., Alexandria, Virginia, USA, 2002 (IEEE Computer Society, Los Alamitos 2002) pp 17-25
124. Toffoli, T.: Non-conventional computers. Encyclopedia of Electrical and Electronics Engineering vol. 14 (Wiley & Sons 1998) pp 455–471

125. Toffoli, T.: Nothing Makes Sense in Computing Except in the Light of Evolution. J. of Unconventional Computing. **1**(1), 3–29 (2005)
126. Torresen, J.: Possibilities and Limitations of Applying Evolvable Hardware to Real-World Applications. In: FPL'00: *Proc. of the 10th International Conference on Field-Programmable Logic and Applications*, ed by Hartenstein, R., Grünbacher, H., Villach, Austria, 2000. Lecture Notes in Computer Science, vol 1896 (Springer, Berlin Heidelberg New York 2000) pp 203–239
127. Torresen, J.: A scalable approach to evolvable hardware. Genetic Programming and Evolvable Machines. **3**(3), 259–282 (2002)
128. Tour J. M.: Molecular Electronics. World Scientific (2003)
129. Tufte, G., Haddow, P.: Prototyping a GA Pipeline for Complete Hardware Evolution. In: EH'99: *Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, ed by Stoica, A. et al., Pasadena, CA, USA, 1999 (IEEE Computer Society, Los Alamitos 1999) pp 143–150
130. Turing, A. M.: On Computable Numbers, with an Application to the Entscheidungsproblem. In: Proceedings of the London Mathematical Society **42**(2) (1936-37) pp 230–265 (with corrections from Proceedings of the London Mathematical Society, Series 2, **43** (1937) pp 544–546)
131. Vassilev, V., Job, D., Miller, J.: Towards the Automatic Design of More Efficient Digital Circuits. In: EH'00: *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, ed by Lohn, J. et al., Palo Alto, CA, USA, 2000 (IEEE Computer Society, Los Alamitos 2000) pp 151–160
132. Vassilev, V., Miller, J.: Scalability Problems of Digital Circuit Evolution. In: EH'00: *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, ed by Lohn, J. et al., Palo Alto, CA, USA, 2000 (IEEE Computer Society, Los Alamitos 2000) pp 55–64
133. Vašíček, Z., Sekanina, L.: Evolutionary Design of Combinational Circuits. Elektrorevue. No. 43 (2004) `http://www.elektrorevue.cz/clanky/04043/`
134. Wakerly, J.: *Digital Design: Principles and Practices* (Prentice-Hall, London 2000)
135. Wall, M.: GAlib: A C++ Library of Genetic Algorithm Components, version 2.4. Massachusetts Institute of Technology, 1996, `http://lancet.mit.edu/ga/dist/galibdoc.pdf`
136. Wiedermann, J., van Leeuwen, J.: Relativistic Computers and Non-uniform Complexity Theory. In: UMC'02: *Proc. of 3rd Conf. on Unconventional Models of Computation*, ed by Calude C. S., Kobe, Japan, 2002. Lecture Notes in Computer Science, vol 2509 (Springer, Berlin Heidelberg New York 2002) pp 287–299
137. Wegner, P., Goldin, D.: Computation beyond Turing machines. Commun. ACM. **46**(4), 100–102 (2003)
138. Wolpert, D. H., Macready, W. G.: No Free Lunch Theorems for optimization. IEEE Trans. on Evolutionary Computation. **1**(1), 67–82 (1997)
139. Xilinx, Inc. WWW home page: `http://www.xilinx.com` (2005)
140. Yao, X., Higuchi, T.: Promises and Challenges of Evolvable Hardware. In: ICES'96: *Proc. of the 1st International Conference on Evolvable Systems: From Biology to Hardware*, ed by Higuchi, T. et al., Tsukuba, Japan, 1996. Lecture Notes in Computer Science, vol 1259 (Springer, Berlin Heidelberg New York 1997) pp 55–78
141. Zebulum, R., Pacheco, M., Vellasco, M.: *Evolutionary Electronics – Automatic Design of Electronic Circuits and Systems by Genetic Algorithms* (CRC Press, Boca Raton 2002)
142. Zebulum, R. S., Stoica, A., Keymeulen, D., Sekanina, L., Ramesham, R., Guo, X.: Evolvable Hardware System at Extreme Low Temperatures. In: ICES'05: *Proc. of the 6th International Conference on Evolvable Systems: From Biology to Hardware*, ed by Moreno M. et al., Barcelona, Spain, 2005. Lecture Notes in Computer Science, vol 3637 (Springer, Berlin Heidelberg New York 2005) pp 37-45
143. Zhang, Y., Smith, S., Tyrrell, A.: Intrinsic Evolvable Hardware in Digital Filter Design. *Applications of Evolutionary Computing* Lecture Notes in Computer Science, vol 3005 (Springer, Berlin Heidelberg New York 2005) pp 389–398
144. Zhirnov, V., Cavin, R., Hutchby, J. A., Bourianoff G. I.: Limits to Binary Logic Switch Scaling – A Gedanken Model. Proceedings of the IEEE. **91**(11), 1934–1939 (2003)
145. Zhu, J., Sutton, P.: FPGA Implementations of Neural Networks – A Survey of a Decade of Progress. In: Proc. of the 13th International Conference on Field-Programmable Logic and Applications. Lecture Notes in Computer Science, vol 2778 (Springer, Berlin Heidelberg New York 1997) pp 1062–1066

Appendix: Human-Competitive Results in Genetic and Evolutionary Computing

1. Spector, L. et al., 1998: Creation of a better-than-classical quantum algorithm for the Deutsch-Jozsa “early promise” problem (B, F)
2. Spector, L. et al., 1999: Creation of a better-than-classical quantum algorithm for Grover’s database search problem (B, F)
3. Spector, L. et al., 1999: Creation of a quantum algorithm for the depth-two AND/OR query problem that is better than any previously published result (D)
4. Barnum, H. et al., 2000: Creation of a quantum algorithm for the depth-one OR query problem that is better than any previously published result (D)
5. Spector, L. et al., 2003: Creation of a protocol for communicating information through a quantum gate that was previously thought not to permit such communication (D)
6. Spector, L. et al., 2003: Creation of a novel variant of quantum dense coding (D)
7. Luke, S., 1998: Creation of a soccer-playing program that won its first two games in the Robo Cup 1997 competition (H)
8. Andre, D. et al., 1999: Creation of a soccer-playing program that ranked in the middle of the field of 34 human-written programs in the Robo Cup 1998 competition (H)
9. Koza, J. et al., 1994: Creation of four different algorithms for the transmembrane segment identification problem for proteins (B, E) [58, 59]
10. Koza, J. et al., 1999: Creation of a sorting network for seven items using only 16 steps (A, D) [59]
11. Koza, J. et al., 2003: Rediscovery of the Campbell ladder topology for lowpass and highpass filters (A, F)
12. Koza, J. et al., 1999: Rediscovery of the Zobel “M-derived half section” and “constant K” filter sections (A, F) [59]
13. Koza, J. et al., 1999: Rediscovery of the Cauer (elliptic) topology for filters (A, F) [59]
14. Koza, J. et al., 1999: Automatic decomposition of the problem of synthesizing a crossover filter (A, F) [59]
15. Koza, J. et al., 1999: Rediscovery of a recognizable voltage gain stage and a Darlington emitter-follower section of an amplifier and other circuits (A, F) [59]
16. Koza, J. et al., 1999: Synthesis of 60 and 96 decibel amplifiers (A, F) [59]
17. Koza, J. et al., 1999: Synthesis of analog computational circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions (A, D, G) [59]
18. Koza, J. et al., 1999: Synthesis of a real-time analog circuit for time-optimal control of a robot (G) [59]
19. Koza, J. et al., 1999: Synthesis of an electronic thermometer (A, G) [59]
20. Koza, J. et al., 1999: Synthesis of a voltage reference circuit (A, G) [59]
21. Koza, J. et al., 1999: Creation of a cellular automata rule for the majority classification problem that is better than the Gacs-Kurdyumov-Levin (GKL) rule and all other known rules written by humans (D, E) [59]

22. Koza, J. et al., 1999: Creation of motifs that detect the D-E-A-D box family of proteins and the manganese superoxide dismutase family (C) [59]
23. Koza, J. et al., 2003: Synthesis of topology for a PID-D2 (proportional, integrative, derivative, and second derivative) controller (A, F) [60]
24. Koza, J. et al., 2003: Synthesis of an analog circuit equivalent to Philbrick circuit (A, F) [60]
25. Koza, J. et al., 2003: Synthesis of a NAND circuit (A, F) [60]
26. Koza, J. et al., 2003: Simultaneous synthesis of topology, sizing, placement, and routing of analog electrical circuits (A, F, G) [60]
27. Koza, J. et al., 2003: Synthesis of topology for a PID (proportional, integrative, and derivative) controller (A, F) [60]
28. Koza, J. et al., 2003: Rediscovery of negative feedback (A, E, F, G) [60]
29. Koza, J. et al., 2003: Synthesis of a low-voltage balun circuit (A) [60]
30. Koza, J. et al., 2003: Synthesis of a mixed analog-digital variable capacitor circuit (A) [60]
31. Koza, J. et al., 2003: Synthesis of a high-current load circuit (A) [60]
32. Koza, J. et al., 2003: Synthesis of a voltage-current conversion circuit (A) [60]
33. Koza, J. et al., 2003: Synthesis of a cubic function generator (A) [60]
34. Koza, J. et al., 2003: Synthesis of a tunable integrated active filter (A) [60]
35. Koza, J. et al., 2003: Creation of PID tuning rules that outperform the Ziegler-Nichols and Astrom-Hagglund tuning rules (A, B, D, E, F, G) [60]
36. Koza, J. et al., 2003: Creation of three non-PID controllers that outperform a PID controller using the Ziegler-Nichols or Astrom-Hagglund tuning rules (A, B, D, E, F, G) [60]
37. Lohn, J.: et al., 2004: An Evolved Antenna for Deployment on NASA's Space Technology 5 Mission (D, E, G) [71]
38. Spector, L., 2004: Automatic Quantum Computer Programming: A Genetic Programming Approach (B, D)
39. Fukunaga, A., 2004: Evolving Local Search Heuristics for SAT Using Genetic Programming (B?)
40. Lipson, H., 2004: How to Draw a Straight Line Using a GP: Benchmarking Evolutionary Design Against 19th Century Kinematic Synthesis Statement (A, B)
41. Khosraviani et al., 2004: Organization Design Optimization Using Genetic Programming (E, G, H)
42. Stoica, A. et al., 2004: Taking evolutionary circuit design from experimentation to implementation: some useful techniques and a silicon demonstration – polymorphic NAND/NOR gate (A, D) [114]
43. Preble, S. et al, 2005: Two-dimensional photonic crystals designed by evolutionary algorithms (B, E, D) [87]
44. Bartels, R. et al., 2005: Applications to attosecond dynamics of high-harmonic generation and shaped-pulse optimization of coherent soft-x-rays (D) [6]
45. Koza, J. et al., 2005: Automated Re-Invention of Six Patented Optical Lens Systems using Genetic Programming (A, F, G)
46. Massey, P. et al., 2005: Evolution of a Human-Competitive Quantum Fourier Transform Algorithm Using Genetic Programming (B, F, G)
47. Corno, F. et al., 2005: Evolving Assembly Programs: How Games Help Microprocessor Validation (C, H)

48. Terrile, R. et al., 2005: Evolutionary Computation Technologies for the Automatic Design of Space Systems (E, F, G)
49. Sipper, M. et al, 2005: Attaining Human-Competitive Game Playing with Genetic Programming (Backgammon, Robocode, Chess Endgame) (H)
50. Grasemann, U. et al., 2005: Effective Image Compression using Evolved Wavelets (B, E, F, G)
51. Wright, J. et al., 2005: Highly Constrained Topological Optimization of Heating, Ventilating, and Air-conditioning (HVAC) System (D, E, G)
52. Tay, J. et al., 2005: Evolving Dispatching Rules for Solving the Flexible Job-Shop Problem (B, D, E)
53. Bongard, J., 2005: Reinventing the Wheel: An Experiment in Evolutionary Geometry (A, F, G)
54. Maneeratana, K. et al. 2005: Multi-objective optimisation by co-operative co-evolution—ZDT1–ZDT6 benchmark problems (F, G)
55. Howard, D. et al., 2005: Solution of differential equations with Genetic Programming and the Stochastic Bernstein Interpolation (A, G)
56. Keijzer, M. et al, 2005: Determining Equations for Vegetation Induced Resistance using Genetic Programming (B, D, E, F, G)
57. Patel, S. et al, 2005: Patenting Evolved Bacterial Peptides (A, C, D)
58. Sekanina, L. et al., 2005: Evolutionary Design of Arbitrarily Large Sorting Networks Using Development (D, F, B) [99]
59. Wu, Z. et al., 2005: Optimizing Water System Improvement for a Growing Community (E, G)
60. O'Donnell, T. et al., 2005: Genetic Hybrid Transmit Antenna (A, D, E, G)