POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

An Efficient FPGA-Based System for Complete Evolution of Hardware Components

Relatore: Marco D. Santambrogio

Correlatore: Fabio Cancarè

Tesi di Laurea di:

Matteo RENESTO

Matricola n. 734678

Anno Accademico 2010–2011

Alla mia famiglia

Contents

1 Introduction **1**

1.1 Evolutionary computing and Genetic Algorithms 1

1.1.1 Basic Definitions 4

1.2 Evolvable Hardware 6

1.3 FPGAs as targets for Evolvable Hardware 7

1.3.1 FPGA reconfiguration approaches 8

1.3.2 Relation between FPGAs and Evolutionary Algorithms 10

1.3.3 Hardware Evolution with Genetic Algorithms 11

1.3.4 Issues 13

1.4 Classification of Evolvable Hardware Systems 13

1.5 Objectives of this thesis 16

2 Genetic Algorithms **18**

2.1 Genetic Algorithms 18

2.1.1 Simple Genetic Algorithm 20

2.1.2 Messy Genetic Algorithm 22

2.1.3 Compact Genetic Algoritm 23

2.1.4 Extended Compact Genetic Algorithm 26

2.1.5 Other Genetic Algorithms 27

2.2 Hardware-based Genetic Algorithms 28

2.2.1 First Hardware-based Genetic Algorithms 28

2.2.2 Hardware-based Compact Genetic Algorithms 33
2.3 Conclusions 39

3 FPGA-Based Evolvable Architectures 40
3.1 Extrinsic Approaches 41
3.2 Intrinsic Approaches 44
3.3 A Xilinx Virtex 4-based Evolvable Architecture 51
3.3.1 The Target Device 52
3.3.2 Virtex 4 Bitstream Manipulation 53
3.3.3 Evolvable Region Design 58
3.3.4 Individuals Interface 59
3.3.5 Performance 61

4 Proposed Methodology 63

5 Extrinsic Evolution Analysis 68
5.1 Simulation Framework 68
5.1.1 Simple Genetic Algorithm Implementation 71
5.1.2 Messy Genetic Algorithm Implementation 73
5.1.3 Compact Genetic Algorithm Implementation 75
5.1.4 Extended Compact Genetic Algorithm Implementation 78
5.2 Results Analysis 79
5.2.1 Parity generators 79
5.2.2 Complex multi-outputs functions 82
5.3 Preliminary considerations 84

6 The System-on-Chip Implementation 87
6.1 The System-on-Chip Architecture 88
6.2 Individuals evaluation 89
6.3 Hardware portability of Genetic Algorithms 91
6.4 New Hardware-based Compact Genetic Algorithm 94

6.4.1 Initialization 95
6.4.2 Generation phase 95
6.4.3 Update phase 99
6.5 Additional features 101
6.5.1 Elitism 101
6.5.2 Additional Mutation 102
6.6 Random Numbers Generation 103
6.7 Input-outputs operations and individuals deployment . . . 107
6.8 The Complete Architecture Deployed 109

7 Performance Analysis 111

7.1 Introduction to the case studies 111
7.2 Parity generator evolution 112
7.2.1 Optimal tuning 113
7.2.2 4 bits input function 113
7.2.3 5 bits input function 114
7.2.4 6 bits input function 116
7.2.5 7 bits input function 116
7.2.6 8 bits input function 117
7.2.7 Results summary 117
7.3 Extrinsic, Intrinsic and Complete evolution comparison . . . 119
7.4 Future Works 121
7.5 Not Only Evolvable Hardware 123

8 Conclusions 125

8.1 Results Achieved 125
8.1.1 Extrinsic Evolution System 126
8.1.2 Intrinsic Evolution System 126
8.1.3 Proposed System 127
8.2 Further Experiments 128

8.3 Concluding Remarks 129

List of Figures

1.1 FPGA structure 9
1.2 Genetic Algorithm for EHW 12

2.1 First Hardware Genetic Algorithm. [1] 29
2.2 Another Hardware-based Genetic Algorithm [2]. 32
2.3 One block of the Hardware Compact Genetic Algorithm . . 34
2.4 Topology of the Cellular CGA 37

3.1 Tyrrell's evolvable cell 48
3.2 Multi-device FPGA-based Evolvable Hardware 49
3.3 Virtex 4 CLB, with 4 slices 52
3.4 Internal structure of a slice, some details 53
3.5 Bitstream data flow. [3] 57
3.6 Evolvable region, structure of an individual. [4] 58
3.7 Evolvable region, individual interface 59

4.1 Steps toward a complete system 64

5.1 Cell simulation function 69

6.1 SoC EHW architecture 90
6.2 The testing module 91
6.3 Phases of the hardware-based CGA 95
6.4 Implementation of the hardware CGA 96

6.5 Generation phase 97
6.6 Update phase 101
6.7 High parallelism architecture 102
6.8 Random Numbers Generator, black box model 104
6.9 LFSR noise bit generator 105
6.10 Partial autocorrelation, computed with Matlab 107
6.11 Architecture implemented, FPGA Editor view 110

7.1 Hardware-based CGA evolution 4 bits 114
7.2 Hardware-based CGA evolution 5 bits 115
7.3 Hardware-based CGA evolution 6 bits 115
7.4 Hardware-based CGA evolution 7 bits 116
7.5 Hardware-based CGA evolution 8 bits 117
7.6 Hardware-based CGA evolution 118
7.7 Comparison between Extrinsic, Intrinsic and Complete evolution results 120
7.8 Comparison between Extrinsic and Complete evolution results 122
7.9 The implemented CGA solving 16384 bits OneMax problem 124

8.1 Evolution time with the implemented System 127
8.2 Estimation of larger system performance 128
8.3 Extended Evolvable Component 129
8.4 Evolution of an 8 bits parity generator, with the Extended Architecture 130

List of Tables

1.1 Classification of Evolvable Hardware systems 14

2.1 Hardware-based Genetic Algorithm performance 31
2.2 CGA, CoCGA and CCGA performance 38

3.1 Frame Address Register structure 55
3.2 EHW communication interface 60

5.1 GAs evolving parity generators 81
5.2 Performance evolving an accumulative counter 84
5.3 Performance evolving a multiplier 84

6.1 Random number sequence properties 106

7.1 Hardware-base CGA performance summary 118
7.2 CGA, hardware-based and software-based comparison . . . 119

8.1 Summary of achieved performance 129

List of Algorithms

1 Simple Genetic Algorithm 20

2 Compact Genetic Algorithm 25

3 Extended Compact Genetic Algorithm 27

4 Compact Genetic Algorithm Implementation 76

List of Abbreviations

EHW	Evolvable Hardware
EA	Evolutionary Algorithm
ES	Evolutionary Strategy
FSA	Finite State Automata
GP	Genetic Programming
PSO	Particle Swarm Optimization
GA	Genetic Algorithm
CGA	Compact Genetic Algorithm
SGA	Simple Genetic Algorithm
MGA	Messy Genetic Algorithm
ECGA	Extended Compact Genetic Algorithm
MPM	Marginal Product Model
BB	Basic Block
CoCGA	Co-operative Compact Genetic Algorithm
CCGA	Cellular Compact Genetic Algorithm

VRA Virtual Reconfigurable Architecture

PV Probabilistic Vector

HGA Hardware-based Genetic Algorithm

pe-CGA Persistant elitism Compact Genetic Algorithm

ne-CGA Non-persistant elitism Compact Genetic Algorithm

FPGA Filed Programmable Gate Array

FPAA Filed Programmable Analog Array

ASIC Application Specific Integrated Circuit

LUT Look-Up-Table

CLB Configurable Logic Block

FF Flip-Flop

IOB Input-Output Block

BRAM Block Random Access Memory

PPC PowerPC

ICAP Internal Configuration Access Port

LFSR Linear Feedback Shift Register

MUX Multiplexer

PLA Programmable Logic Array

CPLD Complex Programmable Logic Device

CT Circuit Parameter Tuning

CD Complete Circuit Design

SoC System-on-Chip

RNG Random Number Generator

FAR Frame Address Register

FDRI Frame Data Register Input

Summary

Evolvable Hardware (EHW) is a new field of research. It concerns the creation and the adaptation of physical circuits, through the usage of evolutionary techniques. Its key elements are (i) the Evolutionary Algorithms (EAs), (ii) the device on which the circuits can be deployed and (iii) the integration of these two elements. There exist two main techniques of evolution: Extrinsic evolution and Intrinsic evolution. The former is a simulation-based evolution done offline. The latter is an online approach. Evolution is accomplished by deploying each individual.

Due to their characteristics, Field Programmable Gate Arrays (FPGAs) are the most used hardware devices in EHW works.

In this work we use a Xilinx Virtex 4 FPGA, its Configurable Logic Blocks (CLBs) are composed of four Slices. Each slice is composed of two LUTs (Look-Up-Tables) whose functionality is determined by a 16 bits array stored into a configuration memory. Through modifications of the content of the configuration memory it is possible to change the functionalities implemented on the device. The definition of an individual, the circuit that is evolving, as a portion of the reconfigurable area of an FPGA makes possible to change the function that it implements. When a new population is generated the old one can be wiped out and the area can be used to implement the new one.

An important element embedded in the Virtex 4 FPGA is the Internal Configuration Access Port (ICAP). It allows a self-adaptive behavior,

providing to the implemented architecture access to its own configuration memory. Thanks to the ICAP, multi-device solutions are no more required to use an intrinsic approach, because ICAP allows deployment of new individuals at runtime.

The purpose of this thesis is to develop an efficient system to evolve hardware circuits. That is done implementing an hardware architecture for the Complete evolution of hardware components with an 8 bits data-path and a genotype of 1024 bits. A Complete evolvable system is a step forward from the Intrinsic systems. In such kind of systems also the Evolutionary Algorithm is hardware implemented on the same device of the evolvable circuit. To implement a System-on-Chip (SoC), able to make Complete evolution, without the needs of external interaction, may help to increase the efficiency of the system. The reasons beyond a complete implementation are various, such architecture is able to achieve higher speed, respect to a software implementation or a multi-device implementation constrained by communication bottlenecks.

In this thesis, an efficient SoC, with an hardware genetic algorithm, has been designed and implemented. To prove the validity of the proposed approach, the implementation of a Complete Evolvable System, the proposed architecture has been evaluated, to estimate its performance, evolving some hardware component.

The thesis is organized as follows. Evolvable Hardware is introduced in **Chapter 1**. This chapter describes the key elements, it presents the terminology and the classification most widely used. The last section of this chapter introduces the approach proposed in this thesis and presents the issues that are addressed. **Chapter 2** focuses on the first key element of Evolvable Hardware, describing the state of the art in Genetic Algorithms field. A good understanding of the main Genetic Algorithms is important for the objectives of this thesis. **Chapter 3** describes works proposed in liter-

ature on EHW. In **Chapter 4** the methodology used to analyze EHW issues, and to develop an efficient hardware architecture, is described. The two most important preliminary step of such methodology are described with more details in **Chapter 5**. In **Chapter 6** the implementation of the proposed hardware system is described, a lot of details are provided.

Chapter 7 shows the results obtained evolving some hardware components with the developed system. Such results are compared to those that is possible to achieve with Intrinsic and Extrinsic evolution methodologies to highlight le capability of the developed system. In this chapter some possible future works are also proposed. **Chapter 8** presents the results obtained in this thesis making furthers theoretical considerations.

Sommario

L'Hardware Evolvibile costituisce un nuovo campo di ricerca. Esso riguarda lo sviluppo e l'adattamento di circuiti fisici tramite l'utilizzo di tecniche evolutive. Ie sue componenti principali sono quindi gli Algoritmi Evolutivi (i), i dispositivi su cui i circuiti vengono implementati (II) e l'integrazione tra questi elementi. Esistono due principali tecniche di evoluzione hardware: l'evoluzione Estrinseca e quella Intrinseca. La prima è basata su di una simulazione *offline* del dispositivo. La seconda è un approccio *online*, in cui tutti gli individui generati vengono implementati.

Grazie alle loro caratteristiche, le Field Programmable Gate Arrays (FPGAs) sono i dispositivi hardware più utilizzati per la creazione di sistemi hardware evolvibili. L'FPGA utilizzata per questa tesi è la Xilinx Virtex 4, i sui blocchi logici configurabili (CLB) sono composti da 4 *slices* ciascuno. Ogni *slice* è composta da due Look-up-tables (LUTs), la cui funzionalità è determinata da stringe di 16 bits contenute nella memoria di configurazione. Tramite modifiche al contenuto di questa memoria, è possibile cambiare le funzioni implementata sul dispositivo. La definizione di un Individuo, ovvero del circuito che sta evolvendo, come una porzione dell'area di una FPGA, rende possibile cambiare la funzione implementata da essa. Ogni qualvolta una nuova popolazione di individui è generata, quella vecchia può essere cancellata, e l'area dell'FPGA utilizzata per implementare la nuova.

Di particolare importanza, sulla FPGA Virtex 4, è l' *Internal Configu-*

ration Access Port (ICAP). Essa permette il funzionamento auto-adattativo del dispositivo consentendo l'accesso alla memoria di configurazione all'architettura implementata sul FPGA. Grazie all'ICAP non sono più necessarie soluzione multi-dispositivo per poter implementare sistemi basati su Evoluzione Intrinseca, in quanto permette l'adattamento a runtime delle funzionalità implementate.

Lo scopo di questa tesi è sviluppare un Sistema efficiente per l'evoluzione di componenti hardware. Ciò è fatto implementando un architettura hardware per l'evoluzione Completa di componenti hardware con un data path di 8 bits ed un genotipo di 1024 bits. L'evoluzione Completa costituisce un ulteriore ottimizzazione dell'evoluzione Intrinseca, in quanto nei sistemi di evoluzione Completa anche l'algoritmo evolutivo è implementato in hardware sullo stesso dispositivo del circuito che sta evolvendo. Implementare una architettura System-on-Chip (SoC), in grado di fare evoluzione Completa, senza bisogno di interazioni esterne può aiutare a migliorare l'efficienza del sistema. Le ragioni dietro tale soluzione sono varie, essa raggiunge maggiore velocità di evoluzione rispetto ad un implementazione software o ad un architettura basata su multipli dispositivi e vincolata da collo di bottiglia dovuti alla comunicazione.

In questa tesi è stata implementata una efficiente architettura SoC, con una algoritmo genetico implementato in hardware. Al fine di mostrare la validity dell'approccio proposto, ovvero l'implementazione di un sistema che esegua evoluzione Completa, le performance dell'architettura implementata sono state valutate evolvendo alcuni componenti hardware.

Questo documento di tesi è organizzato come segue. L'Hardware Evolvibile è introdotto nel **Capitolo 1**. Questo capitolo descrive descrive gli elementi chiave e presenta la terminologia e la classificazione più largamente usate. L'ultima sezione di questo capitolo introduce l'approccio proposto e le problematiche che sono affrontate. Nel **Capitolo 2** l'attenzione è rivolta

al primo degli elementi chiave dell'hardware Evolvibile, qui è descritto lo stato dell'arte nel campo degli Algoritmi Genetici. Il **Capitolo 3** presenta brevemente la letteratura nel capo dell'Hardware Evolvibile. Nel **Capitolo 4** è descritta la metodologia utilizzata per analizzare le problematiche dell'Hardware Evolvibile e sviluppare un architettura hardware efficiente. I due passi preliminari più importanti di questa metodologia sono descritti con più dettagli nel **Capitolo 5**. Nel **Capitolo 6** è descritta l'implementazione del sistema hardware proposto, numerosi dettagli sono qui forniti. Il **Capitolo 7** mostra i risultati ottenuti evolvendo alcuni componenti hardware con il sistema sviluppato. Al fine di sottolineare le possibilità di tale sistema, i risultati ottenuti sono stati comparati con quelli che è possibile ottenere tramite evoluzione Estrinseca e Intrinseca. Alla fine del capitolo sono proposti alcuni possibili sviluppi futuri. Il **Capitolo 8** discute i risultati di questo lavoro presentando ulteriori considerazioni teoriche.

Chapter 1

Introduction

This thesis work focuses on *Evolvable Hardware* (EHW), a new field of research. It concerns the creation or the adaptation of physical circuits through the usage of evolutionary techniques. Its key elements are the *Evolutionary Algorithm* (EA), the *device* on which the circuits can be deployed and the *integration* between these two elements. The first section of this chapter introduces the main concepts and definitions about evolvable computing. While the methods applicable to the hardware are explained in the second section. In the third section the classification usually adopted for Evolvable Hardware will be presented. Finally, the objectives of this thesis are introduced in the last section of this chapter.

1.1 Evolutionary computing and Genetic Algorithms

The evolutionary computing concept has been introduced in the '60 by L.Fogel with a series of works [5] on Artificial Intelligence. He proposed to build intelligent agents able to predict the environment and turn the prediction to an adeguate response. The environment and the response can be described both as sequences of symbols. Fogel populated the environment with an initial set of different Finite State Automatas (FSAs) and de-

fined a fitness function to measure the correctness of their behaviour. Such automatas were able to take as input the description of the environment and return as output a response. The structure of all the initial machines was randomly created. They were characterized by a different number of states and different transitions between these states. The fitness of these machines has been evaluated in the environment simulating a biological behaviour. The next step was to put all these machines in a list, sorted according to their fitness value. The list was divided in two halves saving the top half best performing automata and discarding the remaining ones. Machines belonging to this top half were able to *reproduce* themselves. New machines were generated from random mutations of those considered. Mutations were introduced adding a state to a machine or modifying its transitions with uniform probability. The new *offspring* were evaluated in the environment as the previous one. Proceeding iteratively it was possible to obtain a population able to adapt to the environment. Fogel's work is particularly relevant because it is the first research in which concepts that belong to biology, as population, offspring, fitness, mutations and selection are introduced also in the computer science world. Subsequently several approaches that exploit evolutionary concepts have been proposed. It is possible to group them in some main categories: *Evolution Strategies* (ES) [6], *Genetic Programming* (GP) [7], *Particle Swarm Optimization* (PSO) [8], *Genetic Algorithms* (GAs) [9].

Evolution Strategies [10] have been the first approach developed after Fogel's researches by Rechenberg and Schwefel which aim to simulate evolution. Fogel's work is not usually classified as an Evolutionary Strategy. In Fogel's methodology it is possible to identify the structure of a FSA as the genotype and its implementation as the fenotype. Generally speaking, in Evolutionary strategies the difference is not so strict. In recent years also hardware implementations of evolutionary strategies have been proposed

[11] *Particle Swarm Optimization* [8] techniques are similar to *Evolutionary Strategies* [6]. In this case there is a population, whose members are called *particle*. They have to explore an n-dimensional space. They procede with a certain speed, that depends on the size of each update of their representation. The overall goal is to minimize (or maximize) an objective function.

Another application of evolutionary concepts in computer science is *Genetic Programming* [7]. It has been proposed the first time by J. R. Koza [12]. It has been suggested that also programs, or at least structures, can be subject to an evolutive process. Following the Genetic Programming approach it is necessary to define first a series of atomic operations. From these elements it is possible to build a program defining the relation between the atomic operations by organizing them within a tree. Nodes are the elementary operations and connections define which values are taken as input by an operation. Every function takes as inputs the return values of its leafs and forward its outputs to its parent node. Such structure has the important characteristic to be dynamic. The trees can be changed by adding or removing nodes, or modifying the connections among them. These trees can be used as individuals of a population. Similarly to Fogel's approach, it is possible to select the bests, apply mutations or crossover and improve in this way the overall performance. All these techniques have been developed to address some specific categories of problems, but the most general approach inspired from biological evolution are *Genetic Algorithms* [9].

Genetic Algorithms have been proposed by Holland as *an efficient search mechanism in artificially adaptive systems* [9]. GAs provide an higher decoupling between the genotype of an individual and its phenotype, mutations are no more seen as addition of a state, or a module, but as an operation done on genotype. They simulate the evolutionary process of a population that happens in nature. A population is a set of candidate solutions for a given problem. GAs procede toward better solutions by iteratively

evolving the population. Following a more biology-like terminology, these elements can also be called *individuals*. Every individual is described by a series of symbols called genes. They constitute the genotype of the individual. It can be implemented as a series of bits. A function F associates every possible genotype to a value called fitness. The goal of a Genetic Algorithm is to maximize the fitness of the population, and generally return the individual with the best fitness, that is nothing different from the best solution reached. The fitness evaluation can be an extremely complex function or a trivial operation as counting the number of bits in the genotype that are set to one. The main operations, generally done by a Genetic Algorithm, beyond the evaluation of the individuals are: *Selection*, *Mutation* and *Crossing-over*.

- Selection follows the same principle of the *selection* done by Fogel, but GAs instead of taking the best half may use other criteria.
- Mutation consists in modifying randomly one or more elements in the genotype.
- Crossing-Over consists, given a pair of individuals, in swapping elements between their two genotypes.

Crossing-Over introduces mutations that are not random but obtained propagating the genotype of an individual in the population. More modern GAs [13] [14] introduce also other operations, different implementations or a different representation of the population [15]. Genetic Algorithms are among the most important tools used in this thesis work, they will be further analyzed and described in Chapter 2 and Chapter 5.

1.1.1 Basic Definitions

Evolutionary Computing terminology comes from the natural science world, but terms used have often a slightly different meaning in computer

science literature [16]. The most important terms that is necessary to introduce are:

- Individual: it is a candidate solution generated by the evolutionary strategy. It is completely described by a chromosome which defines its behaviour, that is the relation between its inputs and its outputs. For each individual a *fitness* value can be computed. How it can be done will be discussed later.
- Population: a set of individuals. Usually it has a defined and fixed size.
- Chromosome: often is represented as an array of bits, it is a list of symbols that defines in an unique manner all the characteristics of the individual which it refers to. It is the abstract representation of the individual that will be subject to the evolution process. The chromosome is the part that is manipulated by the evolutionary strategy.
- Genotype: it is used as synonym of Chromosome when one talks about genotype of an individual.
- Gene: one element of the chromosome. If the chromosome is a string of bit, it will be one bit. Usually random *mutations* act at this level of granularity.
- Allele: it is the value that a specific gene may assume in a chromosome.
- Locus: the position of a Gene in a chromosome.
- Fitness: the measure of how good is the individual performing in the given task. It may consider multiple objectives.

1.2 Evolvable Hardware

Evolvable Hardware (EHW) is a new field of research firstly explored by Higuchi since 1993 in his research on *Evolvable Hardware with Genetic Learning* [17]. It concerns the creation or the adaptation of physical circuits, through the usage of evolutionary techniques. Its aim is to create optimal performance devices by tailoring the architecture to the specific requirements of a given problem. EHW can become useful in many cases:

- To achieve a more efficient implementation, that saves more resources or allows an higher performance respect to a traditional approach;
- To be able to develop an hardware component also when its task can not be described completely in an analytic way, but only partially, or when it is too complex to do it;
- To implement a component whose characteristics may chance at run-time.

This last scenario requires an online adaptation of the hardware devices that is usually not possible with a traditional implementation of the hardware component. An online adaptation can be also exploited to deal with errors or damages to the hardware devices. The capability of the system to self-repair though the evolution of a different configuration can improve its reliability [18].

It does not exist yet a formal definition, accepted by the whole academic world, of what Evolvable Hardware is. However, it is possible to consider as a good definition the one proposed by Jim Torresen:

- Evolvable Hardware (EHW) is a scheme inspired by natural evolution, for automatic design of hardware systems. By exploring a large design search space, EHW may find solutions for a task, unsolvable,

or more optimal than those found using traditional design methods [16].

The key elements of an EHW system are (i) the *Evolutionary Algorithm* (EA), (ii) the *device* on which the circuits can be deployed and (iii) the *integration* between these two elements. For what concerns EA, the application of Genetic Programming (GP) to the hardware world can be seen as a first example of Evolvable Hardware. Works proposed in literature that exploits GP often focus on evolution with a set of predefined high level functions, or modules. Differently, proposed works that use GAs generally focus on a more fine-graned approach. The main difference between Genetic Programming and Genetic Algorithms is that the first allows to modify the size of the solution adding or removing funcional blocks, while the second kind of works on a predefined number of *evolvable cells*. In GP the individual representation is based on trees, it allows to remove leafs or branches, in GAs representation is based on chromosomes, that are often arrays of bits of fixed length. To reduce the search space, in many proposed works with GAs the connections among the cells are fixed.

1.3 FPGAs as targets for Evolvable Hardware

For what concerns physical devices, Filed Programmable Gate Arrays (FPGAs) are the most used ones. Few projects have been developed using Application Specific Integrated Circuits (ASICs) technology [19]. As well some EHW-oriented architectures [20] have been developed, but they have not obtained success among the researchers. This is probably due to the high cost of using non-commercial devices. The largest part of EHW solutions have been developed using programmable logics, such as *Programmable Logic Arrays* (PLAs), Complex Programmable Logic Devices (CPLDs) or FPGAs. Few exploit *Filed Programmable Analog Arrays* (FPAAs) or other

analog solutions for the evolution *in materio* [21]. The idea beyond this last approach is that conventional methodologies for the creation of EHW systems do not exploit at best all the hardware possibilities. Moreover, the usage of non conventional devices allows to reach extremely efficient results [22] However this is an extremely fine-grained approach, if compared with a more traditional FPGA-based solution, since it works with lower level components. Devices as FPAAs allow to work at transistor-level in order to compose logic blocks [23]. The usage of FPGAs or CPLDs allows to define better the level of granularity, although a more structural rigidity, they are less affected by problems related to scalability. In particular FPGAs allow also to have an higher flexibility respect to CPLDs. Using *Complex Programmable Logic Devices* it is only possible to modify the configuration of the connections between the base elements. FPGAs allows a further flexibility due to the possibility to change also the configuration of their logic blocks called Configurable Logic Blocks (CLBs). FPGAs are configured applying a binary configuration called *bitstream*. This bitstream can be sent to the configuration memory of the FPGA through a configuration port. Most FPGAs allow to change configurations multiple times. Only few of them can be configured only once. Obviously this last family of FPGA is not suitable for building Evolvable Hardware.

Figure 1.1 shows the structure of a generic FPGA. It is possible to see that on the boundary there are Input-Output Blocks (IOBs). While configurable nets connect a grid of Configurable Logic Blocks (CLBs). The internal structure of the CLB depends strictly on the FPGA considered but generally it is made of one or more Look-Up-Tables (LUTs) and other additional resources like Flip-Flops (FFs), multiplexers or multipliers.

1.3.1 FPGA reconfiguration approaches

There are three main approaches reconfigure an FPGA:

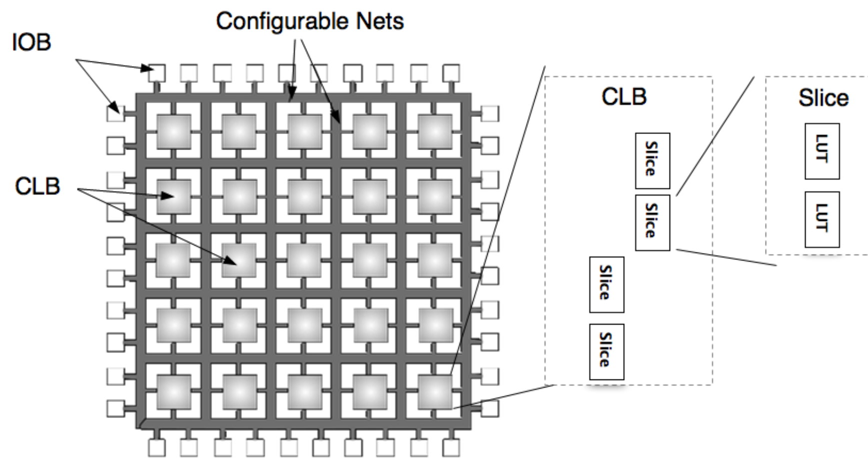


Figure 1.1: FPGA structure

- The first is called *complete* [24], because it modifies the configuration of the whole FPGA.
- The second is called *module based* [25], because it alters just a portion of the area of the FPGA. It allows to change the hardware module implemented in a predefined area of the FPGA. It requires to define fixed connections between the reconfigurable module and the static part.
- A third one is called *difference based* [25], because it makes small changes to the configuration of the FPGA acting just on the configurations of those components which have to change the implemented functionality. How it is possible to modify the configuration of the FPGA is a detail which depends strictly on the FPGA used and the development tools available.

In particular, the majority of the works proposed in literature, including this thesis, use Xilinx FPGAs. Xilinx Inc. is a leader in the market of FPGAs and it produces devices with high flexibility and characteristics suitable for

the implementation of dynamic components.

1.3.2 Relation between FPGAs and Evolutionary Algorithms

For what concerns the *integration* between EA and device, there are two main categories of systems according to the structure of the architecture that is evolving. It is possible to distinguish between *Complete Circuit Design* or *Circuit Parameter Tuning*. The two architectures can be described as follow:

- Circuit Parameter Tuning (CT) consists in designing an hardware component whose functionality is determined by a series of parameters.
- Complete Circuit Design (CD) consists in the evolution of the whole circuit, where all the features that characterize the hardware component can evolve.

However it is not always possible to classify an EHW system according to this schema. It is possible to say that to design an ad-hoc core whose functionality depends from values contained in a *configuration* register is an architecture based on Parameter Tuning. While to design an architecture at low level with an evolutionary approach is a Complete Circuit Design. Uncertainty in the classification may come from the usage of programmable logics. In this case only a set of parameters of the circuit can be changed. The configuration of the FPGA changes, but the structure of the device does not. Architectures based on programmable logics are classified as CD and not as CT because the architecture itself has not been designed with an ad-hoc purpose.

It is possible to say that an EHW architecture has a certain degree of evolvability based on the amount of its characteristics that it is possible to evolve and the amount of those which are statically defined. Due to their relatively small price and large commercial availability configurable hard-

ware is the most commonly used device for the implementation of EHW architectures. Focusing on this case it is possible to give a new definitions of CT and CD architecture design:

- Circuit Parameter Tuning means that on the programmable logic is implemented a preliminary hardware component whose functionality is defined by a chromosome.
- Complete Circuit Design means that directly the functionality that the programmable logic, or just a portion of it, implements it is evolved.

Methodologies based on CT often use *Virtual Reconfigurable Architectures* (VRAs) [26], building ad hoc IP Cores that simulate the behaviour of a generic programmable logic just overcoming some limits concerning the reconfigurability. Approaches that focus on CP act directly on the function implemented on the target devices. In the case of programmable logics it is defined by the content of a configuration memory. Such memory is updated using bitstreams. For this reason techniques for CD are often called *Bitstream manipulation techniques*. They build custom bitstreams to deploy the EHW component.

1.3.3 Hardware Evolution with Genetic Algorithms

Figure 1.2 shows the steps of a Genetic Algorithm that can be used to generate EHW components. Different solutions proposed may apply different genetic operators, as selection, mutation and crossing over. Usually just the overall approach is shared between different algorithms. At the beginning a set of circuits, the population, is randomly generated. Every circuit is evaluated computing its fitness. If none of the already generated circuits reaches a sufficient grade of performance, new circuits are generated from the best ranked in the current population through the usual operations of Mutation and crossing over.

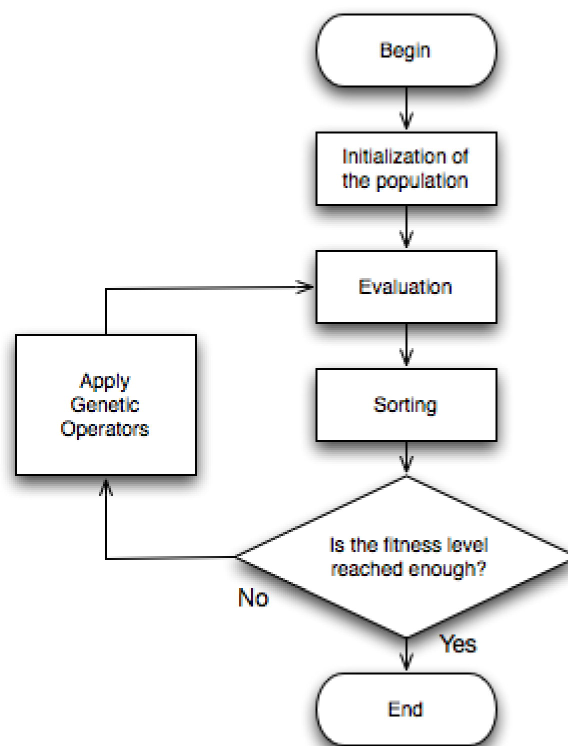


Figure 1.2: Genetic Algorithm for EHW

1.3.4 Issues

Nowadays there are three main issues concerning the realization of EHW systems. They are:

- Scalability;
- Fitness Computation Time;
- Behavioural and Structural Analysis capability.

True hardware components are described by an high number of parameters. This leads to a long chromosome and a large genotype. A growing genotype size means a growing search space and an higher number of generations necessary to converge to an optimal solution. It increases also the concrete risk of stalling in local maximum. It may happen that the population converge toward a suboptimal solution and the variation size is not enough to step out from that partial result. All EHW architectures must take care of this *Scalability* problem. *Fitness computation* is also a computationally expensive task which may have high time requirements. The third problem *Behavioural and Structural Analysis capability* is common to other logics as Neural Networks, where the dynamic implementations may increase the difficulty to understand exactly the behavior of the implemented component. In Chapter 3 it will be described how works proposed in literature deal with these three issues. Analyzing the main Genetic Algorithms, it will be discussed how these issues impact on them.

1.4 Classification of Evolvable Hardware Systems

The most important classification of EHW systems can be done mainly based on how are evaluated the individuals and how is implemented the Genetic Algorithm. There are four main categories of evolution: *Extrinsic*, *Intrinsic*, *Complete* and *Open-Ended*.

Table 1.1: Classification of Evolvable Hardware systems

	Extrinsic	Intrinsic	Complete	Open-Ended
Task Definition	SW	SW	SW	HW
Evolution	SW	SW	HW	HW
Evaluation	SW	HW	HW	HW
Final Deployment	HW	HW	HW	HW

Extrinsic evolution is the first EHW approach and the most simple. The key characteristic that allows to classify an evolvable architecture as *Extrinsic* is that no individual is implemented in hardware but the best performing solution obtained after the evolution process.

In Extrinsic evolution, the key phases of the realization of an EHW system are subdivided between software (SW) and hardware(HW) how it is in table 1.1. All intermediate solutions are evaluated estimating their fitness but without being implemented. This is usually done implementing a framework able to simulate the behavior of a candidate individual by its genotype and to return its expected fitness. Often, also the evolutionary strategy is implemented with a software approach. Extrinsic Evolution main constraint is the limited capability of the software to exploit the high parallelism typical of Genetic Algorithms.

Intrinsic Evolution approach aims to improve the efficiency of the system thanks to a real implementation of all the individuals. As Table 1.1 shows the key aspect that distinguishes Intrinsic evolution from the Extrinsic is that the evaluation of all the generated individuals is not done estimating their fitness with a software simulation, but implementing them all in hardware. This allows to hardware accelerate the fitness computation and to parallelize the operation on different individuals concurrently. To have real benefit from an Intrinsic approach it needs that the overhead of time required to implement the individuals is smaller than the speed-up

achieved thanks to the hardware implementation. It is considered Extrinsic Evolution also the case in which the evolutionary strategy is, partially or completely hardware accelerated, but on a third part device.

Complete Evolution approach consists in implement both the evolutionary strategy and the EHW components in a System-on-Chip architecture. Such definition has been introduces by P. Haddow and G. Tufte presenting their hardware robotic controller [27]. As Table 1.1 shows, the only external interaction consists in the task definition which is set by the user loading the evolution data in an on-chip memory or with an equivalent approach. The reasons beyond an Intrinsic implementation are various, such architecture is able to achieve higher speed, respect to a software implementation or a multi-device implementation constrained by communication bottlenecks. Unfortunately it is immediate to see that there are two main issues with this approach. First, it is necessary to have a device that allows to modify its behavior following the evolution requirements at runtime and internally. It needs to be dynamically *internally* reconfigurable. Deploying new individuals on the evolvable area requires to be able to change the logic function implemented. Only few hardware devices allow to do that. The second issue concerns the scalability of the evolvable architecture. As already discussed real hardware components may be defined by a large chromosome. To hardware implement a parallel Genetic Algorithm that manages them may become expensive in terms of occupied resources. Availability of area puts limits to the exploitable parallelism. With the given definitions may remain uncertain the classification of an embedded evolvable systems that implements in hardware the EHW components but run the Genetic Algorithm in software or with a partial hardware acceleration. It could be classified both as Intrinsic or Complete. However it is possible to label a system in which the software part is the main one as Intrinsic and an architecture that executes the Genetic Algorithm mainly in hardware as Com-

plete.

The last class is *Open-Ended* evolution. An Open-Ended evolvable architecture consists in an architecture for complete evolution that is also able to evaluate the behavior of the evolutionary strategy and the evolved EHW component. To see if it is correct or not, and in the latter case to evolve a new hardware component that matches with the new requirements of the environment, without having necessity of input from the user but having a *Sense-Plan-Act* approach [28].

1.5 Objectives of this thesis

The main objective of this thesis is to develop efficient FPGA-based architecture for the evolution of hardware components. With this aim, it has been implemented an efficient Complete Evolvable System able to achieve good performance, thanks to an hardware-based implementation of the Genetic Algorithm on the same device used to evaluate the fitness of the evolvable individuals.

To be able to make good design choices and to implement such system, it has been first developed an Extrinsic Evolvable Hardware system, which is able to evolve, using the main Genetic Algorithms and EHW individual that could allow also Intrinsic or Complete evolution. It improves the state of the art, by allowing better understanding on how the problems of scalability, observability and fitness computation impact on the design and the implementation of Extrinsic and Complete Evolvable Hardware systems. One of the limits of Extrinsic approaches is the computation complexity of the fitness evaluation. Intrinsic techniques that implement in hardware the fitness evaluation can allow to overcome this limit. However, systems based on Intrinsic evolution need to be connected to workstation to achieve reasonable time[1] performance during the evolution. Once implemented in

[1]Few hours can be a reasonable time, but it depends on the complexity of the considered

hardware the fitness computation, a step forward, that has been done, is the hardware acceleration of the whole architecture, with also an hardware-based implementation of the Evolutionary Algorithm.
This field of research has not been extensively explored yet, but the implementation a System-on-Chip (SoC) architecture can help to increase the efficiency of the system, because it will able to make Intrinsic evolution, without the needs of external interactions. The reasons behind a Complete Evolvable system are various, it can be able to achieve higher speed, with respect to a software implementation or a multi-devices implementation constrained by communication bottlenecks.
In this thesis a Complete EHW system has been realized. In Chapter 6 the proposed SoC Complete Evolvable architecture is described, then some case studies are presented to show the efficiency of the system.

The next chapter will introduces the main families of Genetic Algorithms, while Chapter 3 will describe the state of the art in Evolvable Hardware field.

---
system.

Chapter 2

Genetic Algorithms

To develop an efficient Evolvable Hardware (EHW) system is a complex problem, it requires first a deep understanding of Genetic Algorithms. In this chapter the main GAs are presented. The first section contains a description of the GAs rationale and basic concepts; then some of the most significant GAs developed are presented. The second section of the chapter describes some GA implementations based on parallel hardware technologies. Some Genetic Algorithms have already been implemented in hardware, it is useful to see if some of them match good with the Evolvable Hardware requirements. However, not all Genetic Algorithms are suitable for being implemented using parallel hardware.

2.1 Genetic Algorithms

In the introduction it has been recalled the first Genetic Algorithm proposed by Holland [9]. With respect to the previous evolutionary approaches, Holland introduced the idea of decoupling phenotype and genotype by introducing the *chromosome*. According to such approach, the genetic operators are applied to the chromosome, while the fitness of the individual is generally calculated on the phenotype. In the earliest evolutionary

approach proposed by Fogel [6] and in the first Evolutionary Strategies by Rechenberg and Schwefel [10], the new individuals were obtained just through mutation. Holland proposed a new genetic operator: *crossing-over*.

Another innovation brought Holland studies concerns the selection process. According to the selection method proposed by Fogel, the best performing half of the population is selected to generate new individuals. Holland introduced a different methodology for selection, he proposed to reproduce each parent in proportion to its relative fitness. The same approach has been implemented by Goldberg with the *Roulette Wheel Selection* [29]. It consists in selecting the chromosomes, to be used as parents of the next generation, with a likelihood that depends on their relative fitness.

These two selection methodologies are not the only existing. In 1995 Goldberg and Miller proposed also *Tournament Selection* [30]. They stated that the convergence speed of a Genetic Algorithm depends heavily on the *selection pressure*. It is possible to define the selection pressure as the likelihood of the best individual to be selected. In the earliest simple selection method there is high selection pressure, because the best individuals are are always selected. In the *Roulette Wheel Selection* there is potentially a low selection pressure. If all individuals have similar fitness the probability of being selected is almost uniform.

Goldberg and Miller noticed that a too low selection pressure slows down the convergence speed, while a too high selection pressure increases the risk of stalling in a sub-optimal solution. *Tournament Selection* allows to scale the selection pressure by defining the size of the tournaments. It allows further degree of freedom in tuning the algorithm parameters. In tournament selection the population is randomly divided into groups of equal size. In each group the best individual is selected as tournament winner. The group of the tournament winners can be subject to a further selection, by discarding those individuals that have a fitness value below the

average. Selection pressure can be increased by just increasing the size of the tournaments. On average the winner of a large tournament is expected to have a fitness higher than the winner of a smaller tournament.

After Holland researches many Genetic Algorithms have been proposed. The most relevant and widely used are: *Simple Genetic Algorithm*, *Messy Genetic Algorithm*, *Compact Genetic Algorithm* and *Extended Compact Genetic Algorithm*. These algorithms will be descried in detail in the next subsections.

2.1.1 Simple Genetic Algorithm

The Simple Genetic Algorithm (SGA) is the most used GA, it was proposed by Goldberg in 1989 [29]. It defines concepts belonging to the original proposal by Holland in a more formal way. It has a general purpose aim and can be implemented using one or more genetic operators. Algorithm 1 describes the SGA structure. At the beginning the first population is obtained generating random individuals. The genetic operators are applied on the chromosomes, the individuals representation stored in memory. The phenotype is considered only during the fitness *Evaluation*.

Algorithm 1 Simple Genetic Algorithm

1: $t \leftarrow 0$
2: $P(t) \leftarrow NewRandomPopulation()$
3: $Evaluation(P(t))$
4: **while** $!Termination(P(t))$ **do**
5: $\quad t \leftarrow t + 1$
6: $\quad P(t) \leftarrow Selection(P(t-1))$
7: $\quad CrossingOver(P(t))$
8: $\quad Mutations(P(t))$
9: $\quad Evaluation(P(t))$
10: **end while**

The algorithm iterates until a final conditions is fulfilled. Usually there are three kinds of possible termination conditions:

- An individual with fitness above a predefined target threshold is generated;
- At least N iterations have been performed;
- Variance in chromosomes or fitness values is below a threshold.

The first termination condition requires to know in advance what is the best fitness value achievable. The second is often useful when there is no such knowledge, as for example when the algorithm is used to solve a min-cut problem. The third condition avoids useless iterations with a population that is not optimal and neither has sufficient variance to lead to improvements. The second and the third condition can be used also to avoid infinite loops.

At every iteration selection is done; all the three presented methodologies of selection can be used with Simple Genetic Algorithm. The choice of which one has to be used is up to the developer, and it generally depends on the characteristics of the problem to be solved. In SGA new individuals are obtained through crossing-over after the selection. For each pair of parent individuals in the selection set, it is possible to obtain new individuals for the new generation by crossing-over. It is also possible to introduce *elitism* in the generation. It is also known as *replace-the-worsts* policy and it avoids to lose solutions already reached increasing the convergence rate, but it increases also the risk of stalling in local maximum. After having generated the new population, mutations are applied to its individuals. Introducing mutation after crossing-over allows to explore new solutions introducing new alleles in the genes of the chromosome.

2.1.2 Messy Genetic Algorithm

The intuition behind the Messy Genetic Algorithm [13] (MGA) is to introduce further flexibility by building first Basic Blocks (BBs) and then combine them. The algorithm works executing two different phases:

- *Primordial Phase*. A series of *Basic Blocks* (BBs) are identified within the chromosome.
- *Juxtapositional Phase*. Basic Blocks identified during Primordial Phase are combined to obtain a candidate optimal solution.

Basic Blocks are sets of genes of a chromosome, not necessarily contiguous. Using Simple Genetic Algorithm it is possible to consider the linkage between contiguous genes increasing the granularity of the crossing over. For example swapping two couples of bits instead of just one. Such approach is not able to consider linkage between non contiguous genes. Messy Genetic Algorithm overcomes this limit by building basic blocks that can include also non-contiguous genes.

Primordial phase firstly creates a series of candidate Basic Blocks from the individuals belonging to the population, then it evaluates them to determine the set with the best Basic Blocks. To evaluate the fitness of these BBs to determine the bests is not a simple operation. The authors suggest to evaluate directly just a chunk of the chromosome, if it is possible, but such approach breaks the convention of Genetic Algorithm to think the individual as a Black-Box. It is not always possible, or recommendable to do that. Aiming to solve this issue an alternative methodology to evaluate blocks has been proposed. It consists in completing the missing part of a Basic Block with those from a *template chromosome*. Once obtained a complete individuals from the Basic Blocks it is possible to evaluate them without issues. As template chromosome it can be used the one describing the best individual of the previous generation, or a random template

during the first generation.

Best Blocks identified are used in the second phase to build the individuals of the population. "The objective of doing so is to create an enriched population of Basic Blocks whose combination will create optimal or very near optimal strings" [13]. Selection among individuals is done with *Tournament selection* without replacement.

The algorithm is called *Messy* because, once identified a set of parents, an offspring is no more obtained just with a *neat* crossing over but with the *Messy* operators *Cut* and *Splice*. The aim of these new operators is not to break BBs already identified, reducing the perturbation introduced by the crossing-over recombination. During the *Cut* operation a couple of individuals are each split in two chunks, without breaking their BBs. *Splice* operator recombines chunks from a couple of individuals creating two new individuals. Small random mutations are introduced in the new individuals to allow exploration of novel solutions considering new Basic Blocks. Another innovation introduced in Messy Genetic Algorithm is the possibility to shift a BB from a position in the chromosome to another. This operation is used seldomly, since in the largest part of the works the chromosome defines with a gene in specific locus some specific feature. It is meaningless to move genes from one locus to another, because they have no relationship.

The overall flow of the algorithm does not introduce further changes to the one of SGA besides those just described.

2.1.3 Compact Genetic Algoritm

The Compact Genetic Algorithm (CGA) [15] is inspired by the Simple Genetic Algorithm, but it does not simulate directly any biological process. It is equivalent to a SGA with uniform crossing-over and mutations. CGA changes the population representation. It makes this algorithm more efficient in term of resources and memory requirements. The population of the

Compact Genetic Algorithm is no more represented as a set of individuals but as a Probabilistic Vector (PV) whose length is the same of the individual chromosomes. The assumption behind CGA is that a large population can well be modeled as a Probabilistic Vector with less memory requirements, it allows to represent in a compact form individuals described by chromosomes with binary genes. The Algorithm 2 shows how CGA works. At the beginning all the values in the PV are set to 0.5. This allows to generate the first two individuals randomly. At every iteration a series of random number is generated, one for each gene of the genotype. The genes of a chromosomes are set to 1 if the corresponding value in the PV is greater than the generated random number, otherwise they are set 0. At the first iteration all the genes of all the chromosomes have 50% chance of being one or zero. For each iteration two individuals are generated and evaluated to determine a winner, with an higher fitness value.

Every generation consists in one binary tournament between two individuals. After that, the population is updated according to the alleles present in the winner chromosome. The aim of the update is to make more likely the generations of individuals whose alleles are more similar to those of the winner chromosome and less to those of the loser one. It is done incrementing or decrementing the elements of the Probabilistic Vector with a step of $1/n$, where n is often called *population size*, because it determines the number of possible levels that a value can assume.

Termination conditions are different from those of SGA and MGA. It is always possible to keep the first termination condition, ending the execution when an individual with sufficient fitness is generated, or to halt the execution after a predefined number of iterations, but the CGA terminates the execution also when all the values in the Probabilistic Vector converged to zero or one. This last condition is equivalent to the third one introduced for the SGA, but in the CGA, when that condition is filled, the Probabilistic

Algorithm 2 Compact Genetic Algorithm

```
1: for i = 1 to ChromosomeSize do
2:   P(i) ← 0.5
3: end for
4: repeat
5:   a ← NewIndividual(P)
6:   b ← NewIndividual(P)
7:   winner, loser ← Evaluate(a, b)
8:   for i = 1 to ChromosomeSize do
9:     if winner[i] < loser[i] then
10:      P(i) ← max(0, P(i) − 1/n)
11:    end if
12:    if winner[i] > loser[i] then
13:      P(i) ← min(1, P(i) + 1/n)
14:    end if
15:  end for
16: until Termination(P)
```

Vector itself represents the chromosome of the best individual.

2.1.4 Extended Compact Genetic Algorithm

In the previous section it has been presented the Compact Genetic Algorithm. One of the limits of that algorithm is that it considers all the genes independently one to each other. Therefore, it does not consider potential linkage between different genes. Since it has been demonstrated that linkage learning in GAs improves their search capabilities [31], it may allow algorithms that implement it to increase their convergence rate. Therefore in literature it has been proposed an improved version of the CGA that includes linkage learning. It is called Extended Compact Genetic Algorithm (ECGA) [32]. It exploits the same probabilistic approach presented in CGA, but ECGA introduces linkage learning building more complex probabilistic models. A class of probabilistic models that can be used is the class of Marginal Product Models (MPMs). They are the product of marginal distributions on a partition of the genes. They are similar to the probabilistic model used by the CGA, but they consider at the same time the probability distribution over more than one gene. Authors selected MPMs because they have two useful characteristics:

- They are relatively simple.
- Their structure can be directly translated into a *Linkage Map* which defines precisely probabilistic relationship between different genes.

Different models can be compared using *Combined Complexity Criterion*. It combines the Minimum Description Length (MDL) criterion and the complexity of the population represented, which is derived from the entropy of the marginal distribution.

The Algorithm 3 shows the ECGA overall structure. As in the CGA the individuals are not maintained permanently in memory but they are gen-

Algorithm 3 Extended Compact Genetic Algorithm

1: $t \leftarrow 0$
2: $P(t) \leftarrow NewRandomPopulation()$
3: **repeat**
4: $\quad t \leftarrow t + 1$
5: $\quad Evaluation(P(t))$
6: $\quad P(t) \leftarrow Selection(P(t-1))$
7: $\quad MPM \leftarrow GreedyMPMsearch(P(t))$
8: $\quad P(t) \leftarrow NewPopulation(MPM)$
9: **until** $Termination(P)$

erated at every iteration from a probabilistic MPM. All the individuals are evaluated with the usual methods, as well selection is done. individuals selection and evaluation is accomplished using traditional methods. The most complex operator used by the ECGA is the constructor of MPMs. It is done with a greedy search. The magnitude of the improvement achievable using ECGA, with respect a CGA, depends on the number of subproblems that is possible to identify in the task.

2.1.5 Other Genetic Algorithms

The largest part of the developed Genetic Algorithms can be classified in the four families described above. Search capabilities and performance of GAs can be improved with *competent* implementations, which take care of the peculiarity of the problem addressed, hybrid implementations or parallelization. Some additional family of GAs can be identified:

- Multi-Objective Genetic Algorithms. They do not consider a single fitness value for an individual but an objective vector. Such modification over fitness evaluation impacts on the selection operation. It is still possible to select those individual with an overall score higher

than others but other ad-hoc approaches have been introduced. It is possible to select individuals that *dominates* other, in the sense that they have an higher score for all the objectives (NPGA) [33], or to rank individuals following a *domination* relationship and assign them a fitness value according to it (NSGA) [34]. It is also possible to introduce elitism (CNSGA) [35].

- Co-operative Co-evolutionary Genetic Algorithms can be a further adaptation of GAs for Multi-Objective search [36]. Their basic principle is to divide the population into a set of sub-populations, each converging toward a different objective. Periodically, sub-populations are combined to guarantee convergence toward a global optimum.

2.2 Hardware-based Genetic Algorithms

In this section a series of hardware Genetic Algorithms implemented on Filed Programmable Gate Arrays (FPGAs) will be presented. Implementing Genetic Algorithm in hardware allows to better exploit parallelism. It can be done pipelining Genetic Operators or implementing multiple hardware modules that work concurrently.

Initially, two architectures based on Simple Genetic Algorithm will be presented. The first is a preliminary implementation of a hardware Genetic Algorithm, the latter introduces some improvements to make it more suitable for real scenarios. After that an hardware Compact Genetic Algorithm will be introduced. Its structure has been also used in a cooperative multi-population version that will be presented later.

2.2.1 First Hardware-based Genetic Algorithms

The first Hardware-based Genetic Algorithm (HGA) [1] has been proposed in 1995. The authors argued that if a Genetic Algorithm runs with

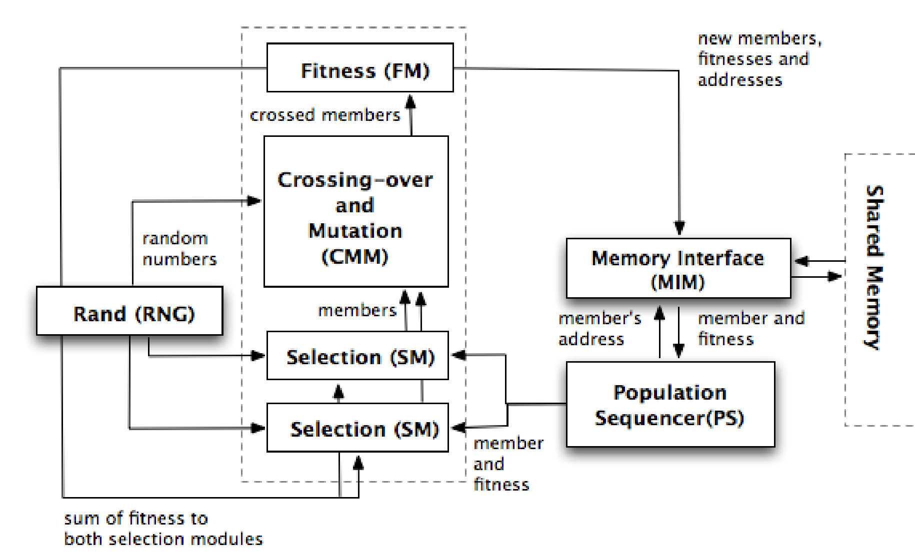


Figure 2.1: First Hardware Genetic Algorithm. [1]

a population of m individuals and takes g generations to converge to the optimal result it will repeat mg times the same set of operations. Considering that 100 is a suitable size for the population and 10^4 or 10^5 are realistic numbers for the generation, parallelization, pipelining and hardware accelerations could provide useful improvements. For these reasons the authors proposed a modular hardware implementation of the most Simple Genetic Algorithm (SGA) proposed by Goldberg [29] and based on selection, mutation and crossing-over.

Figure 2.1 shows a black-box schematic of the Hardware-based Genetic Algorithm implemented with a parallelization of the selection operator. The designed architecture operates the following seven steps:

1. All the parameters that characterize the algorithm are loaded to the *Memory Interface Module* (MIM). It acts as the control unit of the hardware GA and it is the only interface to the external environment.

2. The MIM initializes all the other modules: the *Fitness Module* (FM), the *CrossoverMutation Module* (CMM), the *Pseudo Random Number Genera-*

tor (RNG) and the *Population Sequencer* (PS).

3. The PS starts the execution requesting the individuals in the current population to the MIM and passing them to the selection module.

4. The SM receives new individuals and evaluates them until a pair of them with a sufficient fitness level is found. Later, the SM passes them to the CMM and resets itself restarting the selection process.

5. The CMM module receives a couple of individuals from SM, following random numbers generated from the RNG it applies crossover and mutations. Once the operation is completed the new individuals are sent to the Fitness Module.

6. The FM evaluates the two new individuals and it writes them to the memory through the MIM. The Fitness Module also notifies to the MIM when a sufficient fitness is reached to end the execution of the HGA.

7. The previous steps are repeated until the execution is completed.

From the above description of the steps and of how the modules work, it is clear how the proposed HGA architecture implements the pipelining, stage after stage. Such architecture has been deployed to a multi-FPGAs solution in order to be evaluated, while a software version of the same algorithm was executed on a MIPS architecture to be compared to.

Table 2.1 shows results obtained with such architecture. With respect to a software implementation there is a substantial reduction in the number of clock cycles and an execution time speed-up between 15x and 17x. Moreover the authors claimed that the use of more complex fitness function will increase the speed-up of the hardware implementation. As it was already highlighted, fitness evaluation is an expensive operation that hardware can do much better than software. Presenting their hardware Genetic

Table 2.1: Hardware-based Genetic Algorithm performance

Fit. Fun. F(x)	N. Gens	SGA clk cycles	HGA clk cycles	Speedup
x	10	97064	5636	17.22x
x	20	168034	10622	15.81x
x+5	10	99825	5585	17.87x
x+5	20	170279	10945	15.58x
x^2	10	334210	22892	14.59x
x^2	20	574046	45019	12.75x

Algorithm, a last topic analyzed by the authors is how it scales with respect to the complexity of the faced problems. The area requirements grow according to some parameters as: the size of the population (m), the size of the chromosome (n) and the width of the fitness (fw). A limit of this architecture is that the number of the communication pins between the different modules grows linearly following the chromosome length. This problem can be partially overcome introducing time multiplexing and memory elements. This is strictly constrained by the availability of memory onboard. The module that most suffers from scalability issue is the selection module. Following their estimations, authors reported that the number of Configurable Logic Blocks required increase according to Equation 2.1.

$$NofCLBs = (fw + \log(m))^2 \tag{2.1}$$

Recently, a new implementation of the same algorithm has been proposed [2]. The authors, of this new architecture, address also the problem of efficiency of area utilization. Their aim was to propose an hardware Genetic Algorithm with reduced hardware requirements. Also this new architecture is designed following a modular approach, but there are two important differences from the previous one:

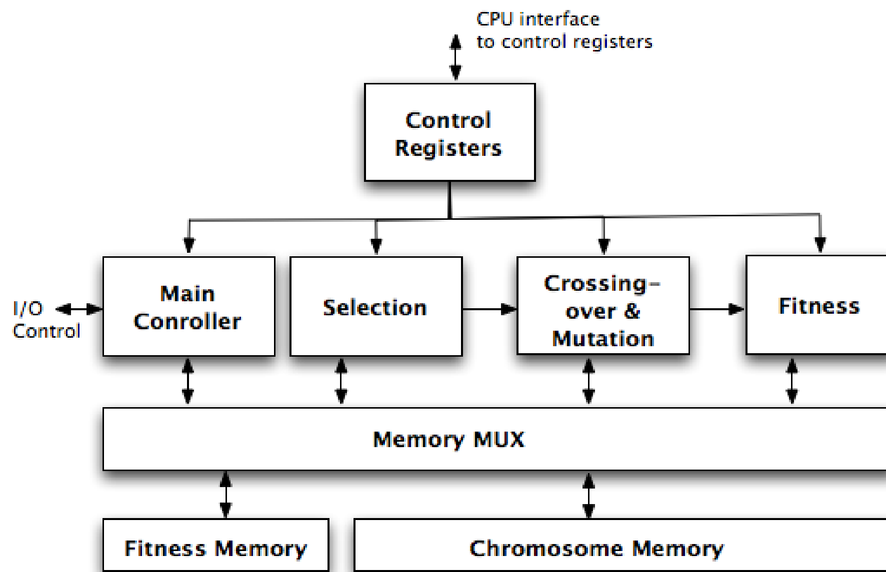


Figure 2.2: Another Hardware-based Genetic Algorithm [2].

- The control module is decoupled from the memory;
- All the modules have access to the same memory.

The memory is no longer a functional module as the others, but it is an external resource that every module can access. Figure 2.2 shows the structure of the proposed architecture. As it is possible to see, there are still signals between the different modules but they do not need any longer to pass the whole chromosome each time, they just need to pass the memory address of the considered element from one stage to the next one. Such implementation is much more area efficient although it suffers more the risk to be bottlenecked by the memory access. Another issue to be considered is the higher complexity of the module that manages memory access and of the module that serves as main controller.

To validate their approach the authors presented results obtained in circuit partitioning. Authors shown that the hardware GA approach allows to be between 10 and 100 times faster than software solutions, compar-

ing the performance obtained on FPGA at 50mhz and software executed on SUN ULTRA10 440 MHz processor system. Also area requirements are contained. This hardware-based Genetic Algorithm requires 167 CLBs to be implemented on a Virtex XCV2000E, with a population of 20 individuals with 8 bits of chromosome length.

2.2.2 Hardware-based Compact Genetic Algorithms

After the first hardware implementations of the *Simple Genetic Algorithm* some researches argued that such algorithm was not suitable for an hardware implementation. When the population size or the chromosome length grow, the amount of memory required to store the population dramatically increases. Population requires wide memories to be stored in and it limits the exploitable parallelism. The memory bottleneck is inevitable. In 2001 two researchers proposed a first hardware implementation [37] of the *compact Genetic Algorithm* [15] described in section 2.1.3. In this GA the population is no more represented in memory as a set of individuals but with a more compact Probabilistic Vector, this influences also the hardware implementation. Moreover the algorithm operates in parallel over all the elements in the Probabilistic Vector by executing a series of simple operations. Figure 2.3 shows one block of the proposed architecture. The number of the blocks depends on the length of the chromosome and each block operates on just one gene. Each block is composed of few simple hardware components, a Random Number Generator (RNG), a Probability module (PRB) a comparator (CMP) and a buffer (BUF). Besides an array of these blocks, the architecture implements also other three modules: two Fitness Evaluators and a comparator. RNGs have been implemented with Linear Feedback Shift Registers (LFSRs) technology [38] to generate pseudo-noise bits. The PRB is composed of a memory and an adder-subtractor to increment or decrement the value. CMPs and BUFs are standard elements available on

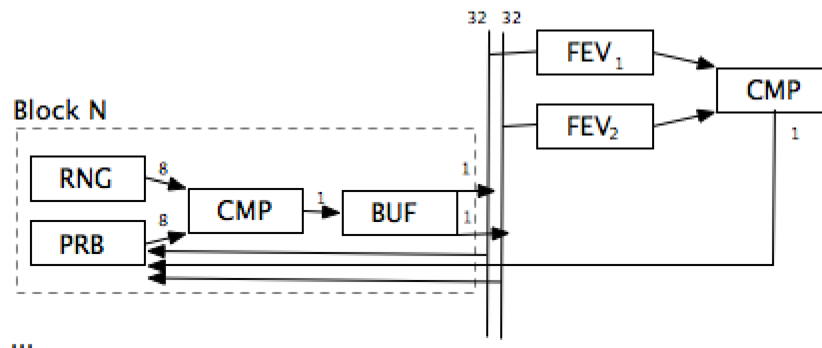


Figure 2.3: One block of the Hardware Compact Genetic Algorithm

almost every FPGA. FEVs need to be customized following the task to be considered.

The architecture operates the following steps:

1. At the startup a probability value of 0.5 is used to initialize the PRB;

2. RNG generates the first random number;

3. The CMP compares the value from the RNG with that one in PRB and returns as output the result of such comparison;

4. The output of CMP is stored in the BUF;

5. The RNG generates a second random number. As the first one it is compared with the Probabilistic Vector;

6. The value stored in the BUF and the result of the last comparison become two genes, one belonging to the chromosome of the first individual, one belonging to the chromosome of the second one. Their relative position inside the chromosome depends on the index of the block considered;

7. The FEV modules compute the fitness of these two individuals;

8. The CMP module, that gathers the output of the FEVs, determines the winner individual;

9. The value of the PRB is updated. The PRB receives two input a the from the winner chromosome and one from the looser. The probabilistic value stored slowly converge toward 1 if the input from the winner chromosome is 1, toward 0 else;

10. All the previous operatios are repeated until the FEV modules do not find a chromosome with a sufficient fitness.

The authors claim that this extremely parallel implementation is able to complete a whole generation in just three clock cycles when the fitness evaluation consists in counting the number of ones in the chromosome (one-max task). Operating with chromosome of 32 bits length and random number with 8 bits of precision this architecture requires 813 slices on an old Virtex V1000FG680. It shows a speedup of 1000x respect to a software implementation that runs on a 200Mhz Ultra Sparc 2. In literature it has been proposed a *linkage learning version* of this algorithm to enhance the problem solving capability, that as been described in section 2.1.4 of this chapter. However the linkage learning used in ECGA [32] is not suitable for an hardware implementation, so the authors decided to try to implement other improvements to extend their research. The two proposed improvements are the Cooperative Compact Genetic Algorithm and the Cellular Compact Genetic Algorithm [39]. They take inspiration from techniques adopted in parallel GAs [40] and cooperative approaches [36].

The Co-operative Compact Genetic Algorithm (CoCGA) introduces the cooperative coevolutionary concept in the compact Genetic Algorithm. In the CoCGA there is a series of CGA cells, each one able to manage a local population and to communicate with its neighbours. Special cells are called *group leaders*: they manage the communications between the CGA

cells. CGA cells are implemented as described above, except for the introduction of a *confidence counter*. The following steps characterized their behaviour:

1. Two individuals are generated. As in normal CGAs.
2. They are compared and evaluated.
3. The probability vector is updated and the confidence counter is incremented according to the fitness value.
4. The probability vector (p) and the confidence counter (cc) are sent to the group leader.
5. The execution of the previous step is repeated until termination conditions are not reached. Usually, until an individual with a sufficient fitness is generated.

While the group leader executes different operations:

1. It checks the confidence counter of each neighbour cell and selects the highest one.
2. It updates its own probability vector according to the valus in the probability vector from the cell with the highest confidence counter.
3. It updates the probability vector of all the neighbour cells with the new probability vector of the group leader.
4. It repeats such operations until the execution is completed.

The topology of the cell can be different, they can be arranged in arrays or grids. Later the same authors proposed also a Cellular Compact Genetic Algorithm. The theoretical approach is similar to that of the CoCGA, but the implementation is different and some modifications have been introduced. First of all, there is no more distinction between group leaders and

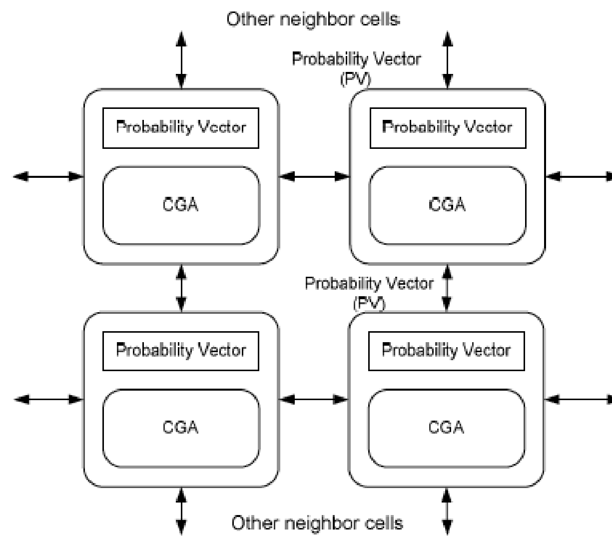


Figure 2.4: Topology of the Cellular CGA

cells. All the cells are equal and embed also the functionalities to manage the communication and the *migration* of the population. Having no more group leaders, the probability vectors are passed directly from a cell to its neighbours. In the same ways also the confidence is computed and passed to neighbour cells. Once received external probability vectors from neighbours, the cell itself selects the one with the highest confidence counter and updates its own probability vector by adaptive combination of the two. The highest confidence counter underlines the highest probability to reach the best solution. Migration of probability vectors is asynchronous. Different cells have different update rate of their confidence counter, this leads to different rate of migration. Figure 2.4 shows the topology of the Cellular Compact Genetic Algorithm. It is possible to see the grid of homogeneous cells and the arrows that show the connections between neighbour cells.

Such algorithm decreases the risk of stalling in local maxima with respect to the normal CGA because it introduces multiple populations that slowly converge towards the optimal solution. A scenario in which all the populations stall is unlikely. In the proposed implementation, the confi-

Table 2.2: CGA, CoCGA and CCGA performance

	OneMax	F1	F2	Speed-up OneMax	Speed-up F1	Speed-up F2
CGA	43362	126967	80027	1x	1x	1x
CoCGA	11492	25542	27757	3.77x	4.97x	2.88x
CCGA	12321	28853	26591	3.51x	4.44x	3.0x

dence counter of a population is realized as a 5 bits counter that is increased every time that a candidate individual has a fitness that is higher than the best fitness previously achieved by that cell. Confidence counter width should depend on the number of possible fitness levels available.

Table 2.2 shows the results obtained with the above implementations. The three tests considered are one-max and two functions, F1 and F2. F1 is a non-continuous function while F2 is a simple equation. The speed-ups are significant but the area requirements of such implementations grow considerably. The CGA requires 1062 Look-Up-Tables on a Xilinx Virtex LX50, while the CCGA requires 1932 LUTs for just one cell on the same board. A 2x2 grid,that has been used for the results in table 2.2, requires 5500 LUTs. As highlighted by the authors a good feature is also the possibility to scale the architecture depending on the complexity of the problem to be addressed. Both the cellular and the cooperative version allow to decide an arbitrary number of cells to be implemented. If this allows to deal with scalability issues of the problem it rises a scalability issue about the implementation.

When the size of the chromosome grows, or multiple cells are implemented, the algorithm shows two limits:

- The integrated communication unit may constitute a bottleneck. To evaluate, monitor and transfer all the Probabilistic Vectors of all the cells may become a critical activity with a large size Probabilistic Vector.

It will require heavy time multiplexing due to the impossibility to have a large bus.

- At every iteration, all the cells in the grid need to test a pair of individuals concurrently. This may become extremely expensive in terms of area if the testing module has to perform complicated tests. To run more complex tests may require to have a complex testing module with large area requirements. The cellular approach needs two testing modules for each cell and multiple cells. If testing module is large, the total amount of area required may become excessive.

2.3 Conclusions

In this chapter a series of evolvable algorithms have been presented, they are the base for the largest part of modern researches in the GA field. Proposed works that exploit Genetic Algorithms often consider the algorithms that have been described in this chapter and try to improve them to solve some specific problems. Multiple optimization problems have been solved thanks to Genetic Algorithms, adapting or accelerating them. The best performance is often achieved by tailoring a Genetic Algorithm on the characteristics of the problem to be addressed. To do that requires a deep understanding of the specific problem features, for this reason the next chapter will focus on the specific task of Evolvable Hardware. Knowing characteristics of the target problem is fundamental to be able to implement efficiently the GAs.

Chapter 3

FPGA-Based Evolvable Architectures

In this Chapter some recent works on Filed Programmable Gate Array-based Evolvable Hardware systems will be discussed, with particular focus on how the solutions proposed in literature deal with the three issues presented in section 1.3.4: *scalability*, *fitness evaluation* and *observability*. First the most relevant Extrinsic evolvable systems will be introduced. In the second section the Intrinsic approaches will be presented. Different techniques have been proposed according to the different characteristics of the hardware devices. In particular, *Virtual Reconfigurable Architectures* and *Bitstream manipulation techniques* will be presented. The third section describes a more efficient architecture implemented with a modern Xilinx Virtex 4. It exploits bitstream manipulation and dynamic reconfiguration, opening the way to the implementation of *Complete* or *Open-Ended* architectures. Such system is that one considered for this thesis, it will be used both for Extrinsic and Complete evolution.

3.1 Extrinsic Approaches

Extrinsic Evolution of hardware component consists in evolution done offline, simulating the behavior of the target device. There are two main extrinsic approaches: *Direct Methods* and *Indirect Methods*. In Direct Methods the chromosome bits represent directly the architecture encoding, while in indirect method they do not. In this latter case, the architecture configuration does not evolve directly, but the Evolutionary Algorithm operates just on an abstraction of the device. It may represent just connections between predefined components, or configurable modules, that can implement a reduced set of functionalities, but independently from the device structure.
A chip for the six-multiplexor problem [41] has been evolved using direct methods.
The usage of a Direct Methods requires to implement a software simulator of the architecture. It requires a deep knowledge of the architectural structure of the target device. Such simulator has been used to evaluate the behaviour of a 108 bits chromosome, where 12 bits were used to define the functionality of a logic cell and the others 96 to define its connections. The circuit has been evolved using a Simple Genetic Algorithm (SGA) with the basic operations of selection, mutation and crossing-over. Having a population of 100 individuals and testing each of them over all the possible 64 combinations of inputs. In such conditions, a performance of 100% has been reached after 2000 generations. The same authors have validated their approach also evolving a *XOR* and a 3 bits accumulator, with similar performance [42] [43]. However, to implement just a slightly larger component with such approach would have led to a wider search space, so new more complex solutions have been explored. Since the first researches in EHW field, one question that rised was if it was better to focus on algorithm speed-up or rather to design a better algorithm, able to achieve evolution goals in less iterations. Indirect extrinsic implementations have been pro-

posed following this second idea [44]. Among the solutions proposed in literature to address this issue, it is possible to find approaches based on *variable length chromosome* [45], *functional level evolution* [46] [47], *decomposition and incremental resolution* [48].

The variable length chromosome approach has been introduced to handle the scalability problem by using the most compact representation of the chromosome. This approach consists in changing the chromosome length at runtime. It is done by dynamically associating some features of the individual to just one gene of the chromosome, including in the genotype only those features that are identified as bringing a relevant contribution to the circuit functioning.

Such method has shown good results in image recognition [49], so it has been applied also for hardware evolution. Theoretically this approach could be applicable also to an intrinsic evolution, because it does not put requirements on the evaluation and neither it modifies the approach used for the fitness computation. Much more unlikely is to apply variable length chromosome to hardware-based systems. It is immediate to see that the dynamic properties of this structure match bad with the rigidity of an hardware implementation, which is based on registers of fixed length.

Another extrinsic approach is the functional level evolution [46]. This technique has shown good results [50] in the image recognition area. Such approach for EHW design allows to explore the behavior of the component that is going to be implemented. Moreover, the chromosome size is reduced thanks to high level components that reduce the search space size, allowing to handle scalability issue. The genes need only to define what task to perform in what block and the topology of the connections. Blocks usually implement only a reduced set of functionalities, so small chromosomes are enough to define their behavior.

Usually also the connections schema is simplified, with respect to di-

rect approaches, and just an high level representation is provided. To use an high functional level decomposition makes also the problem suitable for the application of *Cartesian Genetic Programming* technique, already described in section 1.1. On the other hand this solution leads to a reduction in the flexibility of the system, because it puts a lot of additional constraints on the implementable components.

Incremental Evolution aims to improve the possibility of the extrinsic evolutions by introducing the innovative element of functional decomposition and trying to achieve an higher flexibility. The principle of *Bidirectional Incremental Evolution* is "to divide a complex task into simpler sub-tasks, to evolve each of these sub-tasks and then merge incrementally the evolved sub-systems, reassembling a new evolved complex system" [48]. Partitioning the problem and evolving the single parts allows to reduce the global complexity of the system. Merging the evolved parts is a task with the same complexity of the functional evolution. However, improvements are achieved because also the subparts themselves are evolved. Such decomposition of the system must be possible and meaningful for the component that is evolving. In the methodology proposed by T. Kalganova a further degree of freedom has been achieved by introducing the possibility to have a dynamic number of functional elements in the design. The key elements considered in that approach are: how decomposition is done and how it is possible to evolve such modular system.

A first method of decomposition is output based. According to the Shannon's theorem [51], a multi-outputs function can be decomposed in a series of simpler functions, one for each output. Subsequently, the incremental evolution is obtained solving smaller problems first and more complicated later, adapting slowly the individual to them. Once the subparts tailored for solving the smaller tasks are identified, they are grouped into larger units that evolve to handle more complicated tasks. The approach has been

called bidirectional because the already identified groups can be splitted, when they do not lead to improvements of the fitness value. T. Kalganova obtained as result that her methodology allows to hit a performance of 100% after 150 000 generations evolving an EHW that implements a function with 7 inputs and 10 outputs. While by comparison, a simple direct evolutionary methodology obtains no more than 91% after the same number of generation, *stalling*. Probably also other evolutionary methodologies are able to obtain comparable results, but this methodology shows well what is the direction of the research in Extrinsic approaches. Incremental evolution is an other technique not suitable for Intrinsic architecture. Incremental evolution may require to explore intermediate results of the component and this is not always possible in real implementations, and when it is possible it may require to introduce in the evolvable architecture modules customized for this task.

3.2 Intrinsic Approaches

A first step forward, from the extrinsic evolutions, is to evaluate real implementations of the EHW candidates with the Intrinsic approaches. This choice allows to speed-up the evaluation phase, drastically decreasing the number of clock cycles necessary to compute the fitness of an individual. The Fitness Evaluation is generally a repetitive operation that hardware can do in parallel, over more individuals at the same time. On the other hand a new problem is introduced: how to deploy quickly at run-time new individuals. With the Extrinsic approach it was possible to evolve a component and then synthesize or implement it using the methodologies usually applied in hardware design. The usage of intrinsic methods requires to make the evolutionary strategy able to modify quickly the configuration of the hardware device that is evolving. How this can be done is heavily influ-

enced by the target device considered.

One of the first intrinsic EHW system has been proposed by Thompson in 1996 [52]. An evolvable core able to distinguish between a square wave with frequency 1 kHz from one with frequency 10 kHz was designed. The target device of this evolution was a portion of the Xilinx XC6216 FPGA. It was a square area of 10x10 CLBs. The Evolutionary Algorithm was running on an external workstation. Every time that a new individual had to be tested, a new configuration bitstream was created and the new configuration was deployed on the FPGA. Individuals were evaluated by providing inputs and monitoring the outputs. Thompson was able to achieve a working component after three weeks of evolution. Such evolvable architecture was strictly dependent on the target device selected, it was not possible to use it with different FPGAs, and it was susceptible to external conditions. However he demonstrated the effectiveness of the EHW approach to solve real tasks.

Thompson developed also a new system called *evolvatron* [53]. It was still based on Xilinx XC6216, but no more dependent on the specific device used and external factors like the temperature. Also other researchers proposed solutions based on this FPGA [54] [55].

Nowadays there are two main approaches develop Intrinsic Evolvable Hardware Systems: *Virtual Reconfigurable Architectures* and *Bitstream manipulation techniques*.

Virtual Reconfigurable Architectures [57] have already been introduced presenting the approaches based on functional decomposition. In that case the aim was to simulate the behavior of some hardware modules to evolve the component to be implemented on FPGA. In the intrinsic evolution case the purpose was slightly different. It is always possible to develop an architecture that implements high level functions, but now the aim was to have an architecture that allows a rapid deployment of the individual. Now the

deployed architecture itself is reconfigurable. An example could be a virtual FPGA with a rapidly accessible configuration memory. The cost of this approach is an higher area requirement for the deployment of the circuit, but it allows to achieve good time performance.

Bitstream manipulation consists in modifying directly the FPGA configuration. It requires to know how the FPGA is configured and how such configuration can be changed. Not all FPGAs are suitable for this approach and not all the parameters of the FPGA can be easily modified. A partial solution to this problem is to modify only those parameters that can be easily changed.

A first Virtual-Reconfiguration-Circuit system has been proposed in 1998 by P. Haddow and G. Tufte [58]. Later, the most recent Virtual Reconfigurable Architecture (VRA) for the intrinsic design of evolvable hardware it that one proposed by P. Ke, X. Nie and Z. Cao [26]. They implemented a VRA, described in Hardware Design Language, as a second layer of reconfiguration over an FPGA with the purpose to provide a genotype of reduced size and fast internal reconfiguration. It allows to deal with several of the previously identified issues: fitness computation can be hardware accelerated, chromosome size is reduced thanks to a more functional approach and deployment of individuals is fast. Such virtual architecture can be synthesized and deployed on a wide range of FPGAs due to its high level description in HDL. They selected the 2x2 multipler as case study and defined an optimal Virtual Reconfiguration Architecture to solve that specific problem. They confirmed the validity of their architecture by evolving such multiplier in 4928 generations of 20 individuals each. The authors claim to have in this ways a speed-up of 100 times with respect to an extrinsic implementation of the same architecture. On the other hand such implementation requires around 1000 slices for 20 individuals, which means 50 slices for each individual. Considering that a 2x2 multiplier is a function

with 4 inputs and 4 outputs that could be implemented with 4 Look-Up-Tables (LUTs) of a Virtex 2 FPGA, it appears clearly how inefficient were VRAs in terms of area, although they provide extremely high performance in terms of hardware speed-up and good algorithmic optimization possibilities.

Bitstream manipulation techniques allow to use a direct approach on FPGA. These approaches have been made possible with the modern FPGAs when *JBits* [59] has been released. It is a Java tool able to modify the specific content of a Configurable Logic Block or a Look-Up-Table without having to synthesize a new architecture. It overcomes the need to have a VRA by allowing to make quickly partial modifications to the architecture. It makes possible to use as evolvable components directly the FPGA CLBs and the communication lines among them. The same tool is also able to read back the configuration of the FPGA, allowing to explore the current configuration of the device. This new tool has been first used by two Xilinx researches that proposed an evolvable architecture called GeneticFPGA [60]. However the purpose of their work was no to create an efficient EHW component to address some real problem, but only to show the capability of an approach based on bitstream manipulation to build working individuals that can be implemented in a safe and stable way. They introduced the idea that is possible to map directly the content of a Look-Up-Table (LUT) on a series of genes. Each value of a LUT can be mapped into a single gene. It allows to define the chromosome as the content of a series of LUTs. In this way the chromosome can be directly implemented on FPGA, without having to pass through a process of hardware synthesis, placement and routing. Jbits takes as input parameters which property of the design is going to be modified and the new configuration, that is a chunk of the chromosome. Then it builds a partial bitstream applying modifications to the FPGA configuration, through the *SelectMAP* port [61].

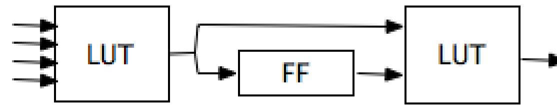


Figure 3.1: Tyrrell's evolvable cell

More recently some applications to robotics and implementations of logic circuits [56] have been proposed by Tyrrell and others. They proposed an important improvement that allowed to boost the performance reducing the size of the problem that has to be solved when evolving an individual. They introduced the idea to use partial reconfiguration provided by Jbits only to modify the content of the LUTs. Connections between LUTs can be statically defined or implemented with a customizable communication infrastructure. The main limit of this approach consists in the absence of memory elements available in the design. A further improvement of the same architecture consists in the introduction of a multiplexer, associated to each LUT. The multiplexer allows to decide if the next stage will be directly the connection to the output or a connection to a FF where the output value can be stored.

Figure 3.1 shows the idea besides the last version of the evolvable cell developed by Tyrrell. It uses two LUTs and one register. In this way he obtained a *sequential* component with one register. The final output depends on the current primary inputs and on the state-value memorized in the register. A more recent system that exploits Jbits has been proposed by C. Lambert, T Kalganova and E. Stomeo. [62].

As shows in Figure 3.2, their proposed system is divided in three different parts. There is one FPGA that represents the individual. A second FPGA on which is implemented the evolution strategy. A third element to manage the system, allowing to deploy on the first FPGA the bitstream generated in the evolution phase by the second FPGA. This approach works also in

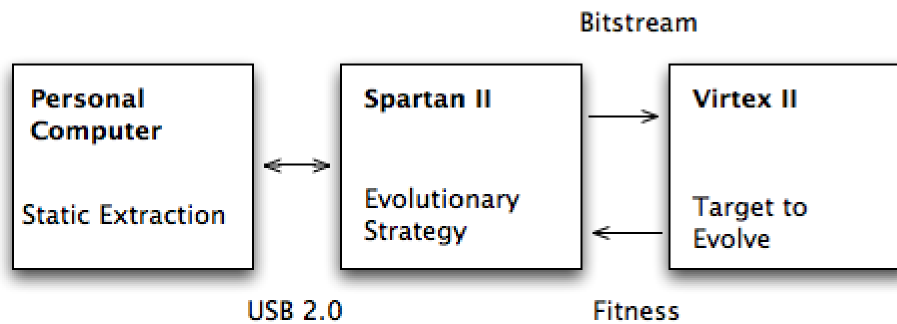


Figure 3.2: Multi-device FPGA-based Evolvable Hardware

the case in which it is not used just one FPGA for implementing an individual, but multiple FPGAs. In this way it is possible to implement at the same time multiple individuals making parallel evaluations. Common parts, as the hardware modules belonging to the Evolutionary Algorithm core and the bitstream manipulation core, are implemented in a first FPGA, then a series of FPGAs are connected to it. They implement EHW modules and their number depends on how many individuals it is necessary to evaluate concurrently.

Also according to this methodology, the functionalities required to reconfigure the target EHW are still provided by Jbits.

A methodology that avoid to use Jbits has been proposed by Upegui and Sanchez [63]. Not using such tool makes possible to work also with devices not supported by it. They proposed to use two additional methodologies for the partial reconfiguration of the FPGA, supported by Xilinx: *Module Based* and *Difference Based* [25]. The first allows to implement a modular design in which it is possible to change just one module. It requires to be able to generate the partial bitstreams of that specific module, so if theoretically it can be used, practically it is not suitable for a rapid deployment of individuals. The latter, called *Difference Based*, is more suitable for the task. It allows to generate partial bitstreams to make small modification

to the configuration of the FPGA, changing just some components as the configuration of a LUT.

This methodology needs however the usage of an external tool such as *FPGA editor* to generate a partial bitstream that modifies the FPGA configuration. That is a low level design tool, given the low level design of an architecture it allows to make local modification and to generate a *partial bitstream*. Such bitstream can be downloaded on the FPGA configuration memory to deploy the changes. The evolutionary strategy can generate a script that given the implementation files of the architecture can apply some small changes to the EHW modules. Then a partial bitstream is obtained much more quickly than synthesizing an architecture and can be deployed on the FPGA. To apply such methodology it is necessary to know exactly the position of what CLBs must be modified. They are the CLBs that implement EHW modules. Authors proposed to use arrays of *hard macros* to implement evolvable cells. The usage of hard macros allows to define a priori the placement coordinates and partially also the routing.

In bitstream manipulation techniques, the next step is to manipulate the bitstream directly without having to use external tools, but it requires a deep reverse engineering work due to the lack of official documentation. Direct bitstream manipulation techniques have three requirements:

- Knowledge about the bitstream format. It is necessary to know how to build a bitstream that implements a specific functionality.
- Capability to build up the bitstream that configures the FPGA to deploy an individual on an evolvable hardware module.
- Capability to do the previous operations quickly.

In the past years some methodologies [64] [65] to exploit also this approach have been proposed but some constraints remain. Researchers proposed to analyze and compare partial bitstreams obtained with FPGA editor to un-

derstand how they are built. That made possible to build partial bitstreams, to deploy on the board the evolvable individuals. An additional issue is rised by the fact that each FPGA model uses different bitstream formats. Bitstreams for different family FPGAs have different format, so a lot of effort in bitstream analysis is required every time that one changes device.

Moreover, an evolvable system that can be deployed on a certain FPGA-model cannot be deployed on another one without re-implementing the bitstreams generation phase, adapting it to the new device. Once understood the exact structure of a bitstream, it is neither trivial to exploit it. Generally bitstreams are not small enough to be stored on FPGA memories. This requires the availability of an external memory to store them. A third problem is how to deploy quickly bitstreams on the FPGA once they are successfully built. If the evolutionary strategy that builds bitstream is not on the same board that will implement EHW, they can be downloaded into the configuration memory through the SelectMAP port.

3.3 A Xilinx Virtex 4-based Evolvable Architecture

Xilinx Virtex 4 is one of the devices that allow the highest flexibility in EHW implementations. An EHW architecture that uses such FPGA has been developed at *Politecnico di Milano* in the last years [4]. Since it will be used also in this thesis work as object of the evolution, it will be described with more details. It follows the same principles used also in [66], where the target of the evolution is the content of the FPGA LUTs. Such evolution is done with bitstream manipulation techniques.

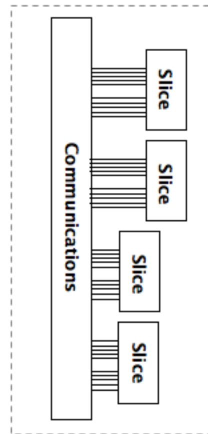


Figure 3.3: Virtex 4 CLB, with 4 slices

3.3.1 The Target Device

This system has been implemented on a Xilinx XC4VFX12 [67], belonging to the Virtex 4 family [68]. Its CLBs are composed of four *slices*. Every slice is composed of two LUTs (Look-Up-Tables) with four input nets. The top LUT is called LUT-G, the bottom LUT-F. Their functionality is determined by a 16 bits array stored into the configuration memory. Among the additional elements available inside a slice there are two Flip-Flops (FFs) and two Multiplexers (MUXs).

Figure 3.3 shows the structure of a CLB with four Slices, while figure 3.4 shows some details of the internal structure of a slice. In the CLB shown in the first figure it is possible to see also an element on the left. It is the multiplexer that manages the local routing lines inside the CLB and the inputs of the different slices.

Through modifications of the content of the configuration memory, it is possible to mutate the behaviour of the slices. It makes this devices useful for the implementation of EHW components. CLBs are not the only elements available on a Virtex 4, there are also Input-Output Blocks (IOBs), Block Random Access Memorys (BRAMs), Flip-Flops (FFs), and other em-

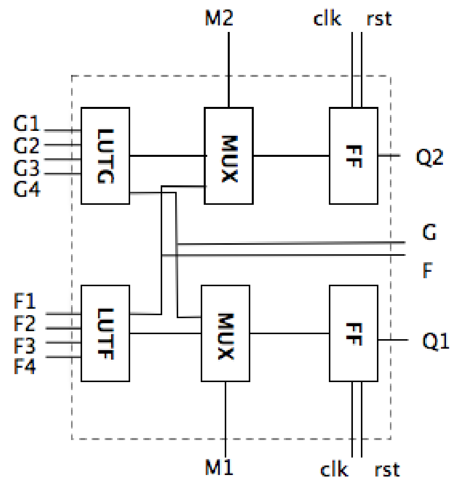


Figure 3.4: Internal structure of a slice, some details

bedded devices such as an integrated PowerPC (ppc405). An important element embedded in the Virtex 4 FPGA is the Internal Configuration Access Port (ICAP) [69] . It allows a self-adaptive behavior, providing to the implemented architecture access to the configuration memory.

To create an efficient System-on-Chip EHW System, that is the purpose of this thesis, it is required to be able to deploy the individuals at runtime through the ICAP. For this reason in the next section a detailed description of the Virtex 4 bitstreams manipulation is provided.

3.3.2 Virtex 4 Bitstream Manipulation

Among the different techniques available for bitstream manipulation that has been proposed in literature, it has been used the one firstly proposed by Upegui and Sanchez in [63] and that has been also applied to the Virtex 4 device [64]. This is based on a direct manipulation of the bitstream without using external Xilinx tools. That makes such approach suitable to design an adaptable architecture for single-chip implementation. In order to avoid damages to the device due to random modifications of the configu-

ration memory of the FPGA and to simplify the bitstream analysis process, the idea is to modify only the content of the LUT on the FPGA. The choice of what connections will be used is done before the implementation and it is not subject to runtime modifications. This approach allows to partially overcome routing limits highlighted in literature [62], due to lack of public available documentation. In this way the definition of such communication infrastructure between the evolvable LUTs becomes a fundamental characterization of the EHW architecture design. Due to the characteristics of the Virtex 4 device, that allows a two dimensional reconfiguration, it is possibile to relax some area constraints typical of older FPGAs [70]. It allows to build a parallel architecture that, on the same device, manages at the same time multiple evolvable individuals. That was not possibile for Upegui and Sanchez due to the reconfiguration limit of their Virtex 2 FPGA.

Bitstream manipulation has been already introduced in section **??** and it has been said that the bitstream format depends on the target device. Now it will be described how to build a partial bitstream that modifies the content of the FPGA LUTs, to deploy the individuals on a Xilinx Virtex 4. First, since the implemented EHW system needs to change the configuration of one or more LUTs, it is necessary to know how to retrieve the *Frame Address* from the slice coordinates. The Frame Address is used to specify the portion of the FPGA area that *frame* is going to modify. When writing a frame a bitstream needs to change to write first the frame address into the *Frame Address Register* (FAR). The FAR is a 32 bits register divided into 6 fields, as 3.1 shows.

It must be written at the beginning, but while writing large contiguous frame it is update automatically. The values to be written into FAR are linked to the slice coordinates. The equations that state that relation have been obtained through bitstream analysis and now they can be found in literature [4].

Table 3.1: Frame Address Register structure

Field	index
Top/Bottom	22
Block Type	19 to 21
Row Address	14 to 18
Mayor Address	6 to 13
Minor Address	0 to 5
unused	23 to 31
Total	32 Bits

Through bitstream analysis [4], it has been derived that these parameters are linked to the slice coordinates by the following equations:

$$MayorAddress = Mayor(X) = \left\lfloor \frac{x}{2} \right\rfloor + Adj(X) \tag{3.1}$$

$$Adj(x) = \begin{cases} 1 & if 0 \leqslant X \leqslant 23 \\ 3 & if 24 \leqslant X \leqslant 31 \\ 4 & if 32 \leqslant X \leqslant 47 \end{cases} \tag{3.2}$$

$$MinorAddress = Minor(x) = \begin{cases} 21 & if X is even \\ 19 & if X is odd \end{cases} \tag{3.3}$$

$$RowAddress = Row(Y) = \begin{cases} 1 & if 96 \leqslant Y \leqslant 127 or \\ & if 0 \leqslant Y \leqslant 31 \\ 4 & if 32 \leqslant Y \leqslant 95 \end{cases} \tag{3.4}$$

$$Top/Bottom = Top/Bottom(Y) = \begin{cases} 1 & if 0 \leqslant Y \leqslant 63 \\ 0 & if 64 \leqslant Y \leqslant 127 \end{cases} \tag{3.5}$$

Then, after having specified the frame address it is necessary to write into the Frame Data Register Input (FDRI) the configuration of the resources within the frame. The LUT-F and LUT-G configurations bytes have not a

fixed starting point in the bitstream data flow, but their position depends on X and Y coordinates of the slice which they belongs to [4]. The following set of equations states the starting position, in the case in which the Top/Bottom value is 1, otherwise equations are slightly different.

$$\mathrm{TopByteStart}(Y, LUT) = 5(Y\%32) + \mathrm{Adj}_{LUT}(LUT) + \mathrm{Adj}_{Y}(Y) \tag{3.6}$$

$$\mathrm{Adj}_{LUT}(LUT) = \begin{cases} 0 & \mathrm{if} LUT = LUTF \\ 2 & \mathrm{if} LUT = LUTG \end{cases} \tag{3.7}$$

$$\mathrm{Adj}_{Y}(Y) = \begin{cases} 4 & \mathrm{if} Y\%32 > 15 \\ 0 & \mathrm{if} Y\%32 \leqslant 15 \end{cases} \tag{3.8}$$

$$\mathrm{ByteStart}(Y, LUT) = \begin{cases} \mathrm{TopByteStart}(Y, LUT) & \mathrm{if} Y\%32 \geqslant 64 \\ 162 - \mathrm{ByteStart}(Y, LUT) - \mathrm{Adj}_{G}(LUT) & \mathrm{if} Y\%32 \leqslant 63 \end{cases} \tag{3.9}$$

$$\mathrm{Adj}_{G}(LUT) = \begin{cases} 0 & \mathrm{if} LUT = LUTF \\ 1 & \mathrm{if} LUT = LUTG \end{cases} \tag{3.10}$$

Figure 3.5 shows the complete bitstream dataflow. It is composed of:

1. Comment data. They are informations like bitstream filename, date and FPGA model.
2. Two synchronization words. They are *0xFFFFFFFF* and *0xAA995566*.
3. Init. data. A series of packets that is sent to begin the reconfiguration process and set registers like FAR.
4. Conf. Words. They are contained into a large packet with the configuration information that are written into FDRI to set LUTs content. Such packet has also a checksum as central word.

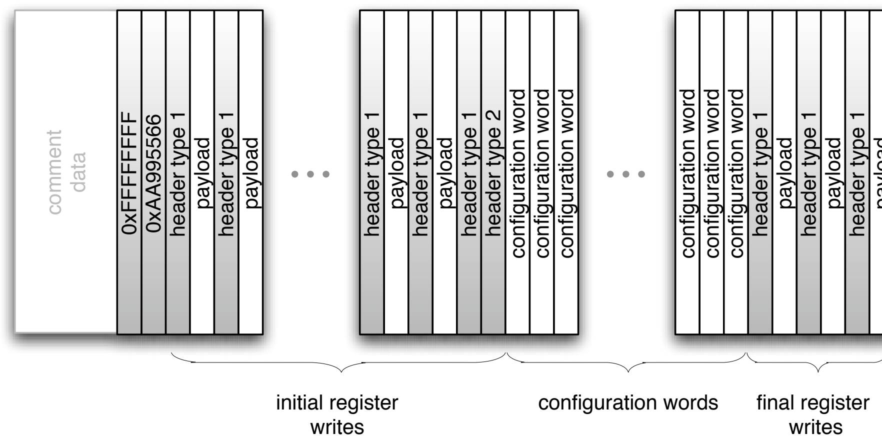


Figure 3.5: Bitstream data flow. [3]

5. Final writes. Finalization packages are sent to complete the execution.

The following algorithm shows how the checksum central word is built.

```
1  def computeWord(Y, LUT, BitPos):
2          CheckWord = 704 + BitStart(Y,LUT)
3          #Check if it is necessary to add an offset
4          if Y>63 and Y%32>7:
5          CheckWord += 32
6          if Y<=63 and Y%32<=23:
7          CheckWord -= 32
8          #Computes the odd parity bit
9          if len(filter(lambda x: x == '1',bin(CheckWord))) % 2 == 1:
10         CheckWord += 0x800
11         return CheckWord
```

Bitstreams can be sent both to the SelectMAP port and ICAP port, to do in the first case external reconfiguration, while in the latter an internal reconfiguration of the device. In the prospective to create a System-on-Chip (SoC) architecture with an intrinsic evaluation of the individuals, it is necessary to be able to internally reconfigure the portion of FPGA that implements the EHW modules.

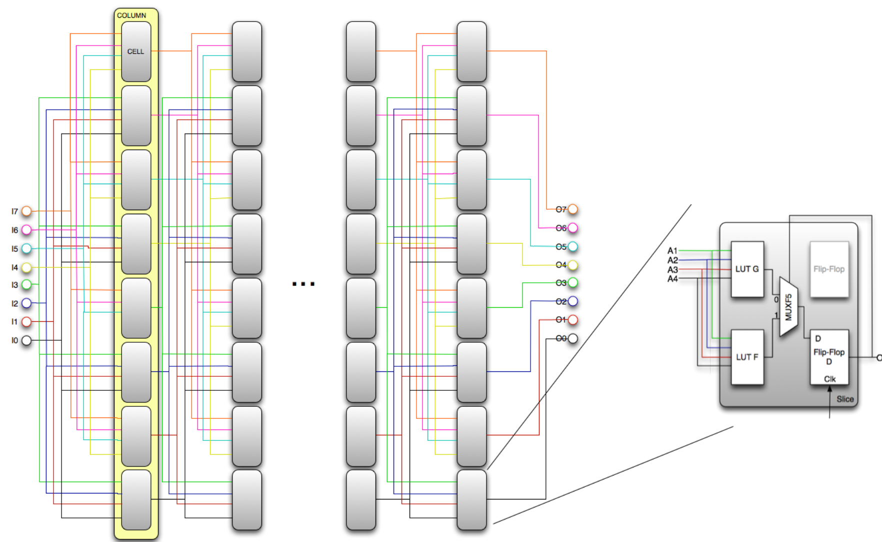


Figure 3.6: Evolvable region, structure of an individual. [4]

3.3.3 Evolvable Region Design

The target EHW individual considered has an 8 bit data path, with 8 bits of input and 8 bits of output. It is composed of 32 smaller evolvable cells, each with five inputs and one output. The five inputs of such cells are 4 external inputs plus one bit for the actual state, a loop back from the output value. They can be primary inputs or nets from other cells outputs. The cell is designed such that the resources which it needs to be implemented can be found into a single slice. It needs just a LUT-F, a LUT-G and a Multiplexer (MUX). In this way it is possible to allocate four cells for each CLB. In a cell, the two LUTs receive the same 4 inputs, while the MUX is driven by the additional value from the state register and determines LUT output signal will be forwarded to the Flip-Flop. Further detailed informations on the FPGA structure can be found into the Virtex 4 user guide [71]. The behaviour of a single cell is defined by 32 bits: 16 for the LUT-F and 16 for the LUT-G. This mapping between the cells of an individual and the resources of the FPGA allows to minimize reconfiguration time.

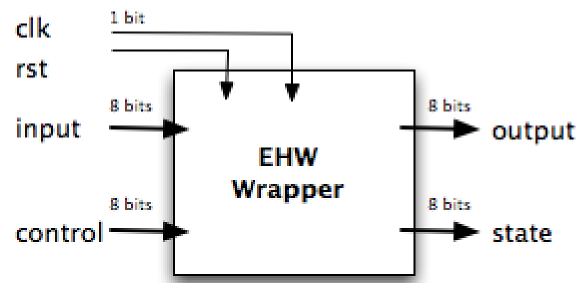


Figure 3.7: Evolvable region, individual interface

Figure 3.6 shows the complete structure of the 8 bits data-path EHW individual, on the left, and of the basic cell, on the right. The loopback of the output value has been introduced to avoid stability problem. As the right part of the Figure shows the 8 bits data-path is obtained with a composition of two columns of 8 4-bits-cells each. Connections between different cells are statically defined at design time and are not subject to evolution.

The solution adopted is to connect the odd cells of a column with the 4 less significant bits that come from the previous column (or from the primary input) and the even cells to the 4 most significant ones. The proposed element needs in this way two columns of eight bits each. One cell is defined by 32 bits, there are 32 cells over the two columns and the whole EHW individual is described by a 1024 bits chromosome. The static communication architecture has been implemented through an FPGA editor script, realizing an Hardware Macro [72].

3.3.4 Individuals Interface

In order to be able to use the EHW individual, it has been realized a wrapper module with a synchronous Finite State Automata (FSA) designed in VHDL. This module manages input signals and gather outputs when they are ready allowing communication with the evolvable hardware component that it wraps.

Table 3.2: EHW communication interface

	Name	Value	Description
Control	RESET	0x02	Prepare for a new exection.
	START	0x01	Start to process the input
	EMPTY	0x00	Empty word
State	WAIT	0x02	System waiting for data
	RUNNING	0x02	System processing data
	COMPLETE	0x03	Execution completed

As Figure 3.7 shows there are two inputs and two outputs besides clock (clk) and reset (rst) signals. The module receives 8 bits words as input and returns 8 bits results as output. The two additional registers, control and state, manage the execution. Through the control signal, input commands are provided, while from the state register it is possibile to read information on the state of the execution. Table 3.2 provides more detailed information.

Between two different commands it is necessary to write an EMPTY word into the INPUT register. After that an execution has completed, to begin a new one it needs to receive the RESET input command. When the STATE is no more on COMPLETE it is possibile to send the START command, the system will read the input data and start a new execution. Looking into the internal structure of the automata, there are 8 states: *WAIT_DATA*, *RESET_FF*, *ASSERT_CE*, *CLOCK_1*, *CLOCK_2*, *CLOCK_3*, *CLOCK_4*, *DEASSERT_ALL*. When the FSA is into *DEASSERT_ALL* state, the execution is completed and the STATE register contains 0x03. If the CONTROL register is EMPTY, the next clock cycle the automata moves into the *WAIT_DATA* state and the state register will contain 0x02. Now The RESET command will reset the internal state of the EHW element and it will set the output value to 0x00. The START command enables the transition toward *AS-*

SERT_CE state. In this phase the input is red and the execution starts. It will last between the state *CLOCK_1* and *CLOCK_4*. Then the execution is completed and the FSA is back to the state *DEASSERT_ALL*, with the state register set to 0x03, the last bit means that the execution is completed.

3.3.5 Performance

The evaluation of an individual with a four columns depth needs four clock cycles to complete, plus other three clock cycles for input/output related operations and other three to reset the internal FFs before a new execution. In total it takes 10 clock cycles. An extrinsic software simulation of the same hardware individual may require hundreds clock cycles.

Until now, few works have been carried out on Complete Hardware Evolution. The first [27] is the work in which the definition of complete evolution has been introduced. It concerns robotic, the creation of a simple hardware robot controller using an evolvable strategy, that is hardware implemented as well. However it is not a good sample of hardware component. It is extremely tailored on a peculiar task and there are no consideration concerning scalability issues, deployment and many other typical constraints. The robot has been realized with Lego Mindstorm, implementing the controller on an FPGA following the approach proposed by Thompson and described in subsection 3.2. The innovation consisted in putting also the evolvable algorithm on the same chip. Nowadays there are not yet open-ended hardware architecture. In his Phd. thesis [65] A. Upegui argued that this is the only category that can be considered truly EHW. For this reason he tried to identify some open-ended task that could theoretically benefit from an evolvable hardware implementation. However to be able to realize *Open-Ended* system in the future, it is necessary to be able to realize *first reliable complete* systems that exploit the most modern devices and powerful evolutionary algorithms. Realizing complex system

requires to deal efficiently with EHW issues. Aiming to improve the state of art in this direction, the next chapters first will focus on the integration between an evolvable individual and Genetic Algorithm. Then a efficient hardware implementation of a selected GA will be proposed to show how issues related to EHW can be addressed efficiently implementing a Complete Evolvable Hardware system.

Chapter 4

Proposed Methodology

In this Chapter, the approach used in this thesis to address Evolvable Hardware issues and to develop a Complete system, that efficiently settle them, is presented. First the overall approach to the problem is presented. Then, details on the single steps are provided.

In Chapter 3 several Evolvable Hardware systems have been presented, including the architecture that is used in this thesis, that is described in Section 3.3. In the largest part of the evolvable systems the Evolutionary Algorithm is provided with a software-based implementation. However, it is better to have also an hardware-based evolutionary algorithm, to create a real *Complete* evolvable system, on a System-on-Chip evolvable architecture. The computation performance of PowerPC (PPC) embedded on Xilinx programmable logics are not comparable with those of a workstation. For this reason it is necessary to implement an hardware-based Genetic Algorithm, designing a Complete architecture. However, solutions adopted to address EHW issues with Extrinsic Evolution may not be suitable for a Complete architecture.

The flow chart in Figure 4.1 shows the five key steps of the proposed methodology to address the problem of creating a Complete Evolvable Hardware architecture, beginning from the definition of an EHW system

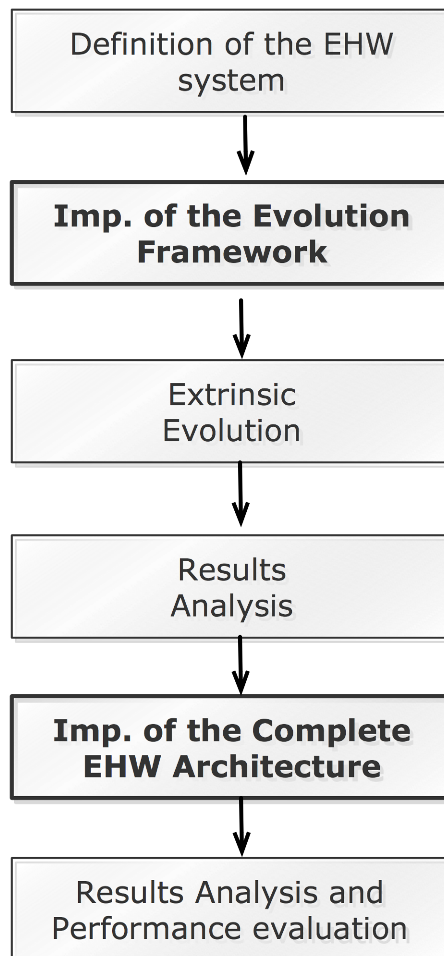


Figure 4.1: Steps toward a complete system

and arriving to an efficient hardware architecture. The first step can be considered the statement of problem, while the last one as its solution. The three intermediate steps are the procedure to approach the problem.

The first step is the definition of the Evolvable Hardware system that will be used. To make an efficient implementation it is necessary to focus on one evolvable system. Considering its characteristics and the relation between the Evolvable Hardware genotype and the functionalities implemented. The considered EHW system has already been presented in section 3.3 and it will be used in the next steps.

The second step is the implementation of a *software framework* able to *simulate* the specific Evolvable Hardware system, allowing to evolve it with various algorithms. In particular it is necessary to consider the algorithms present in literature and to make an implementation of them that can work with simulated EHW system. To choose what algorithm to implement in hardware for the Complete Evolvable system, it is important to study the behaviour of a software implementation first. This framework allows Extrinsic Evolution of hardware components and its details are described in the next chapter.

From the simulation-based Extrinsic evolution results, it will be possible to gather information about how efficient are different algorithms, with different characteristics. These results are important in order to take correct design choices, that can allow to create an efficient hardware Complete architecture. The evolution has been executed running the framework on an Intel Core2Duo 2.20 Ghz with 4gb of RAM.

The results obtained from Extrinsic evolution need to be analyzed with a precise methodology. The most relevant information that are necessary in order to take wise design choices go besides the simple time performance of a software execution, because an hardware implementation may be parallel and structurally different from a software one.

The most relevant characteristics to be considered, of the different algorithms, are:

- Convergence Rate: How many generations are necessary to obtain en evolved Evolvable Hardware that achieves the best performance. The software framework allows to evaluate how complexity impacts on the convergence rate of the various algorithms. It may be useful to know also the convergence rate after a certain number of generation.
- Population size: It indicates how many individuals are tested for every generation. From the convergence rate and the population size, it will be possible to determines the absolute number of evaluation done. It determine often the execution time required by the evolution. If it is generally true that larger the population is less generation are required, on the other hand increasing the population size may increase unnecessarily the number of evaluation to be done. Therefore, it may lead to a slowdown of the system.
- Memory requirement: Usually it is not a relevant aspect to run an evolution on a workstation, but moving toward an hardware implementation on Filed Programmable Gate Array, it may become a critical aspect. Memory availability on FPGAs is constrained and, moreover, the memory access to a Block Random Access Memory (BRAM) is not a parallel operation, but sequential. This fact constraints further the achievable parallelism. Memory requirement is usually proportional to the number of individuals that needs to be available in memory at the same time.

The same algorithm may have different convergence rate, population size or memory requirement depending on some parameters. These parameters need so to be evaluated in function of the time required to evolve an Evolvable Hardware component. Moreover, implementing an algorithm

with hardware it is not likely possibile to choose freely parameters as population size, but there are constraints depending on the hardware resources, as area and memory, available.

Once that the an efficient Genetic Algorithm with a correct parameterization has been identified as suitable for an hardware implementation, a Complete EHW architecture can be implemented.

In this thesis, this methodology has been used to develop of an *efficient hardware architecture* for the Complete Evolution, with individuals evaluation and evolution, both implemented in hardware. The next chapter focuses the key steps of the above methodology while Chapter 6 focuses on the description of the implemented hardware architecture.

Chapter 5

Extrinsic Evolution Analysis

This chapter focus on the two most important phases of the proposed methodology described in the previous chapter. The extrinsic evolution with the *Simulation Framework*, that is described in the first section, and the *Analysis of the results* obtained, that is described in the latter section.

5.1 Simulation Framework

The Simulation Framework is an application composed of three main parts, the *Simulation Function*, the *Testing Unit* and the *Evolutionary Algorithm*.

The Simulation Function is the software part that models the behaviour of an hardware evolvable system. It receives as a parameters a data structure that represents an Evolvable Hardware individual and the input data. Simulating the behaviour of that individual, it computes the expected output value. Such procedure is a composition of a key function *compute*, which simulates the behaviour during a clock cycle of one basic evolvable cell. The cell considered is the one described in section 3.3.3, it is composed of two Look-Up-Tables (LUTs) connected to a multiplexer (MUX) driven by the former output value of that cell, which is stored in the Flip-Flop (FF).

```
int compute(int inputs, Cell * actCell)
        int C=0;
        int index=0;
        index= (actCell->FlipFlop << 4) | inputs;
        C = actCell->FlipFlop = (actCell->luts & (1 << (31-index)));
        return C;
```

Figure 5.1: Cell simulation function

The procedure in Figure 5.1 shows how the behaviour of the cell is simulated with bitwise operations. The cell receives an input of four bits, that is implemented in the simulator as an integer number in the interval 0 to 15. These primary inputs are negated and forwarded to both the Look-Up-Tables (LUTs) in the cell, as it happen on the real device, but only one of the two must be considered, depending on the value of the Flip-Flops (FF). This is done adding the FF value as the most significant bit of the input. Now the two LUTs can be considered as just a larger one, of 32 bits instead of 16. If the value in the FF is 0, the output value will be selected on the second LUT, so the most significant bits of the *luts* variable, because the input complement will be in the range 16 to 31. Elsewhere the output will be selected in the range 0 to 15. Having two LUTs with 4 inputs and a MUX driven by an additional input, it allows to have a component that behaves as a single LUTs with 5 inputs, one of which is the state. The negation of the input serves as a index of what element of the LUT is taken as output value of the cell. This describes exactly the behavior of the cells implemented on the Filed Programmable Gate Array. Modifying the content of the *luts* variables, it is possible to simulate a modification in the contend of the LUTs that may change the output of the cells.

This function needs to be called multiple times, over all the cells of the individual, to simulate the whole behaviour by forwarding the outputs of an earlier cell to the input of the next one.

The final output consists in the last 8 outputs gathered from the last 8 cells of the simulated individual. It is necessary to guarantee that the connections between the simulated cells match with those of the *Hard Macro* implementing them on the device, in order to have a consistent behaviour of the results obtained. It is a known fact that Xilinx hardware synthesis, placement, and routing tools may modify the internal connections of the Hard Macros. To have a consistent behavior, that still match with that one simulated, it is necessary to restore the original connections between the cells of the individuals. This can be done editing the placed design after the routing phase and before the bitstream generation. However this must be done just when the overall evolvable architecture is implemented. Individuals deployment does not suffer of such issues, because the connections are fixed.

The framework receives as input a parameter that defines what Genetic Algorithm must be used for the current simulation and the name of the file that contains the configuration parameters. In such configuration file is specified also the path to the *tests database*. It is a textual database which contains the set of tests that must be used to evaluate the individuals fitness during the evolution. For different Genetic Algorithms, there are different parameters to be specified, but some may be common. For example the population size is a common parameter for almost all the algorithms. In the following section, presenting the implementation of the algorithms, the parameters which characterize them will be presented too.

The fitness calculation is the last important aspect to be considered before to discuss the details about the algorithms. Generally speaking, the goal of the evolution is usually to have an individual able to hit a performance of 100%. Such individual will make no mistake when it works with the test cases used during the evolution phase. The database that implements the test set contains couples of two *words* of eight bits length each.

The first is the input word, while the second is the the expected output. An output word that may also contain *don't care* (DC) bits. Where there is a DC bit, both one or zero are considerable as correct results. Once defined test sets, there are still two general methods to calculate the fitness. The first is an *All or nothing* approach. It is possible to add one to the fitness value, for each test for which the result of the simulation matches with the expected results. Otherwise the hamming distance between the expected solution and that one obtained can be considered, smaller the divergence is, higher will be the fitness. In the case in which the simulated function F has as target domain the set {0,1}, these two methods are equivalent.

For this thesis purposes four Genetic Algorithms have been implemented, together with some modification of them. They are: the Simple Genetic Algorithm (SGA), the Messy Genetic Algorithm (MGA), the Compact Genetic Algorithm (CGA) and the Extended Compact Genetic Algorithm (ECGA). In the next subsection, the implementation of these Genetic Algorithms, that is used by the framework, will be described in details.

5.1.1 Simple Genetic Algorithm Implementation

The first algorithm implemented is the Simple Genetic Algorithm (SGA), proposed by Goldberg in 1989 [29], and described from a theoretical point of view in section 2.1.1.

To implement the algorithm, a customized implementation of each of its genetic operators has been provided:

- Evaluation. It uses the *simulation* function introduced in the framework to obtain the outputs. The hamming distance between the outputs and the expected outputs is the score obtained by a chromosome in a specific test case. Fitness is computed running all the test cases and summing the scores.

- Selection. This operator has been implemented as *Tournament Selection*. It receives as input the evaluated population and returns a subset that contains the selected individuals.

- Crossing-over. It has been implemented an uniform probability crossing-over. It returns from a couple of chromosomes a new one. It is used to generate the new population by applying it multiple times to all the possible couples in selection set.

- Mutation. Also this operator acts with uniform probability and single gene granularity.

- Termination condition. All the three termination conditions have been implemented. The execution terminates when there is no more variance in the population, a maximum number of generations has been executed, or an optimal individual is generated. The highest score possible is the product of two factor: the number of tests and the output width.

A *replace-the-worsts* policy is applied to the population, selected individuals are maintained and worsts discarded. Then, selected individuals are inserted in the new population together with the newly generated.

To customize the behavior of the algorithm, the following parameters have been defined in the configuration file:

- Population Size (n). It defines how many individuals there are in the population. N individuals are initialized at the begin and N must be at the end of every iteration of the algorithm. So all pruned elements need to be replaced.

- Mutation probability (mp). It is the probability to mutate one single gene of a chromosome, from 0 to 1 or vice versa.

- Crossing-over probability (cp). Given two individuals it is the probability to have a crossing-over between a pair genes in the same position.
- Tournament Size (ts). It is an important parameter for the selection, besides the already discussed properties it determine indirectly also how many individuals are selected. The population is randomly divided is a series of set each of ts size, for each of them is selected the individual with the highest fitness. So, the number of selected individuals is N/ts.

This algorithm generally benefits from a large population, it reduces the number of generation required to converge to an individual with the best fitness. Unfortunately, on the other hand increasing the population size also increases the amount of time required to perform an iteration. Especially, evaluation time may become a bottleneck. To achieve the best performance it is necessary to budget will the population size, but there is no systematic methodology to do that. This is a topic often discussed in literature [73].

An additional parameter that generally caracterize the algorithm is the number of bits, or genes, of the genotype. Due to the fact that the simulator is not with general purposes but address a specific problem the individual structure has been hardcoded.

5.1.2 Messy Genetic Algorithm Implementation

The Messy Genetic Algorithm (MGA) has been implemented with the most significant customization with respect to its original formulation, already presented in section 2.1.2. Authors introduced MGA with the purpose to handle complex problems, with unknown structure, by identifying first subproblems and Basic Blocks (BBs) that solve them. Differently, now the task is to implement a messy approach to address a specific problem,

which structure is in large part known. That allows to make some preliminary considerations and to adapt the implementation according to them. In particular, looking into the evolvable individual structure and the specific feature of the problem it is possible to say:

- There are no clear subcomponents that may address some subproblems, because the connections among cells constitute a dense mesh. Analyzing paths between primary inputs and outputs, it is evident that all the outputs depend on almost all the other cells, except other output cells and half of the 3rd column cells. All the output values depend on 65% of the cells. Moreover, the cells belonging to the first and the second column influence all the outputs.

- Partial evaluation of a chromosome is not possible, to be simulated an individual must be complete. Exploiting a *template* chromosome may not lead to significant results, still due to the structure of the Evolvable Hardware module.

- Having to multiple test Basic Blocks may require a large increase in the number of simulation to be done. In Evolvable Hardware fitness evaluation is a critical operation, extremely time consuming and it is better to minimize the number of evaluations required.

The solution adopted has been to implement a *pseudo*-MGA, such that it makes more greedy the identification of the Basic Blocks following the Evolvable Hardware characteristics. It has been done according to the following guidelines:

- Chunks of 16 contiguous bits are considered instead of single genes as base to build Basic Blocks.

- Primordial phase searching should not require too many additional evaluation.

Blocks of 16 contiguous bits in the chromosomes represent Look-Up-Tables (LUTs), they will not be split over more Basic Blocks. That allows to handle contiguous linkage. The algorithm procedes alternating the two phases, one *Primordial* and one *Juxtapositional*. The Primordial phase generates individuals by evaluating some candidate blocks with a template chromosome. Candidate blocks are obtained by taking some 16-bits chunks from the selected individuals and merging them with a template chromosome. Those 16-bits chunks belonging to a candidate Basic Blocks are labeled in order to be able to identify them in the next phase. High-fitness blocks are divided from low-fitness blocks applying selection on the chromosomes generated during primordial phase. During Juxtapositional, phase the *Cut* operator will unlikely break those blocks identified in the previous phase. Primordial and Juxtapositional phases both act generating the same number of individuals and running the same number of tests.

Operations of *Fitness Evaluation*, *Mutation* and *Selection* are implemented and parametrized as in the Simple Genetic Algorithm.

5.1.3 Compact Genetic Algorithm Implementation

The Compact Genetic Algorithm (CGA) [15], presented in section 2.1.3, introduces further innovations modifying how the population is represented in memory. In MGA and SGA the population is a set of N individuals. It requires N*1024 bit to be stored. The Compact Genetic Algorithm uses instead a Probabilistic Vector (PV) of 1024 elements. How many bits it takes in memory depends on the precision of the numbers in the Probabilistic Vector, but generally they are less than those required by the population of MGA and SGA.

The Algorithm 4 shows the CGA in the version that has been implemented in the framework.

The additional parameters that characterize the proposed Compact Ge-

Algorithm 4 Compact Genetic Algorithm Implementation

1: **for** $i = 1$ to ChromosomeSize **do**
2: $P(i) \leftarrow 0.5$
3: **end for**
4: **repeat**
5: $a \leftarrow NewIndividual(P)$
6: $b \leftarrow NewIndividual(P)$
7: $a.fitness \leftarrow Evaluate(a)$
8: $b.fitness \leftarrow Evaluate(b)$
9: $winner, looser \leftarrow CompareFitness(a, b)$
10: **for** $i = 1$ to ChromosomeSize **do**
11: **if** $winner.dna[i] < looser.dna[i]$ **then**
12: $P(i) \leftarrow max(d, P(i) - t)$
13: **end if**
14: **if** $winner.dna[i] > looser.dna[i]$ **then**
15: $P(i) \leftarrow max(1 - d, P(i) + t)$
16: **end if**
17: **end for**
18: **until** $Termination()$

netic Algorithm are the step size (t) and the margin (d):

- Step Size (t). It defines the size of the step used to update the Probabilistic Vector. An higher t allows to converge faster to a solution but increases the risk of stalling in local maxima.
- Threshold (d). It is the margin that defines the minimum probability that every element has to be 0 or 1. Consequently the maximum probability will be 1-d.

It is not necessary to define a population size because at every iteration just two individuals are generated and the best is taken as winner with probability 1, through a binary tournament.

The main customization that has been done, with respect to the original implementation, concerns the termination condition. CGA execution usually ends when all the elements in the Probabilistic Vector that describes the population converge to 1 (max) or 0 (min), but in this implementation the execution ends when an individual with a sufficient performance has been obtained, or after a maximum number of generations. The elements of the probabilistic array are prevented to converge to 1 or 0 thanks to the introduction of the parameter d, so there is always variance and the third termination condition becomes useless. Such limit has been introduced to guarantee always a minimal possibility to generate a novel individual not yet obtained or to have a mutation in a sequence that has almost converged to 0 or 1. For trivial tasks this may reduce the convergence speed but for more complicated execution it increases the capability of the algorithm to explore new solutions.

It has been shown that one of the limits of Genetic Algorithm is the needed amount of memory required to represent the whole population, and that this limit is overcome by CGA. However, not having a representation of all the generated individuals rises some issues. Since the CGA

operates on each gene independently, it may lose linkage information. The pe-CGA and ne-CGA algorithms proposed in [74] try to handle this problem and to increase the convergence rate, when the size of the problem grows.

- The pe-CGA finds a near optimal solution, called *elite* chromosome, that is maintained as long as other solutions generated from probability vectors are no better. *Pe* means *persistent elitism*.
- The ne-CGA further improves the performance of the pe-CGA by avoiding strong elitism that may lead to premature convergence. *Ne* means *non-persistent elitism*. In this algorithm, after every N iterations, the elite chromosome is reinitialized with random values. This may lead to a slowdown in the convergence speed, but avoid situations in which the algorithm does not converge.

In the simulation framework a slight modification of the described non-persistent elitism has been implemented. Instead of discarding the elite vector after a fixed amount of N generations, the implemented algorithm discards it with a certain probability at every iteration. Such probability can be tuned in order to have that the expected value of generations, for discarding the elite individual, it is exactly N.

5.1.4 Extended Compact Genetic Algorithm Implementation

The Extended Compact Genetic Algorithm (ECGA) [75] has been introduced as an improvement of the search capability of the Compact Genetic Algorithm, it is described in section 2.1.4. ECGA has been implemented in the simulator by embedding a customization a library described in a technical report from *Illinois Genetic Algorithm Lab* [14]. The *Greedy Search* strategy to build the probabilistic models is the one from *Illigal* and, as in CGA, there is no operation of crossing-over. The *Selection* implemented is

Tournament Based and the size of the tournaments can be arbitrary defined. *Fitness evaluation* function is the one implemented in the simulation for the other algorithms presented above. *Mutations* are introduced with uniform probability. An issue of the ECGA concerns the population size. In CGA the population is fixed to 2, in SGA and MGA it can be defined as an integer positive number greater than 0, but in ECGA the greedy search strategy requires it to be greater than or equal to 50. Moreover, a negative aspect of this algorithm is that if CGA requires $1024 * \mathtt{sizeof(float)}$ bits of memory, ECGA requires $1024^2 * \mathtt{sizeof(float)}$ of them to represent its probabilistic model.

5.2 Results Analysis

In this section results obtained from the Extrinsic evolution analysis are presented, this is an important step of the design flow. Average data refers to the mean value of 10 iterations, when not specified differently. Time performance have been measured executing the software framework on a Intel Core2Duo 2.20 Ghz with 4gb of RAM.

5.2.1 Parity generators

It has been decided to use the evolution of parity generators as first benchmark of the evolvable system, because it is a not complex function, it has one output and allows to decide arbitrary the input to be considered, within the range allowed by the component data-path.

With the given EHW system, it is possible to run tests from 1 bit of input to 8 bits of input, non-relevant input bits are just set to 0. Being only one of the output bit relevants for the output value, the others will not be considered for the fitness computation. The Simple Genetic Algorithm (SGA), the Messy Genetic Algorithm (MGA), the Compact Genetic Algorithm (CGA)

and the Extended Compact Genetic Algorithm (ECGA) will be evaluated with two selected relevant test cases, not with all the 8 possibile. These two selected case are the 4 bits parity generator and the 8 bits parity generator. The first, due to its data-path, is simple but not trivial. It allows 16 possible input words. It is already a simple task, to use smaller input width will be meaningless. The latter test case, with 8 bits of input width, is the generation of the most complex parity generator that can be evolved by the system.

Table 5.1 shows the summary of the analyzed algorithms performance, evolving parity generators with 4 and 8 bits. For what concerns the time, it is necessary to consider that the framework implements output, logging and debug operations at every generations. Their impact on the performance depends on the number of the genrations, therefore it becomes more relevant when there are less individuals in the population. For the Compact Genetic Algorithm, the average number of individuals evaluated has been computed counting two per generations. For the ne-CGA is has been considered one individual per generation.
From the analysis of this data it is possible to make a series of considerations:

- Simple Genetic Algorithm works fine for simple tasks, but its performance drops while working with more complex. It heavily suffers the reductions in the population size.
- Messy Genetic Algorithm allows to achieve the best performance in both the test cases. Moreover, when the number of individuals in the population is comparable, and the number of generations is almost the same, MGA results to be faster than SGA.
- Compact Genetic Algorithm requires more evaluation than MGA, but the execution time remains small, since the algorithm instructions are

Table 5.1: GAs evolving parity generators

Algorithm	Pop. Size	Avg Ind. gen.	time(s)
Parity 4 bits			
SGA	32	14464	7.62
SGA	64	4416	3.66
SGA	96	5664	1.8
SGA	128	7168	3.2
MGA	32	4800	1.2
MGA	64	3968	1.1
MGA	96	4704	1.3
MGA	128	5760	1.45
CGA	2	7650	2.0
ne-CGA	1-2	5528*	2.5
ECGA	64	13312	1123
ECGA	96	10944	845
ECGA	128	10240	660
Parity 8 bits			
SGA	64	16000	75.8
SGA	96	13632	61.9
SGA	128	16512	76.3
MGA	32	9504	50.0
MGA	64	6016	51.7
MGA	96	6912	51.9
MGA	128	8448	62.3
CGA	2	13776	51.8
ne-CGA	1-2	13286*	75.6
ECGA	64	38592	2996
ECGA	96	18912	1635
ECGA	128	29952	2203

all simple.

- Non-persistent elitism CGA (ne-CGA) requires an absolute number of evaluations that is comparable with that one of CGA, but the time to execute the algorithm results to be higher. Probably operations involved in generations management and logging lead to a too high overhead, since they are done at each generation.

- Extended Compact Genetic Algorithm requires an average amount of generations to converge, but the time overhead to manage the probabilistic models seems to be a too high price respect to the performance that allows to obtain. Unnecessary complexity makes this algorithm the slowest tested.

5.2.2 Complex multi-outputs functions

In the previous section, the results obtained evolving a hardware component with one bit of output has been shown. Unfortunately, with the selected evolvable individual, it is not possible to obtain the same performance for every kind of task. Now, two 4-bits-output scenarios will be analyzed. Here, all the algorithms that has been considered under perform, with respect to the previous case.

A first case is the evolution of an accumulative counter of four bits length. The task is to implement an hardware component that, given a value *N* of input, returns as output the value *N+1*, or 0 if the input string is 1111. The remaining 4 bits of input are set to 0, while the 4 outputs bit not used are considered *don't care* values. Having a multibits output, the fitness value is computed as the sum of the correct bits for every test case. Thus having 16 test cases, encoded by 4 bits, and 4 bits of output there are 64 possible levels of fitness. Table 5.2 shows the experimental results obtained with the simulation framework. Despite running the algorithms with large

populations, of 128 or 256 individuals, difficulties in convergence has been highlighted. The Simple Genetic Algorithm has never been able to evolve a component that works in 100% of the cases. The performance decreases slowly reducing population from 256 to 128 but however the maximum level of fitness has never been achieved. Compact Genetic Algorithm does not shows good results too, but they were neither expected due to the fact that CGA allow to reduce the memory required for the population representation, but it does not have a search capability higher than the Simple Genetic Algorithm. The only algorithm able to evolve correctly the component is the Messy Genetic Algorithm, with a population of 256 individuals. It allows to achieve fitness 100% only in 20% of the cases but is the best result reached. Such results can not be considered good at all, because they identify a limit in the evolution of the hardware component.

If evolving a counter with only 4 bits of data-path requires 626 generations with 256 individuals each, we can guess that evolving a counter with 8 bits of data-path would be worst. It is possible to see that, to evolve a 4 bit counters 256 individuals have been tested for 626 times. Each test set is composed by 16 test cases, one for each input value. If all individuals are tested all the times with all the tests it means that to evolve a 4 bits counter almost 16k individuals are generated and 2.5 million of tests executed. If for hypotesis 626 generations are enough to evolve also a 8 bits counter, at least 41 million of tests would need to be executed. If one considers that, in a more realistic scenario, to evolve an 8 bits counter would require more than 626 generations, it becomes evident that this is a limit case in which the evolution fails. Looking experimental results, just evolving a 6 bits counter it is no more possible to achieve fitness higher than 85%. A worse case is the evolution of a 2x2 multiplier. As table 5.3 shows, no algorithm is albe to complete the evolution of the component having a 100% fitness. The best result is still obtained with the Messy Genetic Algorithm, but it is lower

Table 5.2: Performance evolving an accumulative counter

Algorithm	Pop. Size	Avg. gen.	best fitness	avg. fitness	100% rate
SGA	256	-	91%	89%	0%
SGA	128	-	89%	88%	0%
MGA	256	626	100%	91%	20%
MGA	128	-	95%	91%	0%
CGA	2	-	82%	81%	0%
ECGA	96	-	85%	81%	0%

Table 5.3: Performance evolving a multiplier

Algorithm	Pop. Size	Avg. gen.	best fitness	avg. fitness	100% rate
SGA	256	-	95.52%	94.3%	0%
SGA	128	-	93.75%	91.1%	0%
MGA	256	-	98.43%	96.87%	0%
CGA	2	-	93.75%	91.5%	0%
ne-CGA	1-2	-	93.75%	93.75%	0%

than expected. Only 98.3%. Moreover, always form table 5.3, it is possible to see that more than one algorithm tends to stall at fitness 93.75%. There is likely a local maximum, which corresponds to that value, from which it is difficult to come out.

5.3 Preliminary considerations

Previous results, concerning the number of generations required for the evolution, can be considered valid also in the intrinsic case. To implement all the generated individuals, instead of to use a simulation of their behavior, has no impact on convergence rate and algorithm behavior.

From the analysis done, it is possible to make some considerations.

Among those analyzed, the best Genetic Algorithm, with the best search capability, is the Messy Genetic Algorithm. When the Algorithm works with a population between 64 and 96 individuals it allows to achieve better results than others algorithms tested. To use more individual introduces unnecessary fitness evaluations, that slowdown the execution. In second place, it is possible to say that also SGA and CGA, which have similar search capability, allow to reach the solution showing good performance. Simple Genetic Algorithm seems to suffer too heavily from reduction in the population size. Compact Genetic Algorithm allows to overcome such issue. Its limit is the absence of elitism which in some circumstances may lead to slowdown. For this reason it has been analyzed also the implementation with non persistent elitism, that however led to uncertain results. Different consideration need to be done on the Extended Compact Genetic Algorithm. For the analyzed case studies, the additional cost of managing the probabilistic models overcomes the improvements achievable in term of convergence rate.

Generally all the presented algorithms scale well, when moving from the evolution of a 4 bits parity generator, to the evolution of an 8 bits parity generator. Doubling the input data-path of the component that is evolving, it doubles the amount of generations required to reach the best solution. For what concerns task as evolving a parity generator, increasing the complexity leads to a linar increasing in the number of generations required. Scalability leads to issues only with reduced population size. Difficulties of SGA to handle more complex problems, with reduced populations has already been highlighted. In that cases it is not possible to say that the algorithm scales well. Problems rise evolving multi-outputs functions. While evolving the counter, success are achieved only thanks to the usage of a large population of 256 individuals, and not always. To evolve such multi-output complex components, it would probably require to modify the fixed

structure of the connections among cells designing a new individual, but that goes beyond the goals of this thesis work. The analysis conduced with

the framework has allowed to identify the behaviour of the algorithms in function of the characterization parameters. Such knowledge achieved has been used to design the complete Evolvable Hardware system described in the Chapter 6.

Chapter 6

The System-on-Chip Implementation

In this Chapter, the implemented efficient hardware-based Complete Evolvable Hardware (EHW) system is described. That has been developed to evolve real hardware components. The first section will introduce the System-on-Chip (SoC) architecture, at high level first, identifying which are its main parts. The second section will deal with the problem of the fitness computation, describing how to speed-up such process with an hardware-based implementation. In the third section, the motivation behind the design choices, taken while implementing the hardware-based Genetic Algorithm, are explained. They are based on the results of the extrinsic evolution presented in Chapter 4. The fourth section will present the implementation of the hardware-based Compact Genetic Algorithm (CGA) used in this system. The fifth section will focus on how a Random Number Generator has been implemented, to provide random values to the Genetic Algorithm. The sixth section describes the software application that manages the input-output operations and the deployment of the individuals. The last section shows the Complete architecture deployed on the FPGA.

6.1 The System-on-Chip Architecture

The SoC architecture can be divided into three main parts:

- Dynamic Part, *The Evolvable Hardware* (EHW). It implements the individuals, one or more. This component will change at runtime the functionality that its EHW modules implement.
- Static Part. It is composed by the base system, the processor PowerPC PPC405, available on the board, the communication infrastructure, the DDR-RAM, the Internal Configuration Access Port (ICAP) core and the *Evolutionary core*. These elements are present on-board, or implemented at design time, and no more modified.
- Software Part. It is the executable binary code, that runs on the PPC. It just manages the execution performed by the hardware part.

The FPGA used is a Virtex 4 [67] XC4VFX12. It is one of the smallest board available, so it will be important to be able to design an architecture that scales well with the size of the FPGA and that is not too expensive in terms of area. The area requirement is a mayor issue to be addressed, especially in the prospective to implement more complex EHW systems on larger FPGAs.

The *Evolvable Hardware module*, the *Evolution module* and the *Software Application* will be presented in the next sections. First the basic architecture will be introduced.

The overall architecture includes a series of predefined components: the PPC405 processor, the central memory and the RS-232 [76], for the input/output communications. The ICAP manager core has been added to allow internal reconfiguration with bitstream manipulation. All these elements are connected to a PLB [77] bus.

The *Complete* evolution of hardware components can be seen as a sequence three main phases, which are repeated until the optimal solution

is achieved. While the number of the generations depends mainly on the quality of the evolutionary strategy used, and on the size of the problem, the time required by a single generation depends on how it is implemented. 6.1 shows the time required by a single generation. It is the sum of the time required by the generation of the individuals (Gen), by their deployment (Dep), and the time that fitness evaluation (Fev) takes. Fitness value is used by the next generation phase to generate the new individuals.

$$Tg = Gen + Dep + Fev \tag{6.1}$$

To implement real hardware components it is necessary to speed-up these three phases as much as possible, budgeting well how many hardware resources to dedicate to a phase or to an other. To implement all the generated individuals makes also possible a parallel hardware-based fitness evaluation, faster than a software-based simulation. On the other hand, increasing the problem size will increase considerably also the number of generations that are necessary to achieve the convergence to the optimal solution. It is not enough to optimize just the fitness computation, but it is also necessary to focus on the overall system performance.

Figure 6.1 shows the overall architecture with its main components.

It consists in two main entities, one that evaluates the fitness of all the individuals, while the other acts on the genotype and on the population representation. This second part uses the results obtained from the test cases to determine which individual has an higher fitness.

6.2 Individuals evaluation

Differently from what done with the software framework, the hardware architecture implements in a certain region of the FPGA the individuals. That makes available to the testing an hardware modules that implements

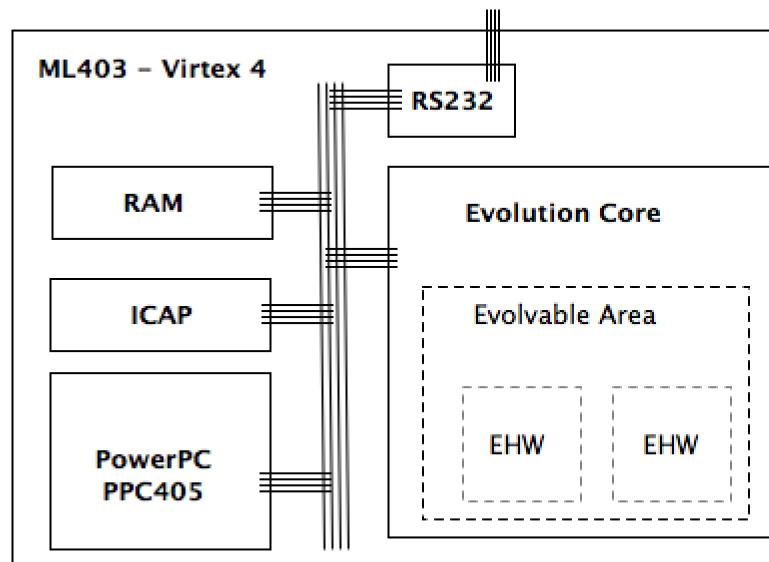


Figure 6.1: SoC EHW architecture

the individuals that must be evaluated. Figure 6.2 shows the testing module for K individuals with an N bits data-path. Into a Block Random Access Memory is stored the testing set. Every element of the set is a triplet (I,O,D), each element of the triplet is a N bits word. The element I is the input word for the individual to be evaluated. The word O is the expected correct output. When the set has a small size, that do not need N bits to be described, but less, it becomes useful the third word: D. It is the *"don't care"* array, it allows to specify that the values of some output bits are irrelevant. To build a parity generator, that need just to divide a set in two class, one bit of output it is enough. The input word I is forwarded to all the K elements implemented on the device which, thanks to the parallelism of the hardware, after 4 clock cycles returns K output words. Comparing the results obtained from the logic OR between the result and the D word and the logic OR between the expected result and the D word allows to determine if the behaviour of the EHW module is correct or not. This, for all the K EHW modules which are under testing. The results are stored into an

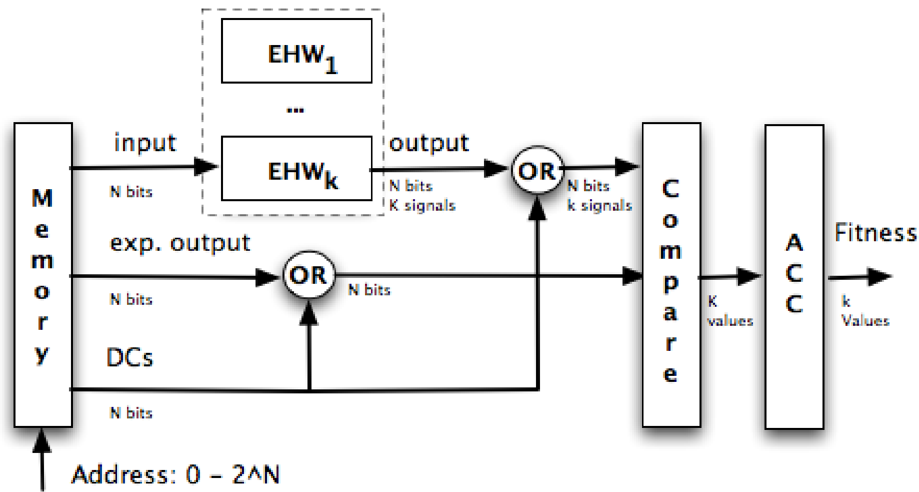


Figure 6.2: The testing module

accumulator for all the tests that are executed. It is important to notice that, with a N bits data path, there can be at most 2^N tests. As already described, the considered individuals that are evolved have a data-path of 8 bits. The number K, of EHW individuals to be evaluated in parallel, depends on the evolutionary algorithm used and on the availability of area on the FPGA.

6.3 Hardware portability of Genetic Algorithms

Which Genetic Algorithm to implement it has been decided according to the simulation results, obtained with the software framework. The Genetic Algorithms have been considered, not only analyzing their convergence rate, but also with focus on the expected resource requirements. From the results obtained with the extrinsic evolution and from a first analysis of the considered Genetic Algorithms structure it is possible to make the following considerations:

- Simple Genetic Algorithm (SGA) is not the best algorithm for an hardware implementation. In Chapter 2 two hardware-based implementa-

tions, that use this algorithm, have been introduced. In both of them there were issues concerning the area requirements and the population management, although small populations and small chromosomes were used. The considered EHW application needs a chromosome that is 1024 bits. It is 128 times more width than the chromosome used by Kooner and others [2]. Such problem rises serious scalability issues. Moreover, the algorithm to work at its best needs a population colse to 100 individuals. This causes two main problems. First, it would require a large amount of memory to store the entire population. Second, if theoretically it is possible to test all the individuals concurrently, practically it is not. It would require to dedicate to the EHW more area than what is available. A reduction of the population size may reduce the resources required by the implementation. But, in that case, it has been shown that SGA has difficulties to converge toward the optimal solution . Moreover, crossing-over remains a critical operation for the implementation, because it needs to access to two chromosomes at the same time. Access to memory is not parallel and neither it is possible to store two individuals in Flip-Flops. It would require thousandths of slices only for them. The only solution to this issue could be to have heavy time multiplexing.

- Messy Genetic Algorithm (MGA) provides significant improvements to the performance of SGA. The convergence rate is higher. For what concerns tests done in simulation, MGA has succeeded evolving a parity generator with 8 bits, also with just 32 individuals instead of 100, but it fails if the number is further reduced to 16. The main limit of this algoritm is the difficulty to make a parallel implementation of the generation phase. The splice and cut approach suffers of the same limits discussed about crossing over, but identification of Basic Blocks requires more complex data structures. Differently from the case of

SGA, it is realistic to implement a Genetic Algorithm with a population of 32 individuals on the device, but it is not certain that is the best option. Another element to be considered is how the algorithm could exploit hardware parallelism. In this case the implementation will be not so different from those proposed in literature for the SGA, with similar limits.

- Compact Genetic Algorithm (CGA) presents good performance in extrinsic evolution, comparable to those of MGA, and no major issue for an hardware implementation. As it has been shown by the implementation proposed in literature [78]. Memory requirement are much reduced, and the amount of memory used depends on the precision of the number into the Probabilistic Vector, which length is fixed. The number of individuals that have to be implemented and evaluated for each generation is two, and two individuals can be deployed and tested at the same time without too high area requirements. This allows to achieve an higher degree of freedom in hardware implementation. However, working with EHW and large chromosome would require to use a more flexible implementation than those proposed in literature for CGA. That allows to makes a good tradeoff between resources available and parallelism achievable.

- Extended Compact Genetic Algorithm (ECGA) presents major issues also implemented in software. The number of generations required is comparable those of others Genetic Algorithms, but the management of the Marginal Product Model is an extremely expensive operation. In this case is neither guarantee that the individuals generation phase depends linearly from the number of individuals in the population and the chromosome size. Therefore, it is possible to conclude that this algoritm is not the best solution for an hardware implementation.

These considerations led to choose for the implementation the Compact Genetic Algorithm. It seems to have the best ratio between hardware requirements and performance achievable. It allows to decide how much memory to dedicate to the Genetic Algorithm with less constraints. The availability of memory impacts only on the precision of the number in the Probabilistic Vector.

6.4 New Hardware-based Compact Genetic Algorithm

There is already an hardware-based Compact Genetic Algorithm proposed in literature [37], but the implementation proposed in this thesis has substantial differences. The architecture presented in literature is based on a series of blocks each aimed to generate a single bit of the chromosome and to update the probabilistic value that drive the generation of that bit. Having a chromosome of 1024 bits and limited area availability makes impossible to replicate the same approach just putting 1024 blocks. It is necessary to have a more scalable implementation. Figure 6.3 shows the phases of the algorithm. The only phase that requires to have access to the whole individual is the fitness evaluation (Fev), which has already been described, since it does not depend on the algorithm. All the other parts can work on the whole genotype, or on some chunks of arbitrary length. The genotype can be divided in two, four or eight parts, and the hardware modules could work in parallel on all them. For reasons of simplicity, an implementation that uses just one chunk on length 1024 will be now presented. After it will be described some small adaptation of the implemented hardware modules to work with multiple chunks, increasing parallelism, since multiple hardware modules can work concurrently.

Figure 6.4 shows the structure of the implemented CGA. The different hardware modules implement the main functionalities that corresponds to

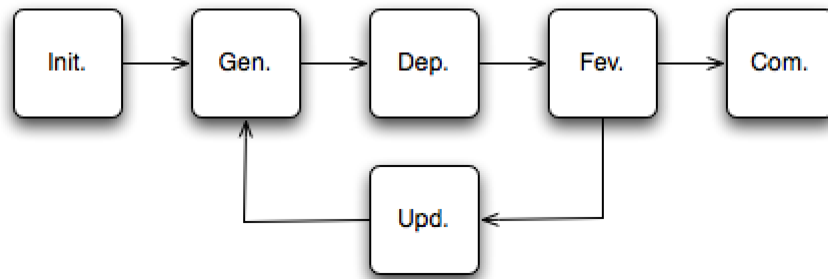


Figure 6.3: Phases of the hardware-based CGA

those shown in figure 6.3. The fitness computation has been described presenting the individuals evaluation. In the next sections the hardware modules that implement the generation phase and the update phase will be described. First a brief description of the hardware core initialization will be done.

6.4.1 Initialization

The first phase is the initialization. First the pseudo Random Number Generator are initialized with a random seed. RNGs will be discussed in details later. Then all the values of the Probabilistic Vector, stored in BRAM, are initialized to 0.5. It has been decided to use 16 bits of precision. The memory that stores the PV has 1024 elements of 16 bits each. Such memory is accessible through an address of 10 bits.

6.4.2 Generation phase

The next phase is the generation (gen). Such operation is parallel, sine two individuals are generated cuncurrently by the same hardware module. Two memory blocks are used, each of them contains 32 values of 32 bits each. Each BRAM stores a chromosome of 1024 bits. The generation is done using a component called *LUTFG generator*, it generates two sequences of

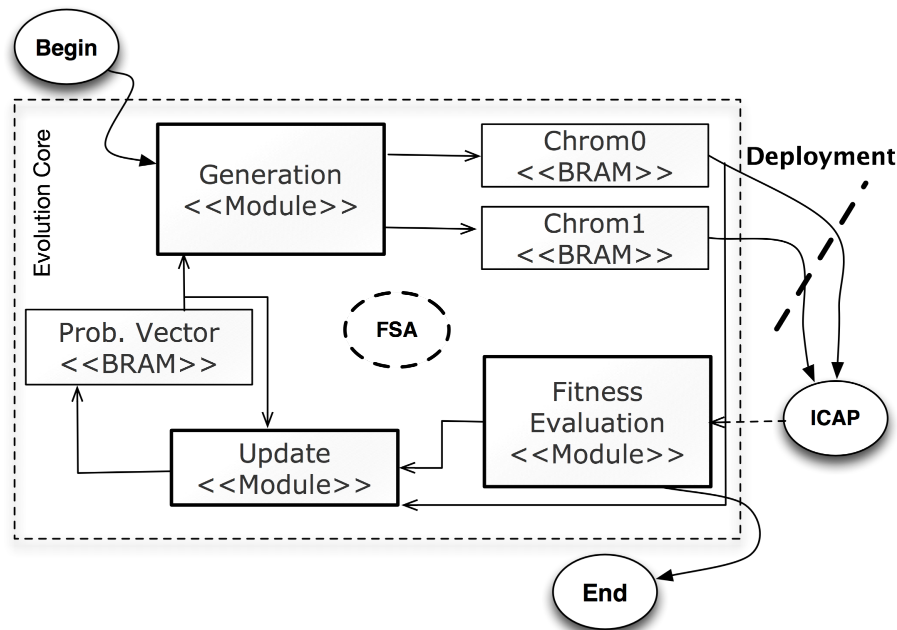


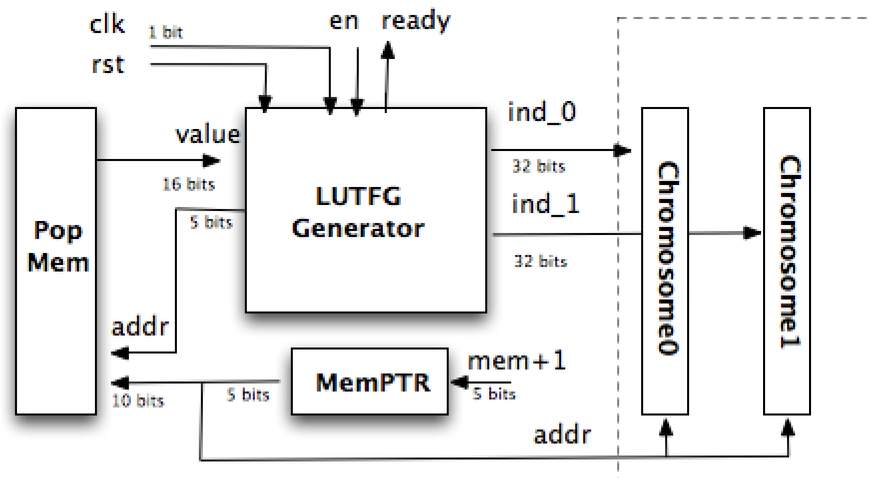
Figure 6.4: Implementation of the hardware CGA

32 bits each. One belonging to a chromosome, one to the other. 32 bits are exactly the representation of the content of the LUTF and LUTG of one evolvable cell.

Figure 6.5 shows in a schematic way how the two chromosome are generated with the *LUTFG* hardware module. The part *a* of the Figure focuses on the complete generation of the chromosome. That is done gathering all the genes of the cells from the *LUTFG*, while the part *b* describes internal details of that component. The Generator procedes iteratively for 32 times filling the buffer that contains the genes. It acts simultaneously on two individuals, having two Random Number Generators. Once the enable signal (en) is set to 1, it procedes through a series of step:

1. Initialize the counter (count) to 00000.

2. Read the input (value) from memory.



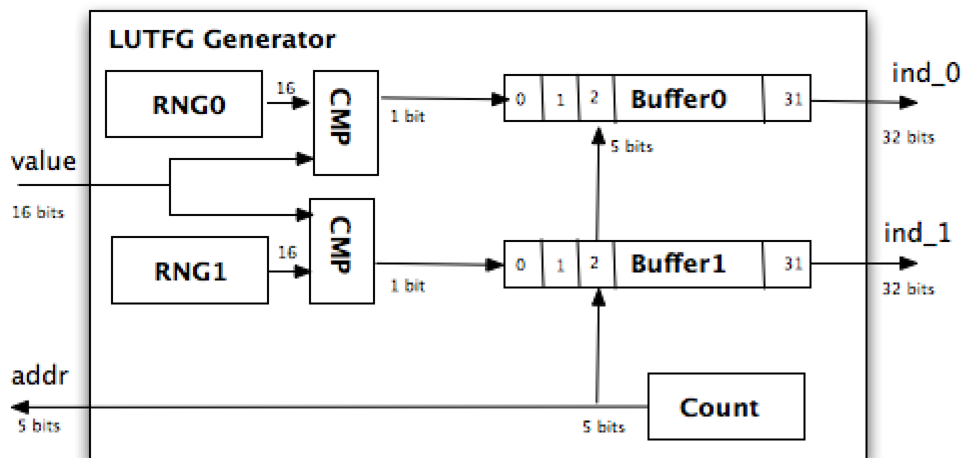


Figure 6.5: Generation phase

3. Each Random Number Generator (RNG) generates a 16 bits random number.

4. Compare the values from the RNGs to that one from input. An RNG returns 1 if the value from RNG is smaller than that from input.

5. Write the result of the comparison in the buffer array.

6. If counter is smaller than 11111 increment the counter by 1 and repeat from step 2.

7. Set the *Ready* signal to 1, to notify that the execution is completed and output is available. When *en* is set to 0 it resets and waits for a new executions.

For the generation of the whole chromosome, the same approach used inside the cell generator is used on larger scale. If there is only one generator, it must be used 32 times to generate all the 32 cells of the two individuals. A memory pointer (MemPTR) store the address of the cell that is currently being generated. It constraints a LUTFG Generator to access only to the portion of the Probabilistic Vector that contains value referred to a certain cell. Memory pointer also addresses what element in the chromosome is going to be written. Incrementing the memory pointer allows to generate the next cell changing the portion of memory which the LUTFG Generator can access to.

After to have generated the two individuals, the next phase is their deployment. In the hardware GAs presented in chapter 2.1 there was no deployment of EHW individuals, but directly fitness evaluation. In EHW case, it is necessary to stall the execution of the hardware-based GA to deploy the Evolvable component. The hardware module goes into *WREC* state, it waits for the individuals deployment. In this phase the multiplexers, that control the access to the core memories, connect them to externally

accessible registers. That allows the software application to read the two chromosomes, to generate the partial bitstreams and to send them to the ICAP. When the two individuals are completely deployed the application notify it to the hardware CGA core through a control register and the system procedes to the next step.

The core uses the testing module already described, implemented with two EHW individuals. When it complete its execution the fitness of the two individuals is gathered. The two fitness values are compared to determine the one with the highest score which is the winner.

6.4.3 Update phase

The next phase is the update of the memory. The values of the PV need to be increased or decreased according to the results obtained from the testing. The update is a parallel operation over all the element of the Probabilistic Vector, but parallelism is limited by the BRAM accessibility. Using only one memory for the Probabilistic Vector, and one for each chromosome, just an update for each iteration can be done. Such operations are performed by two Finite State Automatas (FSAs). The first has two states called *READONE* and *UPDATE*. When the Finite State Automata is in the first state memories are in *read* mode. The FSA iterates over all the genes of the chromosome chunk considered. For a given gene, when the FSA is in the first state, three values are red from memory and stored in buffers: the actual value of the PV for that gene, the gene value in the first chromosome (0) and the gene value in the second one (1). When values are red, the FSA switches to the second state. It implements a sequence of 4 micro phases, each of them lasts one clock cycle:

1. *Set The Address* (STA) phase. The value taken from the Probabilistic Vector is forwarded as input to an *addsub* module, which receives as a second input the *update step* size. In STA phase the core need to set

the two control bits of the *addsub* module. First is determined if the operation to be done is a sum or a subtraction, and an *add* flag is set. Then it is determined if to enable or not the execution of the memory update. The update is executed only if the current probabilistic value is within a certain margin. That prevents not only overflow, but it also avoids the value to converge to 1 or 0 prematurely, maintaining in this way a certain probability of mutation. Exactly as it has been done in the framework with the software-based implementation of the algorithm. In the next clock cycle, the FSA switches to ne next state.

2. *Read From Adder* (RFA) phase. The result obtained from the *addsub* module is taken and the enable bit is set to 0.

3. *Write* (WRT). It set the write bit of the population memory to one.

4. *Complete* (CMP). It waits that the memory write is completed. After that it is possible to execute a new *READONE*, processing the next element.

Figure 6.6 shows the sequence of the operations in the case in which the update operation is executed. Once the update of all the genes in the considered chunk of memory is done, the iteration is completed and a new generation is the next step.

Further parallelism can be easy achieved splitting the Probabilistic Vector and the chromosomes on more BRAMs and replicating the hardware modules that execute the *generation* of the individuals and the *update* of the Probabilistic Vector. Figure 6.7 shows the schema of an high parallelism version of the proposed hardware-based Compact Genetic Algorithm, with multiple hardware modules that operates concurrently. However, that architecture has not been implemented, since there is lack of resources availability and no possibility to introduce parallelism in deployment, which is the

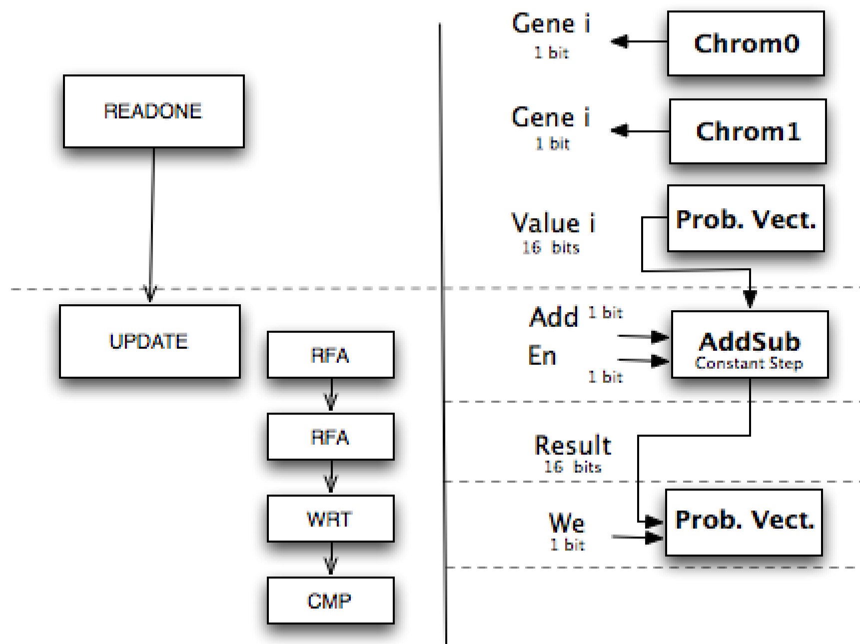


Figure 6.6: Update phase

most complex operation of the considered system.

6.5 Additional features

To the hardware-based implementation of the CGA a series of additional features has been added. They are: the possibility to introduce *Elitism* and the possibility to introduce an additional *Mutation*. Such features can be used or not, depending on the characteristics of the problem to be addressed.

6.5.1 Elitism

Despite its performance in the analyzed case studies have shown no real improvement, it has been decided to implement also the possibility to introduce elitism because it is extremely easy and requires flew hardware

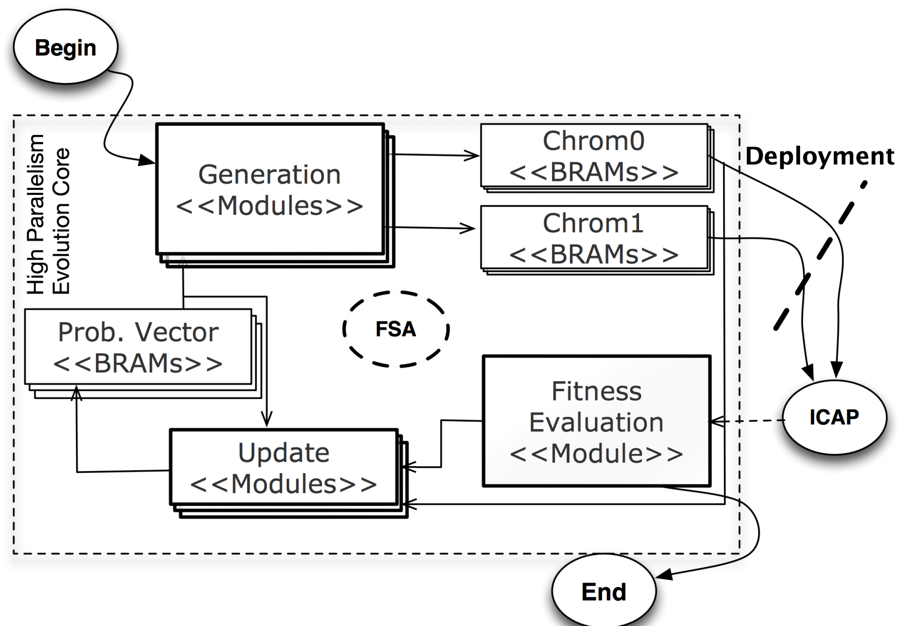


Figure 6.7: High parallelism architecture

resources. It has been added the possibility for the user to set an *elitism flag* to one, to introduce elitism. The implementation of this feature is based on a small modification of the generation phase. The only difference from the base implementation is that, when elitism is enabled, the generation module is prevented from writing the memory that contains the winner chromosome.

Elitism can be enabled and disabled at runtime, without further modifications *persistent elitism* and *non persisted elitism* [74] are available both.

6.5.2 Additional Mutation

To step out from local maxima and to increase the probability to find the global maximum another mechanism has been introduced: additional mutation. In the implemented Compact Genetic Algorithm is already embedded an uniform probability mutation mechanism with a single-gene gran-

ularity. The level of such mutation, defined by the threshold d, needs to be large to reduce the risk of stalling in local maxima, but at the same time small enough to do not generate too noisy chromosomes. These two goals are in contrast, for this reason it has been introduced an additional mutation to step out from local maxima. The single-gene mutation is kept small to avoid noisy chromosomes, that are likely to be pruned by the high selection pressure threshold. The additional mutation introduces, with an uniform probability, a mutation over a whole LUT, so 16 contiguous bits in the chromosome. In this way, it is possible to have a large mutation but without noisy effects on the chromosomes. The probability of this 16-genes-size can be defined, by the user, according to the characteristics of the problem to be solved. The time require by the introduction of the additional mutation is extremely reduced since the population-led chromosome and the random mutated chromosome can be generated in parallel.

6.6 Random Numbers Generation

To provide random numbers to the hardware-based Genetic Algorithm, an Hybrid Random Number Generator [79] has been implemented. It generates predetermined highly uncorrelated sequences, depending on an initial seed.

Figure 6.8 shows a black box image of a Random Number Generator. The first bit of the control signal is the *REQUEST* bit, when an edge rises the core starts to generate the next random number. The second bit of the control signal is the *REINIT* bit, it forces the core to reset its internal state and load a new seed into its memory. Two outputs bit describe the internal state. The first is used to notify when the random number is ready; while the second bit is 1 when the system is available, 0 otherwise. There are two cases in which the core can be unavailable: when a request is being processed

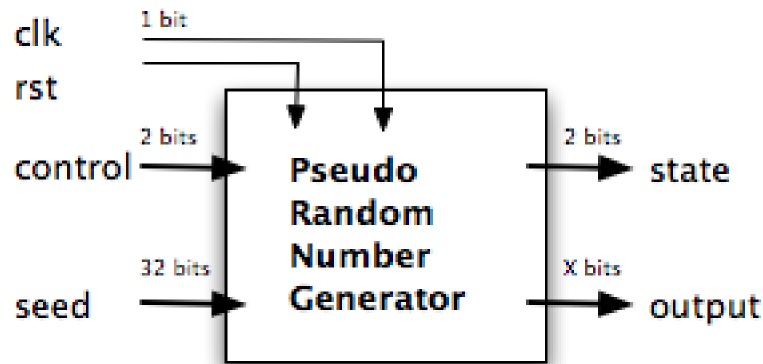


Figure 6.8: Random Numbers Generator, black box model

or when it is reading a new seed value. The Random Number Generator consists in a noise bit generator that fills an output buffer register. The size of the output buffer depends from the size of the output. The implemented evolvable algorithm uses 8 bits random number, so the output size is set to 8. Every noise bit generator requires a 32 bit seed and generate one noise bit for each clock cycle. Using just one noise bit generator, it takes N clock cycles to generate a N bits output number. The Pseudo Noise [79] generator has been implemented using Linear Feedback Shift Registers (LFSRs), as described in the Xilinx applicative guide [38]. This because a Virtex 4 LUTs can be configured as a 16 bits Shift Register. It makes such approach an extremely efficient implementation for area consumption and required clock period.

Figure 6.9 shows an example of the structure that a Linear Feedback Shift Register (LFSR) based noise generator could have. The LFSR has a sequence of 2^N -1 states, given a 16 bit initial seed, it can generate a sequence of up to 65535 random bits before to loop back and restart with the very same sequence. At each clock cycle, the content of the registers is shifted right by one position and the left most register is set to a value that depends from the feedback of some intermediate stages. All the implemented

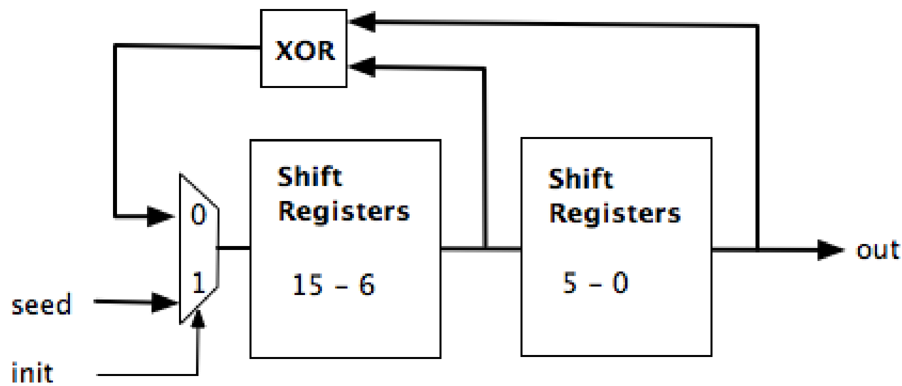


Figure 6.9: LFSR noise bit generator

LFSRs have the following characterization.

- The *Number of stages* is set to 16
- The *Number of taps* in the feedback path is set to two or four. There are two different categories of nose bit generator.
- The *Position of the taps*: It's 6-0 for the 2-taps LFSRs and 9-5-4-0 for the 4-taps ones.
- The *initial seed* must be provided.

Following the Xilinx recommendations, the last output, the tap 0, is always connected to the *last XOR*, in the case of the 4-taps LFSR there are three xor. To achieve better correlation properties, for the Random Noise bit used by the Random Number Generator two different LFSRs, one 2-taps and one 4-taps, have been used. That's why the RNG needs a 32 bits seed, it have to initialize two different 16 bits LFSRs. How the Random Noise bit is derived from the two LFSRs is stated by the 6.2, where *lfsr_i* is a 2-taps LFSR and *lfsr_q* is a 4-taps LFSR. The former is initialized with the first less significant 16 bits of the seed sequence while the latter with the remaining 16 most significant bits.

Table 6.1: Random number sequence properties

	value
Mean	307.2
Max	32767
Min	-32768
Bias	0.93%

$$noiseBit = lfsr_i \oplus lfsr_q \tag{6.2}$$

The XOR between the two signal is introduced with the purpose to reduce the possibility to have biases into the sequence of noise bits. A 16 bits Random Number Generator has been used to generate a sequence of 2000 random numbers that has been analyzed with MatLab in order to determine its quality. Let us call *S* the sequence obtained considering 2000 numbers of eight bits length each, encoded as signed number. All numbers *n* that belong to S are in the interval [-32768,+32767]. As is possibile to see in 6.1, from the results obtained by an early analysis, that the sequence has just a small bias of 1%.

However, this is not enough to say that S is a random sequence, but this can be done using the Matlab function *runstest(S)*. This function perform a test of the hypotheses *H0*, that the number in S come in random order, against the alternative *H1*, that they do not. The test should return H1 if reject the null hypotheses with a significance level of 5%, H0 otherwise. Running the test on the sequence S, it returns H0, so the hypotheses that numbers in the sequence are random is accepted. The analysis of *Auto-Correlation* function, which results are shown in Figure 6.10, gives confirmation of this result.

The Auto-Correlation is the partial cross correlation of a signal with itself. It aims to identify repeated patterns. As it is possible to see from

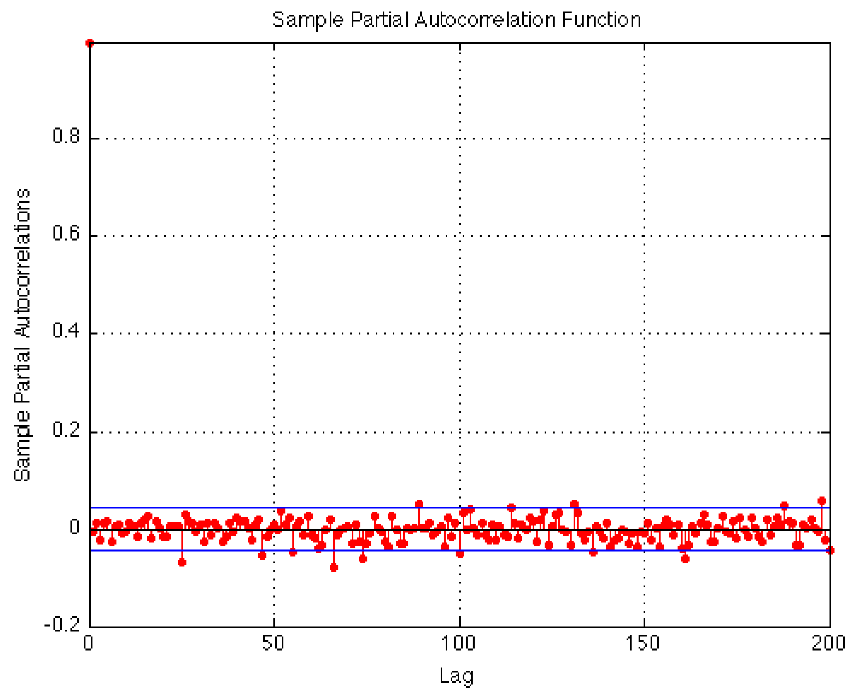


Figure 6.10: Partial autocorrelation, computed with Matlab

Figure 6.10 the correlation is maximum with lag 0, because the sequence is obviously identical to itself but besides that the correlation level is always extremely low, so we can conclude that there are no repeated patterns.

6.7 Input-outputs operations and individuals deployment

The last part of the evolvable architecture to be analyzed is the software application that runs on the PowerPC. Since the algorithm is hardware implemented, it has just two simple taks to perform.

- Manage the overall flow
- Connect the evolutionary core with the ICAP for the deployment.

For what concerns the overall workflow, first the application sets the *update step*, then the command to start the execution initializes the random number generators. At this point the application starts to execute the main loop. It repeats a sequence of operations until an individual with the best fitness is obtained. The execution can be described with the following steps:

1. The core is initialized, then the execution starts.
2. The application reads the two chromosome from the evolutionary core. The driver procedure that reads these elements is blocking. Execution does not procede until they are not available.
3. Partial bitstreams are builded and two individuals are deployed.
4. Once deployment is completed, that is notified to the evolutionary core, which procedes with the testing.
5. The application reads back the two fitness values. This is an other blocking operation for the software. Fitness is returned when is available, in the while the application waits.
6. If fitness is below the target value, the execution goes back to step two, else the it is completed.

Optionally, other functions provided by the evolutionary core driver can be called. When the core is waiting for the individual deployment, it is possible to:

- Reset the Random Number Generator, initializing them with a different seed.
- Gather some debug information, about the internal state of the core.
- Read the Probabilistic Vector content. It may be useful to implement an adaptive behavior, that modifies the update step or some the option following the convergence of the memory.

- Read, not only the fitness value, but also the results of the two EHW modules in each test case. It has been added a register to the testing unit to keep trace of the results.

6.8 The Complete Architecture Deployed

Figure 6.11 shows a view of the implemented architecture obtained with FPGA Editor. It is possible to see two red box on top right. They label the two EHW modules implemented for the individuals testing. Such architecture, implemented with one *LUTFG Generator*, requires up to 90% of the slices available on the FPGA. The yellow box identifies the static part of the evolvable architecture with the hardware-based Compact Genetic Algorithm and the communication infrastructure. The blue box identifies the onboard PowerPC used to run the application that manage the hardware-based evolution.

In this chapter the proposed EHW system has been described in details. The clock frequency of the implemented architecture has been set to 100 MHz, which is the highest affordable. In that condition, the architecture is able to run between 226 and 247 binary generations for second, the performance variance depends on the number of tests to be executed for each fitness evaluation. In the next chapter performance will be analyzed in details, presenting some case studies and comparing the results with Extrinsics and Intrinsics software-based executions of the Compact Genetic Algorithm.

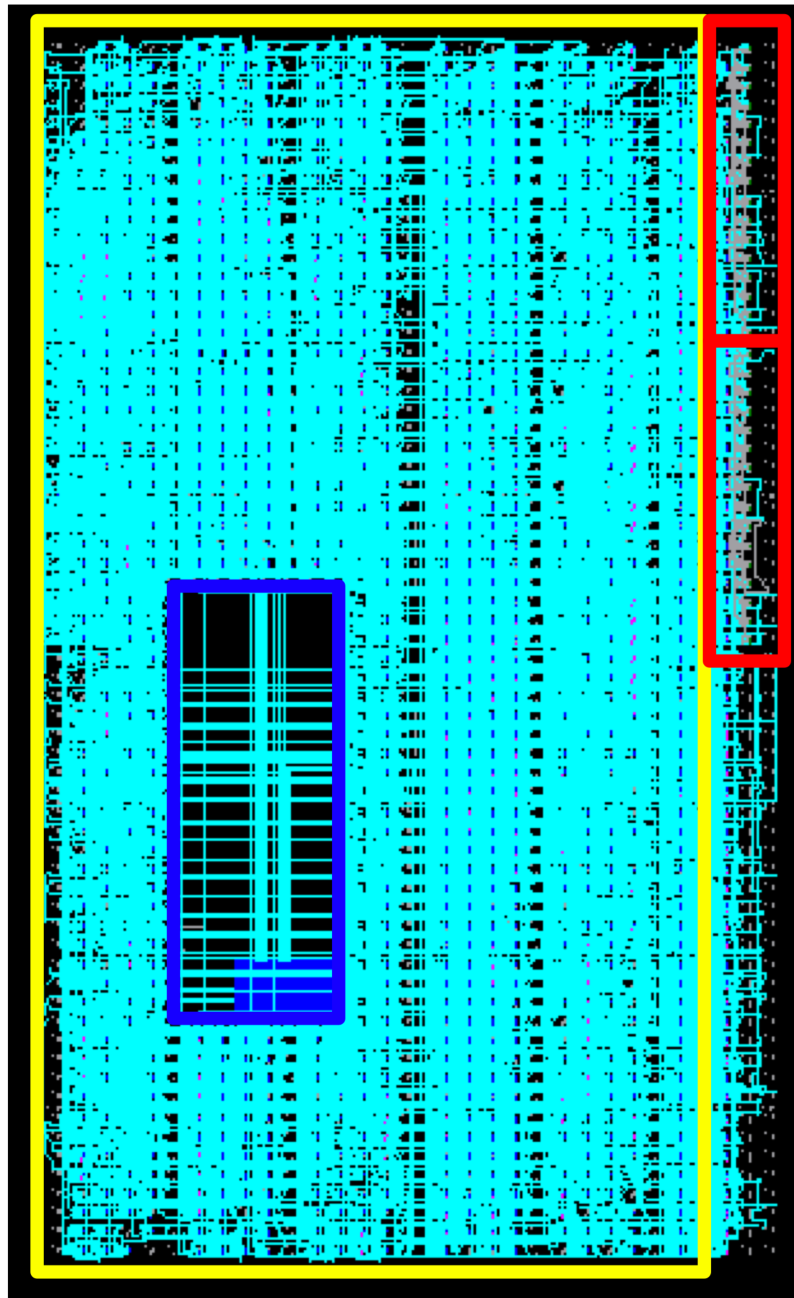


Figure 6.11: Architecture implemented, FPGA Editor view

Chapter 7

Performance Analysis

In this chapter, the performance obtained with the implemented Complete Evolvable Hardware System, described in the previous Chapter, are analyzed in details. The first section introduces the case studies and describes the rational behind them. The second section describes the tuning of the algorithm and the results obtained in the different case studies. The third section compares the results obtained with the Complete EHW system with the results that it is possible to obtain with an Extrinsic and an Intrinsic system, with a software-based execution of the considered Genetic Algorithm. The last two sections of this chapter focus on possible future works and discuss the capability of the implemented hardware-based Genetic Algorithm independently from evolvable hardware. Average data that is presented in the next sections refer to the mean value of 10 iterations, when not specified differently.

7.1 Introduction to the case studies

The aim of this Chapter is to show the performance of the hardware-based implementation of the Genetic Algorithm with respect to an Extrinsic or an Intrinsic software-based execution. The case study presented will be

the same that in the early evaluation of the Genetic Algorithms, done with software framework allowed to evolve successfully the component. The focus is on a single output function with a variable number of inputs.

7.2 Parity generator evolution

The task used to evaluate the system performance is the creation of a parity generator [80]. That is a component which implements a function that associate an input represented by an arbitrary number of bits, within the data-path width, to one output bit. The value of the output bit is 1, if the number of input bits set to one is odd, else it is 7.1 states such definition in a more formal way.

$$parity = count(input == 1)\%2 \tag{7.1}$$

It has been decided to use the evolution of this component as first benchmark of the evolvable system, because it is a not complex function, it has one output and allows to decide arbitrary the input to be considered, within the range allowed by the component data-path.

With the considered Evolvable Component, it is possible to run tests from 1 bit of input to 8 bits of input, non-relevant input bits are just set to 0. Being only one of the output bit relevants for the output value, the others will not be considered for the fitness computation. For them, the *"don't care"* value in the testing memory is set to one. Only parity generators between 4 bits of inputs and 8 bits of input have been evolved, with less than 4 bits the evolution would be trivial and meaningless, while 8 is the max data-path width of the considered evolvable hardware component. The 4 bits parity generator is simple but not trivial, the 8 bits parity generator is the most complex that is possible to evolve. The other cases with 5, 6 and 7 are useful to see how the performance change changing the complexity of the target component and the number of test to be done. To compute the fitness

of a candidate parity generator with N bits of input requires to run 2^{n} test cases. If the 4 bits parity generator evolution requires to run just 16 tests for each fitness evaluation, the 8 bits parity generator requires to run 256 test.

7.2.1 Optimal tuning

Before to be able to evaluate the performance of the architecture, the characterization parameters of the hardware-based Compact Genetic Algorithm need to be tuned. Running a series of test, it has been determined that the optimal step size is 1.57%, for all the case studies. According to the theory on CGAs [15] , that defines the step size as 1/N, where N is the *population size*; the identified optimal step size refers to a simulated population of 64 individuals. Such population size is what has been found to be an optimal population size for the largest part of the cases, in the preliminary analysis of the Genetic Algorithms. The proper value for the mutation threshold has been found to be 3%, exactly as for the software-based execution of the CGA.

7.2.2 4 bits input function

The evolution of a 4 bits parity generator is completed successfully, on average, after 4437 generations, and 22 seconds of execution. The best evolution completed the execution after 3487 generations, while the worst after 6157 generations. Figure 7.1 shows the best performing evolution of a 4 bits parity generator. The blue line shows the fitness of the *best* ever generated, while the red line identifies the fitness value of a random individual sampled with period of 50 generations.

In the case of the 4 bits parity generator the task is still really simple, with flew generations is possible to obtain an high fitness individual, over 80%.

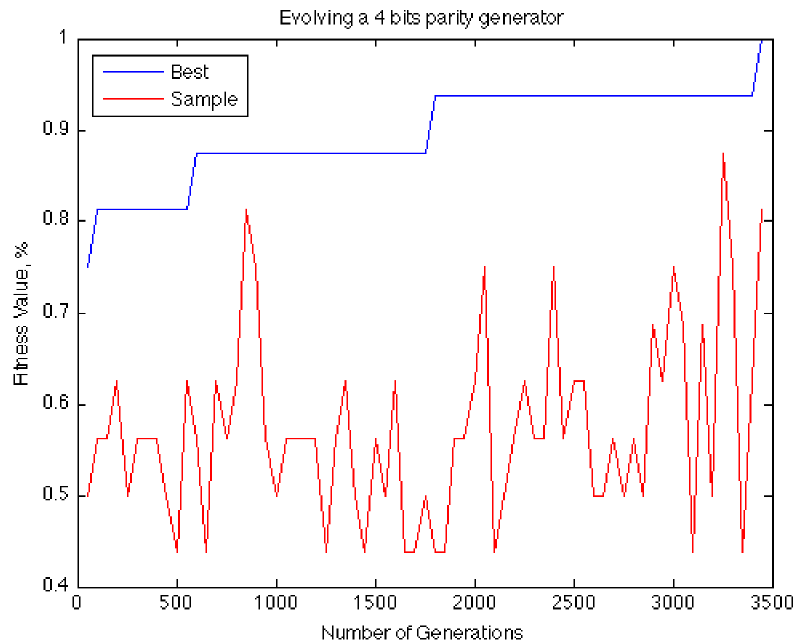


Figure 7.1: Hardware-based CGA evolution 4 bits

7.2.3 5 bits input function

The evolution of a 5 bits parity generator is a slightly more complex taks. On average it is completed successfully after 5772 generations and it requires 25 seconds of execution. The best execution, that is shown in Figure 7.2, completes after 5153 generations, while the worst after 6352. Figure 7.2 shows the blue line that refers to the best solution achieved, and the red line that identifies individuals randomly sampled at every 50 generation. Differently from the case of the 4 bits parity generator, here it becomes possible to see better how updating the Probabilistic Vector allows not only to evolve an individual with the best fitness but increases slowly also the average fitness of the individuals that are generated.

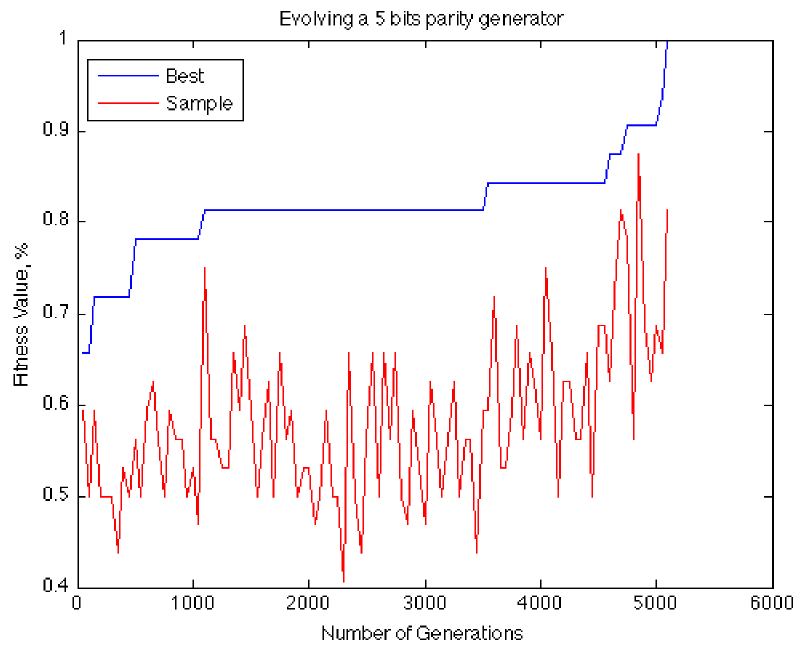


Figure 7.2: Hardware-based CGA evolution 5 bits

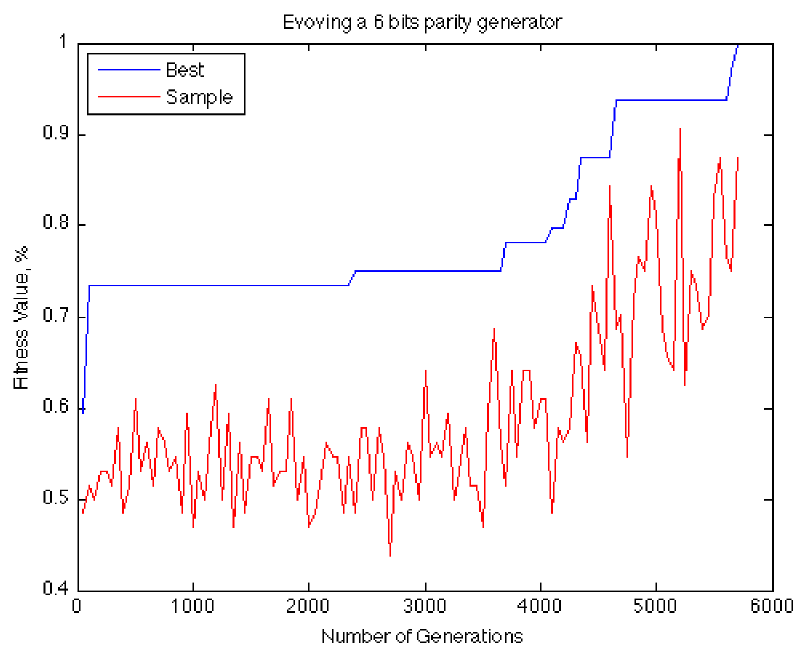


Figure 7.3: Hardware-based CGA evolution 6 bits

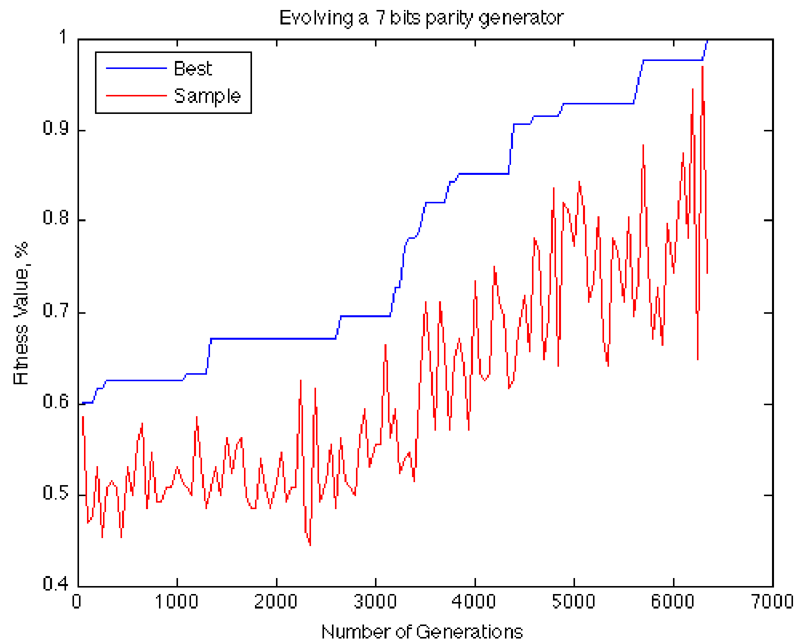


Figure 7.4: Hardware-based CGA evolution 7 bits

7.2.4 6 bits input function

The evolution of a 6 bits parity generator is completed successfully, on average, after 6351 generations and 30 seconds. The best result obtained is successful evolution in 5714 generations, such result is shown in Figure 7.3, while the worst execution completed the evolution in 8431 generations.

7.2.5 7 bits input function

The evolution of a 7 bits parity generator is completed successfully, on average, after 7921 generations and 34 seconds. The best result obtained is successful evolution in 6547 generations, such result is shown in Figure 7.4, while the worst execution completed the evolution in 9848 generations.

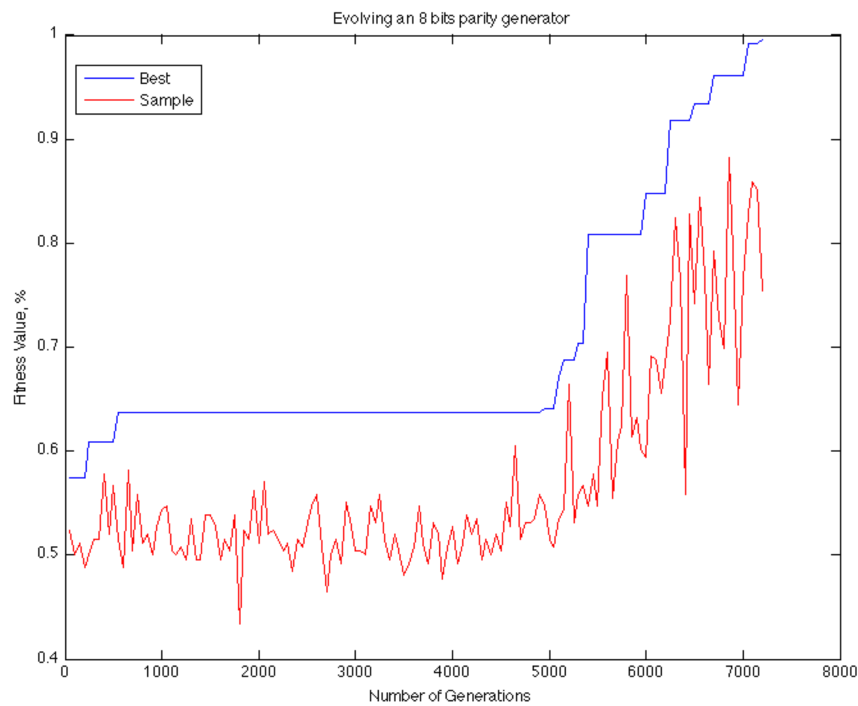


Figure 7.5: Hardware-based CGA evolution 8 bits

7.2.6 8 bits input function

The evolution of a 8 bits parity generator is completed successfully, on average, after 9112 generations and 39 seconds. The best result obtained is successful evolution in 7221 generations, such result is shown in Figure 7.5, while the worst execution completed the evolution in 11506 generations.

7.2.7 Results summary

Table 7.1 shows the experimental results obtained with the architecture described when evolving some parity generators with 4,5,6,7 and 8 bits. As it is possibile to see the evolution has been successful in all the cases.

Figure 7.6 shows all the evolutions plotted on the same graph.

Table 7.1: Hardware-base CGA performance summary

Parity bits	N. Gens: avg	N. Gens: best	N. Gens: worst	fitness
4	4437	3875	4955	100%
5	5772	5153	6532	100%
6	6351	5714	8471	100%
7	7921	6547	9848	100%
8	9112	7221	11506	100%

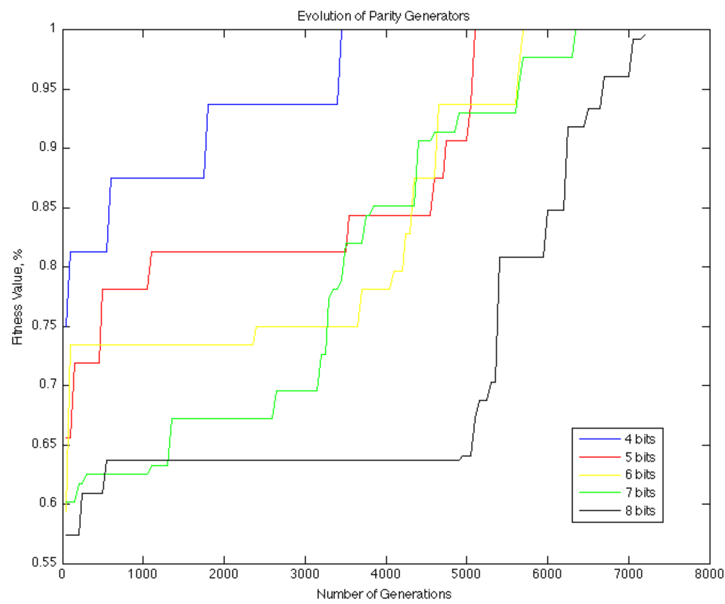


Figure 7.6: Hardware-based CGA evolution

Table 7.2: CGA, hardware-based and software-based comparison

Parity bits	Software N° of Gen.	Hardware N° of Gen.
4	3852	4437
5	5502	5772
6	6932	6351
7	8387	7921
8	6888	9112

7.3 Extrinsic, Intrinsic and Complete evolution comparison

Table 7.2 shows the comparison between the number of generations required by the hardware-based CGA implemented in the architecture and the software-based CGA implemented in the simulation framework. As expected, they are almost the same. More interesting is the analysis of the time performance.

To highlight the validity of the proposed approach and the performance of the architecture implemented, it has been conducted also a comparative analysis of the performance that is possible to obtain from Extrinsic, Intrinsic and Complete evolution. That has been done comparing the time performance measured with the Complete evolvable hardware architecture with the time performance measured performing Extrinsic Evolution with the software-based simulation framework and the same algorithm. The Extrinsic Evolution is executed on a workstation with an *Intel Core 2 Duo* 2.2 ghz and 4 gb of RAM. The performance of the Intrinsic system are not measured but estimated, considering that the hardware-based CGA runs 16.5 times faster than the software-based version on the onboard PPC405. Deployment and Fitness Evaluation in the Complete system and in the Intrin-

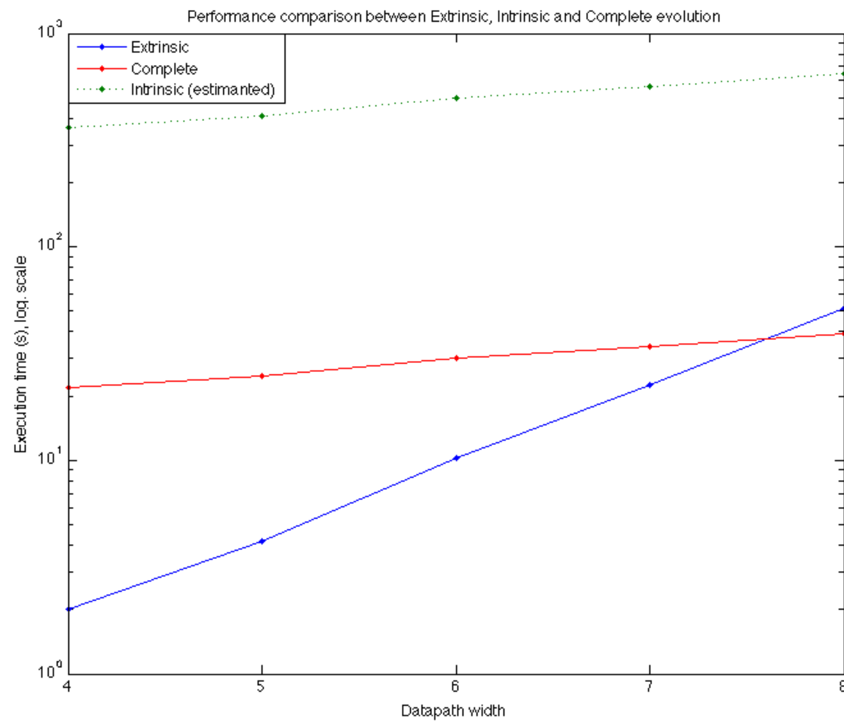


Figure 7.7: Comparison between Extrinsic, Intrinsic and Complete evolution results

sic System are done in the same way by the same components.

The graph in Figure 7.7 shows the comparison between the performance obtained with Extrinsic, Intrinsic and Complete systems. It is a semi-logarithmic graph, to better highlight de difference between the three curves, despite the different order of magnitude between the Intrinsic and the Complete performance. It is possible to see that, in the considered interval, the time complexity of the Intrinsic Evolution and the Complete evolution can be approximated with a polynomial curve, while the Extrinsic Evolution complexity is much more high. It is exponential according to the data-path size, since the number of tests to be executed grow exponentially. Theoretically, also in Intrinsic and Complete evolution the number of tests to be done grows exponentially, but such operation is executed with high hardware

acceleration.

While de difference between Intrinsic and Complete evolution depends just on the speed-up achievable with the hardware acceleration of the Genetic Algorithm, the Figure 7.8 focuses on the difference between Extrinsic and Complete evolution. There it is possible to see clearly the different shape between the two curves, that one representing the Extrinsic evolution and that one representing the Complete evolution. For simple cases, where the numbers of tests to be done to compute the fitness is small, the computation power of the workstation used overtakes limits concerning the simulation of the component. But when just 256 tests need to be done, the Complete EHW system results to be faster than the Extrinsic system. On average, it complete the execution in 39 seconds, while the workstation requires 51.

In the case of the 8 input bits function the achieved speed-up with respect to the software workstation is 1.3x, and is expected to be greater evolving larger components.

7.4 Future Works

Once implemented a working efficient architecture, the next step is to try to expand its capability, partially this has already been done. In particular a series of problems need to be addressed to improve further the performance and to deal with always more complex task:

1. The current EHW module does not support functional decomposition, but it may help a lot to increase convergence rate, by reducing a complex problem to a series of smaller ones.

2. Larger and more recent FPGAs, like Xilinx Virtex 5, can allow better implementations, that exploit higher parallelism. Moreover, Virtex 5

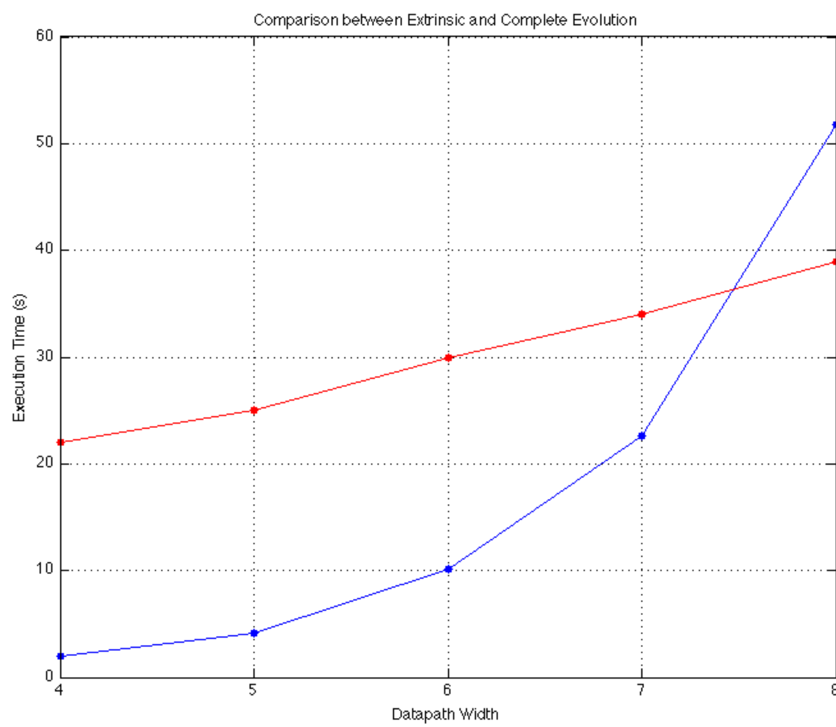


Figure 7.8: Comparison between Extrinsic and Complete evolution results

has Look-Up-Tables with 6 inputs instead of 4, that gives to LUTs higher expressivity.

3. Random Number Generation can be improved as well ,with ad-hoc studies, since it has been identified as a potential critical aspect.

7.5 Not Only Evolvable Hardware

For what concerns the Genetic Algorithm implemented, Hardware-based Genetic Algorithms proposed in literature work with chromosome of 8 [23], 10 [1] or 32 [39] bits. The implemented hardware-based Compact Genetic Algorithm uses chromosomes up to 16384 bits, because the Evolvable Hardware requires such large genotypes. The consideration done analyzing the Extrinsic evolution and the Genetic Algorithms allowed to implement a working Complete evolvable system able to solve complex tasks, as evolving real hardware components. Figure 7.9 shows the implemented hardware Genetic Algorithm, not evolving an hardware component but solving another problem, *OneMax* [81] with 16384 bits. That is a sample execution with no particular tuning, but it shows that the algorithm can be used also to address other problems, besides EHW. It become useful whenever a fitness evaluation could benefit from an hardware acceleration, as the GA.

In this chapter, the experimental results obtained have been presented to show the capability and the performance of the implemented Complete Evolvable Hardware architecture. In Chapter 8 some further theoretical considerations will be done to better highlight the efficiency of the Complete approach to Evolvable Hardware.

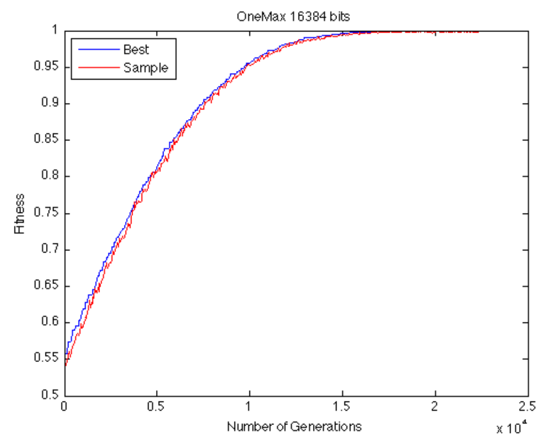


Figure 7.9: The implemented CGA solving 16384 bits OneMax problem

Chapter 8

Conclusions

The conclusions of this thesis are illustrated in this chapter. In the first Section, the results achieved with the systems implemented are presented to highlight the efficiency of the Complete Evolvable Hardware architecture developed. In the second sections, preliminary results obtained by further experiments with a larger Evolvable Component are briefly proposed. The third section summarizes some concluding remarks.

8.1 Results Achieved

In the introduction, the three main issues of Evolvable Hardware (EHW) have been presented: *Scalability*, *Fitness Evaluation Complexity* and *Behavioural and Structural Analysis capability*. It has been argued that Fitness Evaluation Complexity slowdowns the performance of Extrinsic Evolvable Systems and such limit is overcome in Intrinsic Systems, thanks to an hardware implementation of the individual. But Intrinsic system are bottlenecked by multidevice communication, or less efficient for what concerns the execution of the Genetic Algorithm if it runs on an embedded architecture. The proposed Complete Evolvable Hardware System, based on a System-on-Chip architecture address all these issues and it results to be more efficient

than Extrinsic or Intrinsic Implementation, especially for the evolution of large components. The next subsection analyze characteristics and the performance to prove the validity of the proposed System.

8.1.1 Extrinsic Evolution System

The considered Extrinsic Evolvable System is that one implemented with the software framework and executed on an Intel Core2Duo at 2.20 GHz. It has been able to evolve successfully parity generators with 4 and 8 bits of inputs. It has shown limits evolving multi-outputs function that likely depends on the structure of the evolvable component. But for what concerns the performance, results obtained show the exponential growth of time required by the evolution, due to the lack of efficiency of the approach based on a simulation of the hardware components. The Extrinsic approach results to be not suitable for the implementation of real hardware components, when the data path size increases. Moreover it is necessary to consider that, whenever the goal is to have an autonomous evolvable system Extrinsic evolution cannot be used.

8.1.2 Intrinsic Evolution System

The considered Intrinsic system is an embedded system that uses for the deployment and testing of the individuals the same core used by the Complete System, but uses a software-based implementation of the GA. The limits identified in the Extrinsic System can be overcome by the Intrinsic system that implements the individuals and run the same Genetic Algorithms on the onboard PPC405. An high speed-up in the evaluation phase makes the system more scalable, but there is no global speed-up since the execution of the Genetic operators become more slow.

Considering the case of the 8 bits parity generator, despite fitness computation complexity, the Extrinsic Evolution with the software framework

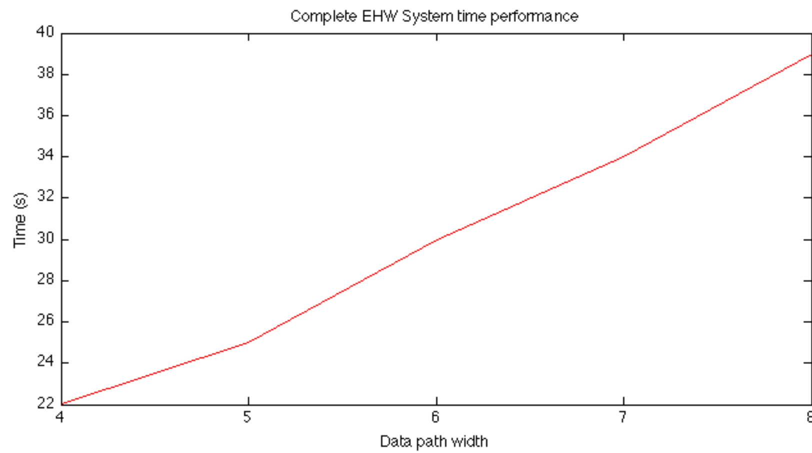


Figure 8.1: Evolution time with the implemented System

is still more than 10 times faster.

8.1.3 Proposed System

Thanks to an hardware-based implementation of the Genetic Algorithm, the proposed SoC architecture of the Complete Evolution of hardware components addresses all the three presented issues and can be more efficient than Extrinsic and Intrinsic Systems. The Figure 8.1 shows the time performance obtained evolving parity generators with 4,5,6,7 and 8 bits of input. But interesting may be also to study what could happen with a larger data-path.

Figure 8.2 highlights the efficiency of the Complete evolution with respect to the Extrinsic Evolution. Since the evolvable component used has a maximum data-path up to 8 bits, a lower bound of the performance cases with more than 8 bits have been estimated considering a linear increase of the number of generations required.

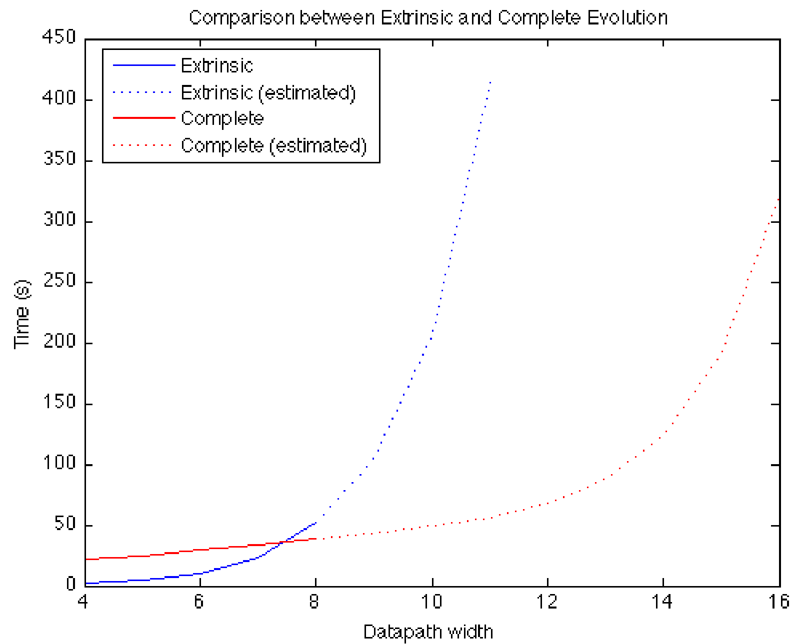


Figure 8.2: Estimation of larger system performance

8.2 Further Experiments

Unfortunately to evolve components with a larger data path do not require just to execute more tests to evaluate the fitness, but it requires to have a more complex evolvable component with a larger genotype. Some experiments has been conducted on an architecture with a data path up to 32 bits.

Figure 8.3 shows such larger evolvable component. It is composed of 16 8-bits-datapath components. Its genotype is 16 times larger than that one of the small system.

The only component that has been evolved with this architecture, to shows validity of the proposed implementation, is an 8 bits parity generator. Figure 8.4 shows the result obtained in the evolution process. The evolution is completed after 67900 and 1358s of execution (22 minutes). A software-based implementation of the same algorithm, executed on the

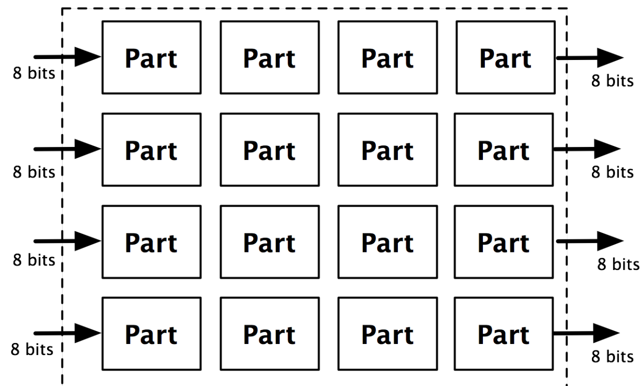


Figure 8.3: Extended Evolvable Component

Table 8.1: Summary of achieved performance

Architecture	Algorithm	Genotype	tests	generations	time
Std	CGA	1024bits	256	9112	39s
Ext	CGA	16384bits	256	67900	1358 s

onboard PPC405, would require 50 times more time to complete. Such evolution has been performed enabling *High Selection Pressure* and *Additional Mutation*.

Table 8.1 shows the performance achieved with both the architectures used and the hardware-based CGA.

To be able to evolve other more complex hardware component requires to have an FPGA with more hardware resources, in order to have more degree of freedom in implementing a Genetic Algorithm with a more powerful search capability , able to accomplish much more complex tasks.

8.3 Concluding Remarks

The proposed Complete Evolvable Hardware System, addressing all the EHW issues, allowed to evolve efficiently hardware components. In

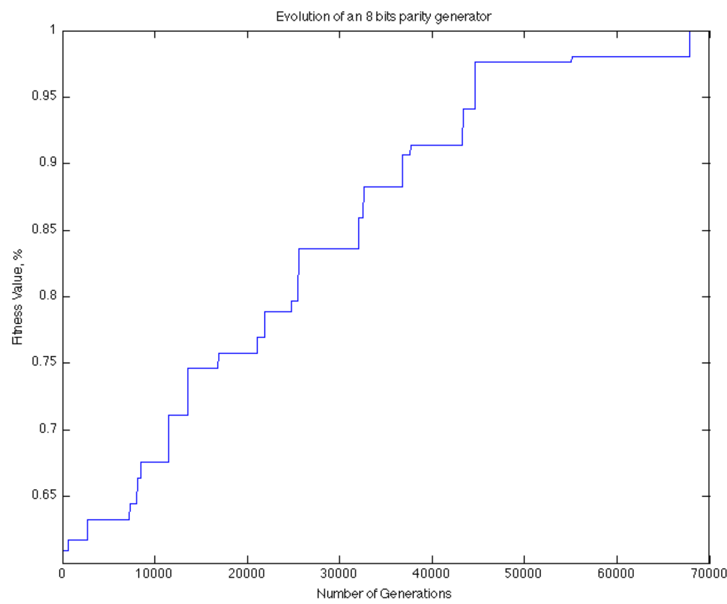


Figure 8.4: Evolution of an 8 bits parity generator, with the Extended Architecture

particular, *Fitness Computation Complexity* is handled with the parallel hardware-based evaluation of the individuals. It has also been shown, in the sections above, that the system *Scales* better than Extrinsic and Intrinsic Systems. *Behavioural and Structural Analysis* of the evolved circuit can be done exploiting the software framework that allows to simulate the circuit.

Bibliography

[1] Stephen D. Scott, Ashok Samal, and Shared Seth. Hga: a hardware-based genetic algorithm. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, FPGA '95, pages 53–59, New York, NY, USA, 1995. ACM.

[2] G. Kooner, Shawki Areibi, and Medhat Moussa. A genetic hardware accelerator for vlsi circuit partitioning. *International Journal of Computers and Their Applications*, 12:163–280, July 2007.

[3] Davide Candiloro. *Thesis: Management and analysis of bitstreams generators for Xilinx FPGAs*. Politecnico di Milano, Italy, 2008.

[4] Fabio Cancare, Marco Castagna, Matteo Renesto, and Donatella Sciuto. A highly parallel fpga-based evolvable hardware architecture. *Workshop PARAFPGA, in Parallel Computing, Lyon France*, Sept. 2009.

[5] David B. Fogel. Applying fogel and burgin's 'competitive goal-seeking through evolutionary programming' to coordination, trust, and bargaining games, 2000.

[6] David B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence (IEEE Press Series on Computational Intelligence)*. Wiley-IEEE Press, 2006.

[7] IEEE Intelligent Systems staff. Genetic programming. *IEEE Intelligent Systems*, 15:74–84, May 2000.

[8] Peter J. Angeline. Evolutionary optimization versus particle swarm optimization: Philosophy and performance differences. In *Proceedings of the 7th International Conference on Evolutionary Programming VII*, EP '98, pages 601–610, London, UK, 1998. Springer-Verlag.

[9] John H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[10] Thomas Back, Frank Hoffmeister, and Hans paul Schwefel. A survey of evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1991.

[11] T. Kalganova C. Lambert and E. Stomeo. Fpga-based system for evolvable hardware. *Transactions on Engineering, Computing and Technology*, 2006.

[12] J. R. Reza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[13] Deb Goldberg and Krob. Messy genetic algorithms: motivation, analysis and first results. In *Complex Systems*, volume 3, pages 493–530, 1989.

[14] Fernando G-Lobo andGeorges R. Harik. *Technical report:Extended Compact Genetic Algorithm in C++*. Illinois Genetic Algorithm Lab, 6 1999.

[15] D.E. Goldberg G.R. Harik, F.G. Lobo. The compact genetic algorithm. *Evolutionary Computation, IEEE Transactions*, 3:287–297, 1999.

[16] Jim Torresen. An evolvable hardware tutorial. In *In Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL'2004*, pages 821–830, 2004.

[17] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, and T. Furuya. Evolvable hardware with genetic learning. In *Proceedings of Simulated Adaptive behavior*. MIT Press, 1992.

[18] M. Garvie and A. Thompson. Scrubbing away transients and jiggling around the permanent: Long survival of FPGA systems through evolutionary self-repair. In C. Metra, R. Leveugle, M. Nicolaidis, and J.P. Teixeira, editors, *Proc. 10th IEEE Intl. On-Line Testing Symposium*, volume 2606 of *LNCS*, pages 155–160. IEEE Computer Society, 2004.

[19] Nicholas J. Macias. The pig paradigm: The design and use of a massively parallel fine grained self-reconfigurable infinitely scalable architecture. In *Proceedings of the 1st NASA/DOD workshop on Evolvable Hardware*, EH '99, pages 175–, Washington, DC, USA, 1999. IEEE Computer Society.

[20] Paul Layzell. A new research tool for intrinsic hardware evolution. In *Lecture Notes in Computer Science*, pages 47–56. Springer-Verlag, 1998.

[21] Julian F. Miller and Keith Downing. Evolution in materio: Looking beyond the silicon box. In *Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware (EH'02)*, EH '02, pages 167–, Washington, DC, USA, 2002. IEEE Computer Society.

[22] S. Harding and J. Miller. Evolution in materio: Looking beyond the silicon box. In *Evolution in materio: A tone discriminator in liquid crystal*, volume 2, pages 1800–1807, Washington, DC, USA, 2004. Proceedings of the Congress on Evolutionary Computation 2004.

[23] Ricardo Zebulum, Didier Keymeulen, Raoul Tawel, Taher Daud, and Anil Thakoor. Reconfigurable vlsi architectures for evolvable hardware: from experimental field programmable transistor arrays to

evolution-oriented chips. *IEEE Trans. Very Large Scale Integr. Syst.*, 9:227–233, February 2001.

[24] Xilinx. *ISE Design Suite Software Manuals and Help*. Xilinx Corp., 20 10.

[25] Xilinx. *Application Note: Two flows for partial reconfiguration: Module based or difference based*. Xilinx Corp., 9 2004.

[26] Peng Ke, Xin Nie, and Zhichao Cao. A generic architecture of complete intrinsic evolvable digital circuits. In *Proceedings of the 2010 3rd International Symposium on Knowledge Acquisition and Modeling*, KAM, pages 379–382, Wuhan,China, 2010. IEEE Computer Society.

[27] P. Haddows and G. Tufte. Evolving a robot controller in hardware. In *Norwegian Computer Science Conference (NIK99)*, pages 141–150, 1999.

[28] Naveed Arshad. Automated dynamic reconfiguration using ai planning. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 402–405, Washington, DC, USA, 2004. IEEE Computer Society.

[29] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[30] B. L. Miller and D. Goldberg. *Technical report:Genetic Algorithm, Tournament Selection and the Effect of Noise*. Illinois Genetic Algorithm Lab, 1995.

[31] Dirk Thierens and David E. Goldberg. Mixing in genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 38–47, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[32] G. Harik. *Linkage Learning via Probabilistic Modeling in the ECGA*. Illinois Genetic Algorithm Lab, 1999.

[33] J. Horn and N. Nafpliotis. *Technical report:Multiobjective optimization using the niched Pareto genetic algorithm*. Illinois Genetic Algorithm Lab, 1993.

[34] N. Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evol. Comput.*, 2:221–248, September 1994.

[35] Kalyanmoy Deb and Tushar Goel. Controlled elitist non-dominated sorting genetic algorithms for better convergence. In *Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization*, EMO '01, pages 67–81, London, UK, 2001. Springer-Verlag.

[36] Maneeratana Kuntinee, Boonlong Kittipong, and Chaiyaratana Nachol. Co-operative co-evolutionary genetic algorithms for multi-objective topology design. *Computer-Aided Design & Applications*, 2:437–496, 2005.

[37] Chatchawit Aporntewan and Prabhas Chongstitvatana. A hardware implementation of the compact genetic algorithm. In *IEEE Congress on Evolutionary Computation*, pages 624–629, 2001.

[38] Andy Miller and Micheal Gulotta. *Application Note: PN Generators Using the SRL Macro*. Xilinx Corp., 2004.

[39] Yutana Jewajinda and Prabhas Chongstitvatana. Fpga implementation of a cellular compact genetic algorithm. In *Proceedings of the 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 385–390, Washington, DC, USA, 2008. IEEE Computer Society.

[40] E. Cantu-Paz. *Efficient and accurate parallel genetic algorithms*. Boston, MA:Kluwer Academic Publisher, 2000.

[41] Tetsuya Higuchi, Tatsuya Niwa, Toshio Tanaka, Hitoshi Iba, Hugo de Garis, and Tatsumi Furuya. Evolving hardware with genetic learning: a first step towards building a darwin machine. In *Proceedings of the second international conference on From animals to animats 2 : simulation of adaptive behavior: simulation of adaptive behavior*, pages 417–424, Cambridge, MA, USA, 1993. MIT Press.

[42] Tetsuya Higuchi, Masaya Iwata, Isamu Kajitani, Hitoshi Iba, Yuji Hirao, Tatsumi Furuya, and Bernard Manderick. Evolvable hardware and its applications to pattern recognition and fault-tolerant systems. In *Papers from an international workshop on Towards Evolvable Hardware, The Evolutionary Engineering Approach*, pages 118–135, London, UK, 1996. Springer-Verlag.

[43] Tetsuya Higuchi, Hitoshi Iba, and Bernard Manderick. Massively parallel artificial intelligence. chapter Evolvable hardware, pages 398–421. MIT Press, Cambridge, MA, USA, 1994.

[44] Xin Yao and Tetsuya Higuchi. Promises and challenges of evolvable hardware. In *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware*, pages 55–78, London, UK, 1996. Springer-Verlag.

[45] Masaya Iwata, Isamu Kajitani, Hitoshi Yamada, Hitoshi Iba, and Tetsuya Higuchi. A pattern recognition system using evolvable hardware. In *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, PPSN IV, pages 761–770, London, UK, 1996. Springer-Verlag.

[46] Masahiro Murakawa, Shuji Yoshizawa, Isamu Kajitani, Tatsumi Furuya, Masaya Iwata, and Tetsuya Higuchi. Hardware evolution at function level. In *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, PPSN IV, pages 62–71, London, UK, 1996. Springer-Verlag.

[47] Julian Francis Miller and Simon L. Harding. Cartesian genetic programming. In *Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, GECCO '08, pages 2701–2726, New York, NY, USA, 2008. ACM.

[48] Tatiana Kalganova. Bidirectional incremental evolution in extrinsic evolvable hardware. In *Proceedings of the 2nd NASA/DoD workshop on Evolvable Hardware*, pages 65–, Washington, DC, USA, 2000. IEEE Computer Society.

[49] Venkatesh Katari, Suresh Ch, Ra Satapathy, Member Ieee, Jvr Murthy, and Pvgd Prasad Reddy. Hybridized improved genetic algorithm with variable length chromosome for image clustering, 2007.

[50] A. Antola, M. Castagna, P. Gotti, and M.D. Santambrogio. Evolvable hardware: A functional level evolution framework based on impulse c. In *Proceedings of the 2007 International Conference on Engineering of Reconfigurable Systems and Algorithms*, ERSA, pages 216–219, Las Vegas, Nevada, USA, 2007. ERSA.

[51] Claude E. Shannon. *Claude Elwood Shannon: collected papers*. IEEE Press, Piscataway, NJ, USA, 1993.

[52] Adrian Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware*, pages 390–405, London, UK, 1996. Springer-Verlag.

[53] Adrian Thompson. On the automatic design of robust electronics through artificial evolution. In *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware*, pages 13–24, London, UK, 1998. Springer-Verlag.

[54] Adrian Thompson and Paul J. Layzell. Evolution of robustness in an electronics design. In *Proceedings of the Third International Conference on Evolvable Systems: From Biology to Hardware*, ICES '00, pages 218–228, London, UK, 2000. Springer-Verlag.

[55] Didier Keymeulen, Masaya Iwata, Yasuo Kuniyoshi, and Tetsuya Higuchi. Online evolution for a self-adapting robotic navigation system using evolvable hardware. *Artif. Life*, 4:359–393, October 1998.

[56] Gordon Hollingworth, Steve Smith, and Andy Tyrrell. Safe intrinsic evolution of virtex devices. In *Proceedings of the 2nd NASA/DoD workshop on Evolvable Hardware*, pages 195–, Washington, DC, USA, 2000. IEEE Computer Society.

[57] Lukáš Sekanina. Virtual reconfigurable circuits for real-world applications of evolvable hardware. In *Proceedings of the 5th international conference on Evolvable systems: from biology to hardware*, ICES'03, pages 186–197, Berlin, Heidelberg, 2003. Springer-Verlag.

[58] PC Haddow and G. Tufte. Evolution of robustness in an electronics design. In *An evolvable hardware FPGA for adaptive hardware. In: Proceedings of the Congress on Evolutionary Computation*, CEC00, pages 553–560, Los Alamitos, CA, 2001. IEEE.

[59] S. A. Guccione, Delon Levi, and P. Sundararajan. *Jbits: a java-based interface for reconfigurable computing*. MAPLD, 1999.

[60] Delon Levi and Steven A. Guccione. Geneticfpga: Evolving stable circuits on mainstream fpga devices. In *Proceedings of the 1st NASA/DOD*

workshop on Evolvable Hardware, EH '99, pages 12–, Washington, DC, USA, 1999. IEEE Computer Society.

[61] Xilinx. *User Guide: Virtex-II Pro and Virtex-II Pro X*. Xilinx Corp., 11 2007.

[62] T. Kalganova C. Lambert and E. Stomeo. Fpga-based system for evolvable hardware. *International Journal of Electrical, Computer and System Engineering*, 3:62–68, 1 2009.

[63] A. Upegui and E. Sanchez. Evolving hardware by dynamically reconfiguring Xilinx FPGAs. In J.M. Moreno et al., editor, *Evolvable Systems: From Biology to Hardware*, volume 3637 of *LNCS*, pages 56–65, Berlin Heidelberg, 2005. Springer-Verlag.

[64] Fabio Cancare, M. D. Santambrogio, and D. Sciuto. A direct bitstream manipulation approach for virtex4-based evolvable systems. In *IEEE International Symposium on Circuits and Systems, Paris*, pages 853–856. IEEE Computer Society, 2010.

[65] A. Upegui. *Dynamically Reconfigurable Bio-inspired Hardware, Phd Thesis*. EPFL, Lausanne, CH, 2006.

[66] Pauline C. Haddow and Gunnar Tufte. Bridging the genotype-phenotype mapping for digital fpgas. In *Proceedings of the The 3rd NASA/DoD Workshop on Evolvable Hardware*, EH '01, pages 109–, Washington, DC, USA, 2001. IEEE Computer Society.

[67] Xilinx. *Product Specification: Virtex-4 FPGA Data Sheet: DC and Switching Characteristics*. Xilinx Corp., 9 2009.

[68] Xilinx. *Product Specification: Virtex-4 Family Overview*. Xilinx Corp., 8 2010.

[69] Xilinx. *Datasheet: LogiCORE IP XPS HWICAP v5.00a*. Xilinx Corp., 2010.

[70] C. Schuck, M. Kuhnle, M. Hubner, and J. Becker. A framework for dynamic 2d placement on fpgas. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –7, april 2008.

[71] Xilinx. *User Guide: Virtex-4*. Xilinx Corp., 12 2008.

[72] Xilinx. *FPGA Editor Guide*. Xilinx Corp., 20 10.

[73] Alan Piszcz and Terence Soule. Genetic programming: optimal population sizes for varying complexity problems. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, pages 953–954, New York, NY, USA, 2006. ACM.

[74] Chang Wook Ahn and R.S. Ramakrishna. Elitism-based compact genetic algorithms. *Evolutionary Computation, IEEE Transactions*, 7:367–385, 2003.

[75] Kumara Sastry and David E. Goldberg. On extended compact genetic algorithm. Technical report, GECCO-2000, LATE BREAKING PAPERS, GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, 2000.

[76] Xilinx. *Datasheet: Seriel communication port*. Xilinx Corp.

[77] Xilinx. *Datasheet:Processor Local Bus (PLB)*. Xilinx Corp.

[78] Y. Jewajinda and P. Chongstitvatana. A cooperative approach to compact genetic algorithm for evolvable hardware. *IEEE Congress on Evolutionary Computation*, pages 624–629, 2006.

[79] Jennifer L. Brady. *Thesis: Limitations of a True Random Number Generator in a FPGA*. Air Force Institute of Technology,Wright-Patterson Air Force Base, Ohio, 2008.

[80] Texas Instruments. *Datasheet: 9-Bit Parity Generators/Checkers (Rev. C)*. Texas Instruments Corp., 1994.

[81] Bulent Buyukbozkirli. *Modeling genetic algorithm dynamics for one-max and deceptive functions*. PhD thesis, East Lansing, MI, USA, 2004. AAI3145986.

Personal

- Matteo Renesto

Education

- M.S. Computer Engineering, Politecnico di Milano, 2008 - present.
- M.S. Computer Science, University of Illinois at Chicago, 2009 - 2011.
- B.S. Computer Engineering, Politecnico di Milano, 2005 - 2008.

Employment

- Spring, 2009. Laboratory Tutor for Software Engineering 1. Prof. C.Ghezzi, at Politecnico di Milano.

Publications

- F. Cancarè, M. Castagna, M. Renesto and D. Sciuto - A Highly Parallel FPGA-based Evolvable Hardware Architecture - *Advances in Parallel Computing*, Volume 19 - 2010. Pages: 608-615.