DYNAMICALLY RECONFIGURABLE BIO-INSPIRED HARDWARE

THÈSE N^{O} 3632 (2006)

PRÉSENTÉE LE 13 OCTOBRE 2006

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Groupe Sanchez

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Andres Emilio UPEGUI POSADA

Ingénieur en électronique diplômé de l'Université Pontifica Bolivariana de Medellín, Colombie
de nationalité colombienne

acceptée sur proposition du jury:

Prof. C. Petitpierre, président du jury
Prof. E. Sanchez, directeur de thèse
Prof. P. Ienne, rapporteur
Prof. J. M. Moreno, rapporteur
Prof. X. Yao, rapporteur

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2006

Abstract

During the last several years, reconfigurable computing devices have experienced an impressive development in their resource availability, speed, and configurability. Currently, commercial FPGAs offer the possibility of self-reconfiguring by partially modifying their configuration bitstream, providing high architectural flexibility, while guaranteeing high performance. These configurability features have received special interest from computer architects: one can find several reconfigurable coprocessor architectures for cryptographic algorithms, image processing, automotive applications, and different general purpose functions.

On the other hand we have bio-inspired hardware, a large research field taking inspiration from living beings in order to design hardware systems, which includes diverse topics: evolvable hardware, neural hardware, cellular automata, and fuzzy hardware, among others. Living beings are well known for their high adaptability to environmental changes, featuring very flexible adaptations at several levels. Bio-inspired hardware systems require such flexibility to be provided by the hardware platform on which the system is implemented. In general, bio-inspired hardware has been implemented on both custom and commercial hardware platforms. These custom platforms are specifically designed for supporting bio-inspired hardware systems, typically featuring special cellular architectures and enhanced reconfigurability capabilities; an example is their partial and dynamic reconfigurability. These aspects are very well appreciated for providing the performance and the high architectural flexibility required by bio-inspired systems. However, the availability and the very high costs of such custom devices make them only accessible to a very few research groups. Even though some commercial FPGAs provide enhanced reconfigurability features such as partial and dynamic reconfiguration, their utilization is still in its early stages and they are not well supported by FPGA vendors, thus making their use difficult to include in existing bio-inspired systems.

In this thesis, I present a set of architectures, techniques, and methodologies for benefiting from the configurability advantages of current commercial FPGAs in the design of bio-inspired hardware systems. Among the presented architectures there are neural networks, spiking neuron models, fuzzy systems, cellular automata and random boolean networks. For these architectures, I propose several adaptation techniques for parametric and topological adaptation, such as hebbian learning, evolutionary and co-evolutionary algorithms, and particle swarm optimization. Finally, as case study I consider the implementation of bio-inspired hardware systems in two platforms: YaMoR (Yet another Modular Robot) and ROPES (Reconfigurable Object for Pervasive Systems); the development of both platforms having been co-supervised in the framework of this thesis.

Keywords: bio-inspired hardware, evolvable hardware, reconfigurable computing, dynamic reconfiguration, partial reconfiguration, adaptation, evolutionary algorithms, neural networks.

Résumé

Depuis quelques années, la quantité de ressources, la vitesse et les capacités de reconfiguration des dispositifs de calcul programmables ont augmenté de manière importante. Actuellement, certains FPGAs commerciaux offrent la possibilité de se reconfigurer en modifiant partiellement les données de leur configuration, ils fournissent ainsi une grande flexibilité architecturale tout en garantissant de bonnes performances. Ces caractéristiques de configurabilité ont attiré l'attention des concepteurs de matériel électronique: on peut trouver plusieurs architectures de coprocesseurs reconfigurables, certaines très spécifiques utilisées par exemple dans les applications cryptographiques, le traitement d'images ou dans le domaine automobile, mais aussi des architectures moins spécialisées pouvant être mises à profit dans de nombreuses applications générales.

D'autre part, il existe ce qu'on appelle le hardware bio-inspiré, un large domaine de recherche incluant plusieurs sous-catégories: hardware évolutif, hardware neuronal, automates cellulaires, hardware à logique floue, entre autres. Les êtres vivants sont remarquables par leur capacité d'adaptation aux changements environnementaux et leur très haute flexibilité à plusieurs niveaux. Les systèmes matériels bio-inspirés ont besoin qu'une telle flexibilité soit fournie par la plateforme hardware sur laquelle ils sont implémentés. En général, les systèmes hardware bio-inspirés ont été implémentés sur des plateformes hardware commerciales ou faites sur mesure. Ces dernières ont été conçues pour supporter des systèmes hardware bio-inspirés en fournissant des capacités de reconfiguration augmentées et des architectures modulaires spécialisées. Ces aspects sont très appréciés pour fournir les performances et la flexibilité nécessaires aux systèmes bio-inspirés. Même si quelques FPGAs commerciaux offrent certaines possibilités de configurabilité évoluée comme la reconfiguration partielle et dynamique, leur utilisation n'est pas encore très bien prise en charge par les fabricants de FPGAs, ce qui les rend difficiles à utiliser pour l'implémentation des systèmes bio-inspirés existants.

Dans cette thèse, j'expose un ensemble d'architectures, de techniques et de méthodologies permettant de bénéficier des avantages de configurabilité offerts par les FPGA commerciaux actuels pour la conception de systèmes hardware bio-inspirés. Parmi ces architectures je présente des réseaux neuronaux, des modèles de neurones à impulsion, des systèmes flous, des automates cellulaires et des réseaux booléens aléatoires. Je propose aussi, dans le cadre de l'adaptation paramétrique et topologique de ces architectures, plusieurs techniques d'adaptabilité comme une forme d'apprentissage Hebbien, des algorithmes évolutifs et coévolutifs, ainsi que des algorithmes d'optimisation par essaim de particules. Finalement, comme exemples d'application, je présente quelques implémentations de ces

techniques bio-inspirées sur deux plateformes dont le développement a été co-supervisé dans le cadre de cette thèse: YaMoR (Yet another Modular Robot) et ROPES (Reconfigurable Object for Pervasive Systems).

Mots-clés: hardware bio-inspiré, hardware évolutif, dispositifs de calcul programmables, reconfiguration dynamique, reconfiguration partielle, adaptation, algorithmes évolutifs, réseaux neuronaux.

Acknowledgments

The acknowledgements! maybe the most difficult section to write in a dissertation! In a thesis many people machinate to push the author towards his final goal. I would like to thank all these people for plotting to do so. But, how to summarize in a couple of pages everyone who took part in this conspiracy? How to thank them all without forgetting someone? And, how to fully express in a single line the gratitude that I have to each of them?

It makes sense to begin with Eduardo Sanchez, my thesis director, who, more than a boss, has been a friend during these four years. I would like to have his energy, being the dean of an institute at HEIG-VD and professor at EPFL is not enough for him. He is also a semi-professional squash player, a renowned DJ, a music lover (with an impressive collection of salsa CDs), a BBQ chef, the colombian scientific consul in Switzerland, and last but not least, recently the happiest grandfather. I am very grateful to him for allowing me to do this thesis, for his friendship, and for being so cool!

I am grateful to Carlos Andres, who guided me during the first year of my thesis. The numerous discussions about diverse topics were very helpful for defining the thesis research line. I am also very grateful to him and Sandra because they constituted my adoptive family at my arrival; they really made me feel like being at home.

Having a good group when doing a thesis is very important. I consider myself very lucky for finding a PhD position at the Logic Systems Laboratory (LSL), which was at this time under the direction of Professor Daniel Mange. I would like to express my sincere appreciation to Daniel for his warm reception upon my arrival. Daniel, more than anyone, understands how to do an excellent work and still provide a friendly and convivial atmosphere.

Fate wanted that during my stay at the LSL, the laboratory closed its doors. However, the spirit of the LSL is far of being dead and remains unperturbable thanks to people like Daniel, who in spite of his retirement is still there; Marlyse, who is one of the key persons at the laboratory, able to solve any complicated bureaucracy in an impressive efficient and simple way; and Auke, Gianluca, and André, who are the ones in charge of continuing the quest toward bio-inspired hardware systems (each one in his style) at the EPFL by leading the BIRG and the CARG groups.

At the LSL I also met other people that I learned to appreciate. When I began my thesis, I shared the office with Carlos and Fabien, the fastest french speaker I have met and the very last FPGA hardware developer of a large dynasty at the LSL. I am thankful to him for his

support in several hardware aspects and for not phoning me drunk at midnight. Some months later Alessandro replaced him, the first PhD student at the BIRG and the most skillful system administrator, who I thank for being watchful of keeping my computer free of virus, for his support in diverse hardware and software aspects, for Mozart, and for his psychological tests. Then, Jonas completed the quartet: the dynamical-systems guy. We never succeeded to collaborate on non-linear oscillators in FPGAs; however, we managed to agree in the office temperature (regrettably for Carlos).

Then the CARG was created, and with it two new PhD students arrived: Pierre-Andre, to whom I am grateful for his support on several hardware aspects, ranging from soldering shortcuts to PCI buses; and Joel, to whom I am grateful for ~~writing~~ helping me to write the french version of the abstract of the thesis.

The BIRG began growing and engaged Ludovic, mainly thanks to his impressive CV: satellite administrator. I thank him (or blame) for the Kwak (a belgian beer), I'm still suffering the consequences! Then, Sarah increased in a 100% the female population in the lab, and introduced the first drummer humanoid robot in the lab for replacing Yann (our obsolete human drummer). Alexander was the last one beginning a thesis before my departure, the proof that not every German loves beer. Paradoxically, he was the only one that I succeeded to convince playing rugby.

With the BIRG, there were also the people from Cyberbotics, Olivier and Yvan, also known as "the bug exterminator", who kindly helped Joel to write my thesis abstract in french, and with whom I have shared the office for the last year.

What would be the LSL without Chico? Certainly quite boring. I would like to thank him for the ”jeux de mots”, for the dead bird in the freezer, and for the diverse environmental activities that we accomplished together ranging from the rescue of a mother duck with its ten babies, to trying to save the plant in the cafeteria from the conspiracy fabricated by most of the aforementioned individuals

The last presented thesis before this one was that of Yann, the obsolete human drummer I mentioned before. His departure was a heavy loss to the lab. He was always among the promoters of numerous activities carried out in the lab. I am really glad of having the opportunity to continue working with him for the next years.

A significative part of the work reported in this thesis was possible due to several persons who have worked on smaller projects, I would like to thank them. These projects leaded to very good results in some cases, while others have been less successful but not less important, since research is a constant process of trial and error. For their engagement in their work, I am grateful to Rico, Elmar, Cyril, Kevin, Jerome, Adamo, Edouard, and Guillaume, who worked in the YaMoR project and did an excellent work. I am grateful also to Alexandre at the HEIG-VD, who designed the ROPES platform, and to Arnaud and Valentin, who were its main users. I am grateful to Gregory and Barthelemy, who had to deal with lots of *fuzzy* documentation about Xilinx design flows, to Jorge, who played around with swarm

simulations, to Christian, who had a hard time trying to make learn a spiking neural network, and to Nicolas, who had a hard time with Linux on an FPGA.

I am grateful to Professors Xin Yao, Juan Manuel Moreno, and Paolo Ienne, the reviewers of this thesis, who spent a part of their valuable time during the summer for reading this document. I am grateful also to Professor Claude Petitpierre who accepted being president of the jury.

Even though money is not happiness, it is important. I am grateful to the Swiss National Science Foundation and the EPFL that supported this project

I am grateful to Jose Diego, who survived to listening twice the presentation for my private thesis defense, and from who I received very helpful feedback. As PhD students, we have shared a lot of experiences and coffees. I am also grateful to Nathan, who carefully read this thesis making valuable corrections and comments.

The thesis was read also by Adona and Jose (not José), our neighbors, who more than neighbors or friends have become our close family in Switzerland. I am very grateful to them for reading this thesis, for the numerous bottles of milk that we have borrowed from them, for all the good moments that we have shared, and for being our family.

I am grateful to the so-called "diagonal group" formed by Ricardo, Andres, Eduardo, Jose Diego, Carlos, and Jose. From them, I have received very useful feedbacks and critics during my thesis, concerning several methodological aspects and some feedbacks for several papers. I thank Andres and Eduardo for trusting on me for my next step after this thesis, and Ricardo for his backup support in Venice.

Living far from one's family is not easy and, in these cases, expatriated people tend to form groups where the social, political, and cultural aspects are left behind, and they have a single thing in common: the search for a new identity. I consider myself very lucky for having found such a group here. Even though the group was very well consolidated since before, they warmly hosted us and we have shared a lot of good and hard moments: Christmas, births, birthdays, baptisms, marriages, hospitalizations, departures, etc. I have already mentioned several of them and I am not going to list them all here, since it would be either very long or unjust. However, they know who they are, and I am really grateful to them for these moments.

I am grateful to the LUC Rugby. Since my arrival in Lausanne the team has been providing the necessary framework for maintaining an equilibrium between research and sport, between family and beer, and between academic and colloquial French, learned in the form of nice rugby songs, merci les gars!

I am grateful to my teachers and professors at the school, the UPB, and the EPFL. From them I have learned a lot of useful and useless things, without which I certainly would not be writing this thesis.

I am grateful to my sisters, Cecilia and Juliana, both being my opposite or my complement (depending of the point of view). From them I have learned to consider different points of view on diverse topics as society, politics, arts, and family. However, it has been impossible to agree with Juliana when discussing about anarchy or science, and with Cecilia, when discussing about my mother.

I am very grateful to my parents, Jairo and Luz Elvira for raising me. I acknowledge and blame them for giving me the elements for becoming what I am. I must also thank my grandparents, Alberto and Lucy, who have played an important role in my life, and have constituted a model of tenacity, discipline, and tolerance. Even though being far, my family will always have a very special place in my heart.

Finally, I would like to have words to express the infinite love and gratitude that I feel for Carolina and Juan Manuel. They constitute my emotional support, my balance, my reason to live, and my reason to do this...

Table of Contents

Abstract iii

Résumé v

Acknowledgments vii

List of Figures xv

List of Tables xvii

1 Introduction 1
- 1.1 Problem description . . . 3
- 1.2 Proposed solution . . . 4
- 1.3 Outline of the thesis . . . 4

2 Reconfigurable Computing 7
- 2.1 From transistors to FPGAs . . . 8
  - 2.1.1 Before programmable devices . . . 8
  - 2.1.2 SPLD . . . 8
  - 2.1.3 CPLD . . . 11
  - 2.1.4 FPGA . . . 11
- 2.2 Dynamically reconfigurable FPGAs . . . 13
  - 2.2.1 Custom reconfigurable platforms . . . 13
  - 2.2.2 Commercial reconfigurable platforms . . . 14
- 2.3 Dynamic reconfiguration of Xilinx FPGAs . . . 20
  - 2.3.1 Self-reconfigurable systems . . . 22
  - 2.3.2 Virtex-II bitstream . . . 23
- 2.4 Conclusions . . . 28

3 Bio-inspired Systems 31
- 3.1 The POE model . . . 32
- 3.2 Phylogeny . . . 32
  - 3.2.1 Evolutionary algorithms . . . 33
  - 3.2.2 Genetic algorithms . . . 34
  - 3.2.3 Particle swarm optimization . . . 35
  - 3.2.4 Cellular programming . . . 37

3.2.5 Fuzzy CoCo . . . 39
3.3 Ontogeny . . . 40
3.4 Epigenesis . . . 41
3.4.1 Artificial neural networks . . . 41
3.4.2 Spiking neurons . . . 43
3.5 Hybrid POE approaches . . . 47
3.6 Conclusions . . . 49

4 Evolvable FPGAs 51
4.1 Evolvable hardware: an introduction . . . 51
4.1.1 Genome encoding . . . 53
4.1.2 Fitness computation . . . 54
4.2 Evolvable hardware: a taxonomy . . . 54
4.2.1 Extrinsic evolution . . . 54
4.2.2 Intrinsic evolution . . . 55
4.2.3 Complete evolution . . . 56
4.2.4 Open-ended evolution . . . 57
4.3 Evolvable hardware digital platforms . . . 58
4.3.1 Xilinx 6200 family . . . 59
4.3.2 Evolution on commercial FPGAs . . . 60
4.3.3 Custom evolvable FPGAs . . . 64
4.4 Conclusions and future directions . . . 65

5 Bio-Inspired Adaptation Components 67
5.1 Computation engine . . . 68
5.2 Adaptation mechanism . . . 69
5.2.1 Parametric adaptation . . . 69
5.2.2 Structural adaptation . . . 70
5.3 Conclusions . . . 71

6 Parametric Adaptation 73
6.1 A hardware-oriented spiking neuron model . . . 74
6.1.1 Hardware-oriented spiking neuron model . . . 74
6.1.2 Computing capabilities . . . 76
6.1.3 Learning . . . 80
6.1.4 The proposed neuron model on hardware . . . 81
6.1.5 Experimental setup and results . . . 82
6.2 Coevolutionary setup for adapting fuzzy systems . . . 86
6.2.1 The evolvable FPGA platform . . . 86
6.2.2 Genome encoding . . . 90
6.2.3 Platform setup and results . . . 90
6.3 A self-reconfigurable platform for non-uniform cellular automata . . . 92
6.3.1 The evolvable platform . . . 92
6.3.2 Experimental setup and results . . . 94
6.4 Conclusions . . . 96

7 Structural Adaptation **99**
7.1 Topology evolution of ANNs: a coarse-grained approach 100
7.1.1 Description of the platform 101
7.1.2 Hardware substrate 101
7.1.3 The proposed on-line evolving ANN 102
7.1.4 Modular ANN ensembles 104
7.2 Topology generation for random boolean networks: a fine-grained approach . 105
7.2.1 Random boolean networks 105
7.2.2 The RBN cell array 107
7.2.3 Setup of the self-reconfigurable system 110
7.2.4 Example task: firefly synchronization 112
7.3 Conclusions 113

8 A Reconfigurable Framework for Modular Robotics **115**
8.1 Yet another Modular Robot - YaMoR 116
8.1.1 YaMoR - mechanics and electronics 117
8.1.2 Reconfigurable substrate 118
8.1.3 Bluetooth - the wireless interface to YaMoR 118
8.1.4 Bluemove - controlling YaMoR via Bluetooth 119
8.1.5 Exploring locomotion 120
8.2 An FPGA dynamically reconfigurable framework for YaMoR 122
8.2.1 Self-reconfigurable machines 122
8.2.2 Reconfigurable controllers 123
8.3 Conclusions 127

9 Reconfigurable Pervasive Systems **129**
9.1 Reconfigurable Object for Pervasive Systems - ROPES 131
9.1.1 ROPES layout 131
9.1.2 FPGA board 132
9.1.3 Communications and reconfigurability 133
9.2 Self-reconfigurable pervasive platform for cryptographic application 133
9.2.1 Reconfigurable computing in pervasive systems 134
9.2.2 Cryptography 134
9.2.3 System description 136
9.2.4 Experimental setup and results 138
9.3 On-line self-reconfigurable system for adaptive channel equalization 141
9.3.1 Particle swarm optimization with discrete recombination 143
9.3.2 Binary radial basis functions 145
9.3.3 Experimental settings and results 146
9.4 Conclusions 153

10 Conclusions and Future Work **155**
10.1 Summary 155
10.2 Original contributions 156
10.3 Future work 157

10.3.1 Hardware substrate 157
10.3.2 Dynamic topology architectures 159
10.3.3 Self-reconfigurable adaptive systems 159
10.3.4 Evolvable hardware substrates 160

Bibliography **161**

Curriculum Vitae **181**

List of Figures

2.1 PROM memory 9
2.2 PLA architecture 10
2.3 PAL architecture 10
2.4 CPLD architecture 11
2.5 FPGA architecture 12
2.6 XC6200 functional unit 16
2.7 Virtex-II architecture 17
2.8 Virtex-II CLB 17
2.9 Virtex-II slice 18
2.10 Stratix-II architecture 19
2.11 Modular design layout 21
2.12 Bus macro implementation 22
2.13 Virtex-II configuration bitstream composition 24
2.14 LUT and multiplexer in a Virtex-II slice 27

3.1 Genetic algorithm. 34
3.2 Supervised learning. 43
3.3 Hodgkin and Huxley model response 44
3.4 Examples of spike response model (SRM) kernels 46

4.1 Analogy life-digital. 52
4.2 Hardware evolution 52
4.3 Divisions of phylogenetic hardware 55
4.4 Virtual reconfigurable circuit 61

6.1 Neuron model response. 75
6.2 Post-synaptic responses 76
6.3 Topology of the network for pattern recognition. 77
6.4 Training patterns. 77
6.5 Number patterns. 77
6.6 Hebbian learning window. 80
6.7 Neuron architecture. 81
6.8 Neural network layout 83
6.9 Neural network activity 83
6.10 Fuzzy system platform 87
6.11 Triangular function calculation 88

6.12 Fuzzy rule hard macro 89
6.13 Output fuzzy sets 89
6.14 Genome 90
6.15 Results 92
6.16 System schematic for self-reconfiguring cellular automata 93
6.17 2-automata hard macro 94

7.1 Network layout 103
7.2 ANN evolution 103
7.3 RBN implementation 108
7.4 Example of an RBN configuration 109
7.5 Self-reconfigurable platform setup 111
7.6 Hard-macro for the RBN cell 111

8.1 YaMoR unit 117
8.2 FPGA board. 119
8.3 Timelines manager. 120
8.4 Real-time modules. 121
8.5 Rolling wheel. 121
8.6 Different YaMoR configurations. 122
8.7 Self-reconfigurable machine. 124
8.8 Reconfigurable controller. 125
8.9 Bus macro for Spartan-3. 126

9.1 ROPES. 131
9.2 Layout of a reconfigurable system in ROPES. 132
9.3 RC4 hardware implementation. 135
9.4 DES hardware implementation. 135
9.5 FSL connections. 137
9.6 Hybrid configuration platform. 138
9.7 Binary radial basis function 146
9.8 Learning performance for the Sphere function 150
9.9 Learning performance for the Rosenbrock function 150
9.10 Learning performance for the Rastrigin function 150
9.11 Learning performance for the Griewank function 151
9.12 Communication system model 151
9.13 The proposed equalizer 152
9.14 Learning performance of the proposed equalizer for a SNR of 15 dB 153

List of Tables

2.1 LUT's frame description . . . 26
2.2 Multiplexers' frame description . . . 28

6.1 Criteria used to describe the quality of classifications. . . . 79
6.2 Neuron parameters . . . 84
6.3 Network discrimination . . . 84
6.4 Synthesized neural network . . . 85
6.5 Comparison between software and hardware performances . . . 91
6.6 Comparison between evolved software and hardware performances . . . 91

9.1 Logic resources . . . 139
9.2 Results under the Xilkernel . . . 139
9.3 Results under uClinux . . . 140
9.4 Reconfiguration time . . . 140
9.5 Minimum data size . . . 141
9.6 Initialization ranges and v_{max} for each function . . . 147
9.7 Mean fitness values for the Sphere function . . . 148
9.8 Mean fitness values for the Rosenbrock function . . . 148
9.9 Mean fitness values for the Rastrigin function . . . 149
9.10 Mean fitness values for the Griewank function . . . 149

Chapter 1

Introduction

> I do not know what I may appear to the world; but to myself I seem to have been only like a boy playing on the seashore, and diverting myself now and then finding a smoother pebble or a prettier shell than ordinary, whilst the great ocean of truth lay all undiscovered before me.
>
> Isaac Newton

Living beings have managed to survive on earth during the last four billion years. The main reason for such a success is certainly their striking capacity to adapt to changing and adverse environments. They possess astonishing faculties to learn from unknown situations, to adapt their behavior and their shape to environmental changes, to self-reproduce while keeping a species' most useful features, and to self-repair without any external intervention. A key element for exhibiting these features is their completely distributed and self-organized structure at four basic levels:

- At *cellular level*, the basic component of living beings -the cell- is composed of a set of molecules that allows it to move, to grow, to reproduce (Mitosis and Meiosis), to decode DNA chains, to synthesize proteins, etc. It is by a constant interaction among the different components in the cell (DNA, ribosomes, mitochondria, centrosomes, etc.) that the cell's development and reproduction is guaranteed.
- At *individual level*, the interactions among cells allow the construction of a complete organism. From an initial mother cell, an organism is constructed by cellular reproduction. The interaction among cells, their differentiation mechanism, and their self-regulation constitute an impressive self-organizing system exhibiting an incredible level of complexity. A good example is the human brain, hundreds of billions of neurons simultaneously firing electrical pulses in an asynchronous way, which allows humans to

infer a world perception, to have feelings, to feel danger, to coordinate movements, and, thought not its forte, to perform mathematical operations.

- At *species level*, individuals manage to establish social rules for interacting among members of the same species. Troops of mammals, such as lions, gorillas, and wild horses live in communities under the guidance and protection of a leader. Bees and ants, in spite of their relative low individual complexity, exhibit very impressive social structures where each individual has a task in the colony, and together display very interesting levels of complexity.

- Finally, at *life level*, plants receive energy from the sun, herbivores eat plants, carnivores hunt herbivores, and decomposers feed from dead plants and animals, reducing them to minerals and gases required for growing plants again. In this way, what we know as a food chain is formed. It is surprising how such a system can find a natural equilibrium for allowing every species to survive; some species may disappear, others might migrate, but life continues its path. During billions of years living beings have evolved in a completely self-organized way, their behavior and their morphology in order to adapt to a number of environmental changes.

Bio-inspired systems aim to extract some interesting features from these living beings, such as adaptability and fault tolerance, for including them in human-designed devices. In engineering and science one can find several examples of bio-inspiration: airplane's wings have been inspired from birds, sonar and radar technologies take direct inspiration from bats' navigation system, and vaccines use the knowledge acquired from observing natural immune systems. These are just some of the numerous lessons that scientists and engineers have learned from mother nature.

These bio-inspired designs have been classified according to their application field. Among these fields one can find: bio-inspired materials, bio-inspired locomotion, bio-inspired sensing, self-adapting systems, bio-inspired autonomous robots, etc. Electronic circuit design has been also inspired by biology at several levels. Biological systems exhibit several desirable features for electronic circuits: robustness, adaptability, development, scalability, and autonomy, among others. Neural hardware and evolvable hardware are two examples where human-designed circuits take inspiration from nature.

Several issues arise when defining the hardware substrate supporting a bio-inspired architecture. Designers must typically design their own integrated circuit, implying a very expensive and time-consuming design process, or they must implement a virtual reconfigurable substrate by using commercial reconfigurable devices, which is very inefficient in terms of resource utilization.

The central aim of this thesis is to explore the feasibility of efficiently implementing bio-inspired systems in current commercial FPGAs by exploiting their partial and dynamic reconfigurability features. At the same time, this thesis proposes several architectures and methodologies for the design of bio-inspired reconfigurable hardware systems.

1.1 Problem description

User consumer electronics are day by day exhibiting higher performance, enhanced flexibility, and lower cost. current, a mobile phone provides a wide variety of applications such as camera, video recording, video encoding and decoding, streaming TV, mp3 decoding, Bluetooth communication, games,- and it can even be used for making phone calls! In the near future, it can be easily envisioned that the whole signal processing will move from a centralized console (a personal computer) to the cellular phone; moreover, one can imagine a large number of new applications to be included in there. Additionally, protocols and standards are being evolved and upgraded more often, obliging consumers to upgrade their telephone every year. This changing scenario is not exclusive for mobile phones; another example is the field of wireless sensor networks. This field is also experiencing the needs of distributing signal processing to wireless nodes, instead of processing in a powerful centralized machine, while upgrading to improved versions of security and routing protocols.

Achieving high performance for demanding applications is not possible with microcontroller based solutions, and specialized hardware coprocessors are required for satisfying the required timing constraints. A specialized coprocessor allows one to perform a more efficient computation for a very specific task by exploiting a specialized hardware architecture. However, these types of hardware solutions lack flexibility; a hardwired coprocessor can only solve the specific problem for which it has been designed, and unlike software-based solutions, a coprocessor's functionality cannot be upgraded after the device has been manufactured.

Reconfigurable computing offers a solution to this problem: a reconfigurable portion of the system -a coprocessor for instance- can be upgraded to allow it to perform any desired task. In this way, one can have a system with a reconfigurable coprocessor, whose architecture can be upgraded whenever it is required. Under this schema we can benefit from the high performance offered by hardware solutions, while keeping the platform flexible and upgradeable. Several issues arise when adopting this reconfigurable scenario: What reconfigurable substrate will support the architecture? How to design and generate the reconfigurable part? Who will reconfigure it? When will it be reconfigured?

Maybe some answers can be found in self-reconfigurable systems. A system is said to be self-reconfigurable when it modifies its own architecture in an autonomous way. Self-reconfigurable systems are popular currently: one can find a number of reconfigurable devices reconfigured by an internal processor [7,14,117,155,203,213]. This reconfigurablity allows an increased flexibility and a high degree of autonomy, while still guaranteeing high performance. However, under this schema the reconfigurable system remains limited to a given number of pre-generated coprocessor configurations, providing solutions only for problems known at design time. The adaptability of this platform is, consequently, limited to a given set of previously known solutions.

Evolvable hardware provides a solution to this adaptability problem. From a given specification and constraints of the new problem at hand, an evolutionary algorithm modifies the hardware in order to find a coprocessor implementing the desired solution. One of the most important issues to tackle when evolving hardware is to define the hardware substrate to be evolved. Two main types of reconfigurable substrates are typically used: a custom evolvable platform and a virtual reconfigurable architecture built on top of a commercial FPGA fabric. However, both solutions present non-negligible drawbacks. Creating a custom evolvable

hardware chip carries the general problems of designing any integrated circuit: it implies very high fabrication costs and very long times for designing, manufacturing and validating. Using virtual reconfigurable architectures constitutes a cheaper solution that can be developed in a reasonable amount of time. However, the fact of building a reconfigurable architecture on top of another reconfigurable architecture implies an inefficient resource utilization, given the reconfiguration level redundancy.

1.2 Proposed solution

This thesis proposes several methodologies and techniques for conceiving bio-inspired self-reconfigurable systems able to benefit from current commercial FPGAs' dynamic partial reconfigurability. Current design tools and techniques support the implementation of partial self-reconfigurable systems, however, conventional tools and techniques lack the required flexibility for supporting systems featuring on-line architectural adaptation.

The proposed methodology is composed of two main components: a computation engine and an adaptation mechanism. The computation engine consists in the bio-inspired architecture computing the solution. In this thesis the computation engine is presented in the form of neural networks, spiking neural networks, fuzzy systems, cellular automata, and random boolean networks. The adaptation mechanism, on the other hand, allows the computation engine to adapt to performing a given computation; it is basically presented in the form of evolutionary and learning algorithms.

The proposed system, along with the proposed bio-inspired techniques, allows a piece of hardware to self-adapt by reconfiguring the hardware supporting it, in an autonomous way, without external human intervention. The main advantage, with respect to existing self-reconfigurable platforms, consists in the fact that every possible architecture to be implemented is not required to be explicitly specified at design time, but it is the platform itself which determines it.

1.3 Outline of the thesis

This thesis is structured in ten chapters. The first four are introductory chapters for the topics of reconfigurable computing, bio-inspired systems, and evolvable hardware. The next three chapters present the methodologies and the architectures developed during this thesis, and also present some results for each of them. Then, the next two chapters present two system prototypes developed during this thesis. Finally, the last chapter concludes. A more detailed content description is presented in the next paragraphs.

Chapter 2 presents an introduction to reconfigurable devices, more precisely to FPGAs. By introducing, in historical order, different types of programable circuits, this chapter discusses the features, advantages, and drawbacks of each of them. In this chapter, the dynamic partial reconfigurability of FPGAs is specially emphasized, more precisely for Xilinx devices.

Chapter 3 introduces the field of bio-inspired systems in the framework of the POE model. It presents the background for the bio-inspired techniques that are further used in this thesis, by classifying them in each of the axes of the POE model. This chapter also presents the possible hybrids among the POE axes.

Chapter 4 concentrates on the phylogenetic axis of hardware systems: evolvable hardware. It focuses, more precisely, on evolvable hardware on FPGAs, presenting implementation issues, classifications, hardware platforms, and implementation techniques for these types of systems.

Chapter 5 introduces an organizational approach for implementing self-adapting systems, consisting of two main parts: a computation engine (the hardware substrate solving the problem at hand) and an adaptation mechanism (the algorithm performing the adaptation on the computation engine), composed subsequently of two basic forms of adaptation: parametric and structural.

Chapter 6 develops the concept of parametric adaptation, presenting several techniques along with their respective computation engines. This chapter proposes architectures for spiking neurons with hebbian learning, coevolutionary fuzzy systems, and evolving non-uniform cellular automata.

Chapter 7 develops the concept of structural adaptation, presenting two techniques for adapting topologies by exploiting partial reconfigurability on FPGAs. The first technique constitutes a coarse-grained modular approach, while the second one allows topological modifications at a fine-grained interconnection level.

Chapter 8 presents YaMoR (*Yet another Modular Robot*), a modular robot developed during this thesis, in close collaboration with the Biologically Inspired Robotics Group (BIRG). Special focus is put on the reconfigurability features of the FPGA-based systems contained in each module.

Chapter 9 presents the prototyping platform ROPES (*Reconfigurable Object for Pervasive Systems*) developed during this thesis, in close collaboration with the Reconfigurable & Embedded Digital Systems group (REDS) at the *Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud* (HEIG-VD). It also presents two applications for this platform: a self-reconfigurable cryptographic system, and a self-adaptive channel equalizer.

Finally, chapter 10 concludes, summarizing the most important contributions of this thesis and proposing some guidelines for further research in this topic.

Chapter 2

Reconfigurable Computing

> The machine does not isolate man from the great problems of nature but plunges him more deeply into them.
>
> Antoine de Saint-Exupéry

Hardware engineers have drawn inspiration from biology for designing and building bio-inspired architectures. A very important issue is the hardware platform supporting them. It is highly desirable for the platform to provide flexibility, scalability, and autonomy. Bringing flexibility to hardware devices is not trivial. An electronic circuit can be seen as a set of basic electronic components (transistors, capacitors, resistances, etc) interconnected in a certain way. Modifying such a circuit implies replacing some components or modifying some of its connections. Performing these modifications would be unfeasible without the concept of *reconfigurability*.

An electronic device is said to be configurable (or programmable) when its functionality is not pre-defined at fabrication-time, but can be further specified by a configuration bitstream (or a program). Reconfigurable devices permit configuration several times, supporting system upgrades and refinements in a relatively small time scale. Given this architectural flexibility and upgradeability, they constitute the best candidate for supporting bio-inspired architectures: they offer a set of features that permit the implementation of flexible architectures, while still guaranteeing high performance execution.

This chapter presents a short history of reconfigurable computing devices, and highlights the main feature providing flexibility: dynamic partial reconfiguration. Additionally, this chapter describes how to reconfigure current commercial devices in a dynamic and partial way, by presenting some techniques proposed by the reconfigurable device vendor and others original to this thesis.

2.1 From transistors to FPGAs

2.1.1 Before programmable devices

On 22 December 1947, three physicists -Brattain, Shockley and Bardeen- succeeded in creating the first practical transistor at Bell Labs. It consisted in a PNP point-contact transistor built of germanium, operated in its first demonstration as a speech amplifier. One can consider this day as the birthday of the transistor and of electronics, which have experienced tremendous development during the last 60 years.

After proving the feasibility of including several transistors in a single piece of semiconductor in 1958 by Jack Kilby, it was just a matter of time for experiencing the appearance of the first commercial integrated circuit (IC). During the mid-60s, Texas Instruments began the production of the 74xx and the 54xx series, while RCA introduced the 4000 series. These series consisted in 14-pin and 16-pin ICs providing users with logical gates, buffers, multiplexers, flip-flops, and counters, among other basic circuits. During those days, a circuit designer's task consisted in designing circuit boards where all these logical components where connected together. The result was a huge circuit board, very complicated to design, to debug, and to validate.

The first technological efforts aimed to reducing the board size. During the late 60s one can find the first ASICs (Application Specific Integrated Circuits) [193], including all these logical elements in a single chip. A critical problem with them was, and it is still today, the lack of flexibility. The ASIC design process is expensive, takes a lot of time, and does not accept errors during the circuit design. Once an ASIC has been fabricated, fixing a bug implies the fabrication of a new circuit. Even though the design process can be very expensive, it results in cheaper circuits for large amounts of ICs; the price per transistor is considerably reduced when increasing the amount of ICs, giving good results for large-scale production but too expensive for prototyping.

During the early 70s, there were two births of special interest: RAM (Random Access Memory) [157] and the microprocessor [26]. While Intel introduced the first DRAM (Dynamic RAM) and the 4004 microprocessor (also referred to at the time as "computer-on-a-chip"), Fairchild introduced the first SRAM (Static RAM) [157]. The importance of these two new circuits is widely accepted at present given their omnipresence in daily life objects, and the enormous evolution they have experienced in the last 35 years.

2.1.2 SPLD

The utilization of PROMs (Programmable Read-Only Memory) for implementing combinatorial logic functions was a first step to the further evolution of the so-called PLDs (Programmable Logic Devices) [21]. The designation of "PLD" was used until the introduction of CPLDs (Complex PLD), which consisted in an interconnected array of PLDs. Consequently, their humble predecessor -the PLD- downgraded its name to SPLD (Simple PLD).

PLDs' architectures allow the implementation of any combinatorial function as the canonical sum of products of the input variables. PLDs, based on this principle, are composed of a connection grid of AND gates (the product), followed by a connection grid of OR gates (the sum). There exist several approaches when providing programmability to such an array. De-

pending on the allowed programmable grid we find three families of devices: PROMs, PLAs, and PALs.

2.1.2.1 PROM

In parallel with the birth of RAM and the microcontroller during the early 70s, we find the first programmable ICs in the form of PROMs (Programmable Read-Only Memory) [157]. At the beginning, PROMs were intended to be used as computer memories; however, some design engineers began using them for implementing logical functions, constituting a flexible alternative to previous, rigid technologies.

PROMs are composed of a fixed AND grid at the input (constituting the memory address decoding), followed by a programmable OR grid. Figure 2.1 depicts a PROM's architecture. An input bus (the memory address) determines the data being presented at the output. The address space is predefined at IC fabrication, and it supports all the possible combinations at the input.

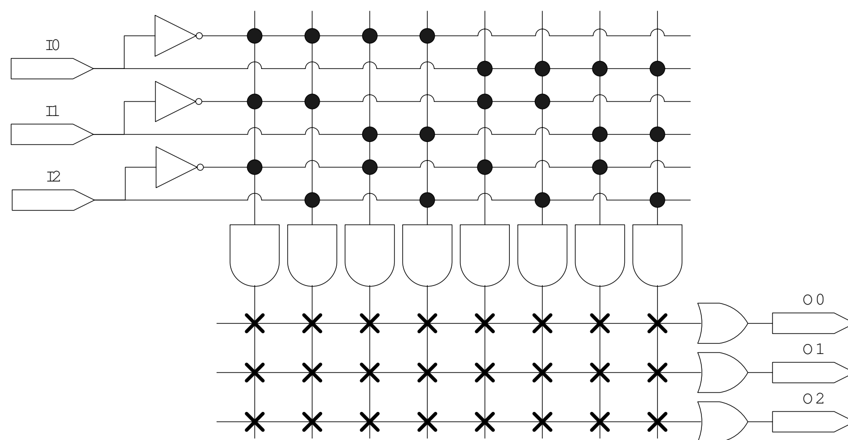


Figure 2.1 *Programmable Read-Only Memory - PROM.*

Different IC fabrication technologies can support the programming of the OR grid. The first PROMs used a fusible-link technology. When unprogrammed, every link in the OR grid is active; the programming consists in burning some fuses for deactivating the links, in order to leave active just the corresponding inputs to the OR gate. Other programming technologies include antifuse links, the use of floating gate polysilicon -as used in EPROM (Erasable PROM) or in the E^2PROM (Electrically Erasable PROM) -, and more recently FLASH technologies [148]. For all of them, the architecture and the operation principle are roughly the same.

2.1.2.2 PLA

In 1975, the first commercial PLA (Programmable Logic Array) [21] was produced, in order to overcome the limitations of PROMs. The main feature of PLAs was the programmability of both: the AND and the OR grids (see figure 2.2), which constituted the most configurable device of the existing SPLDs. Another significant difference from PROMs is the independence

of the number of AND gates from the number of inputs: while a PROM requires an AND for independently decoding every possible combination of inputs, in a PLA one can have different combinations of inputs selecting the same output. Additionally, unlike PROMs, a given value at the input can enable several AND gates, making PLAs more flexible than PROMs.

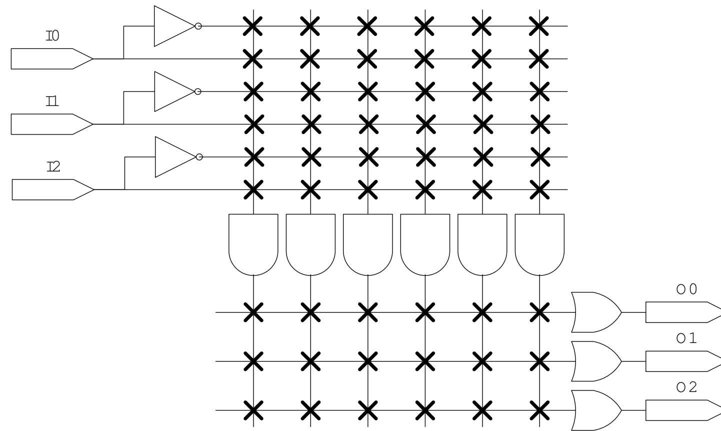


Figure 2.2 *Programmable Logic Array - PLA.*

In spite of their high programmability, PLAs did not succeeded commercially. They presented an important drawback: the delay caused by the two programmable links made them considerably slower than PROMs, which presented a single one.

2.1.2.3 PAL

In order to overcome the delay problems posed by the PLAs, during the late 70s the PALs (Programmable Array Logic) [21] was introduced. PALs circuits, as PROMs, have a single programmable link; however, unlike PROMs, the programmable link is in the AND grid instead of the OR grid (see figure 2.3). The result is a faster circuit, but with a reduced programmability.

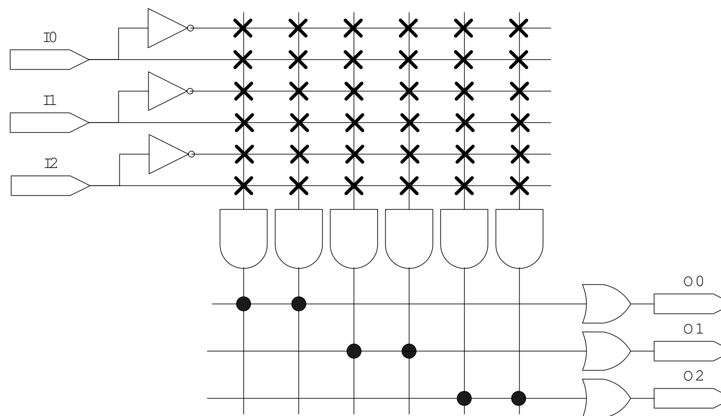


Figure 2.3 *Programmable Array Logic- PAL.*

2.1.3 CPLD

During the late 70s and early 80s, PLDs began their evolution to a more complex concept. Monolithic Memories introduced the MegaPAL: an interconnected array of four standard PALs. The MegaPAL did not have any commercial success, mainly because of its high power consumption, and it was retired from the market.

A few years later Altera introduced a CPLD (Complex PLD) [16] that overcame the problem of its predecessor by using CMOS and EPROM technologies. Additionally Altera reduced the amount of required inter-PLD connections by including programable multiplexers in the inputs of each PLD.

Figure 2.4 illustrates a typical CPLD architecture. An array of PLDs is connected by an interconnection matrix. PLDs can also be accessed by input/outputs blocks providing an enhanced configurability, an increased complexity, and a more flexible architecture than their simple predecessors.

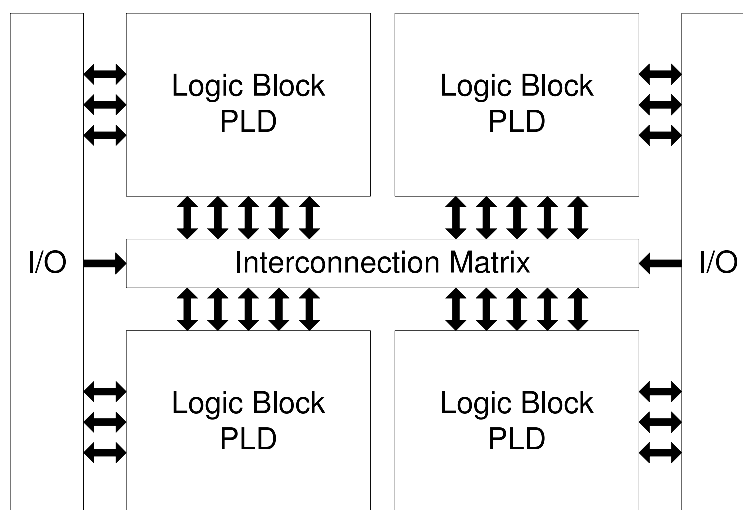


Figure 2.4 *Complex Programmable Logic Device - CPLD.*

2.1.4 FPGA

Science and technology are two elements that always feature human dissatisfaction. Circuits are not the exception to this and applications are always demanding higher speed, lower power consumption, and more logical resources. Increasing CPLDs' performance in these directions has been tackled by technological improvements: increasing scale of integration, exploring new fabrication technologies, etc. However, from the architectural point of view, CPLDs have a scalability problem, when increasing the size of the SPLD blocks, the interconnection array increases in a quadratic way, limiting the maximum circuit complexity that can be tackled by using CPLDs.

During the early 80s, a gap began to become evident in the IC design field. At one side it was possible to benefit from the programmability of CPLDs for fast-prototyping designs. However, when the circuit complexity was relatively high it was not possible to use CPLDs

and the prototype was required to be implemented in ASICs, with the large development time and high costs that it implied. Designers were facing a situation where, ironically, they could not afford to be wrong when designing larger circuits.

In 1984 Xilinx launched to the market the first FPGA (Field Programmable Gate Array) [124, 212], the XC2064, offering an alternative to the previous two approaches. The main advantage of FPGAs with respect to CPLDs is the FPGAs' architecture scalability. Unlike CPLDs, FPGAs' size can be increased without sacrificing performance. However, unlike CPLDs, FPGAs' architecture does not allow accurate prediction of the design timing.

FPGAs are programmable logic devices that permit the implementation of digital systems. FPGAs are typically composed of an array of uniform logic cells, interconnected through programmable interconnection matrices, that can be configured to perform a given function by means of a configuration bitstream (see figure 2.5). Additionally, current FPGAs provide other embedded functional blocks, depending on the targeted application field. Among the typical embedded blocks one can find microcontrollers, RAM memories, and embedded arithmetic operators.

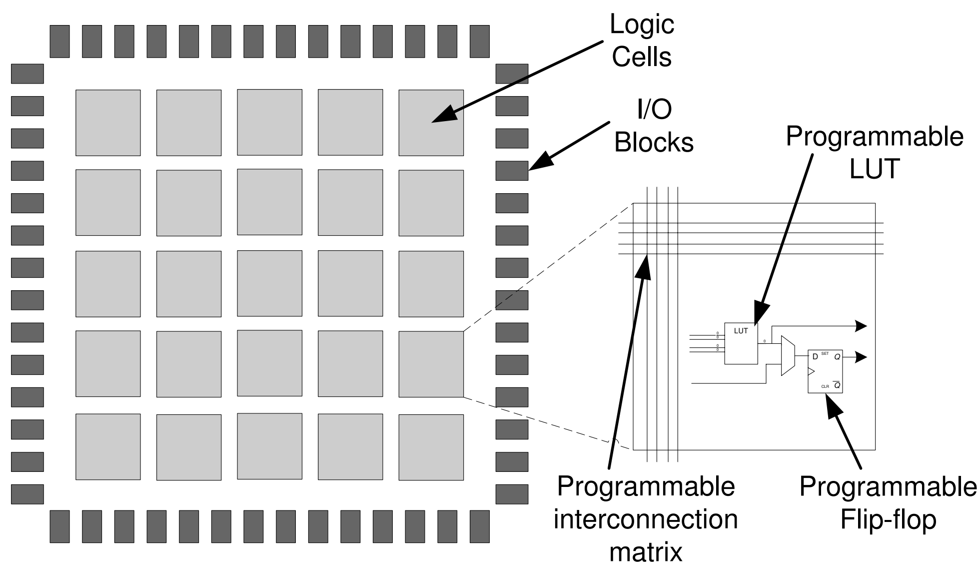


Figure 2.5 *Field Programmable Gate Array - FPGA.*

Figure 2.5 depicts also a typical logic cell architecture. Each logic cell is composed of combinatorial and sequential programmable components, whose inputs and outputs are connected to a programmable interconnection matrix. The most used combinatorial components are look-up-tables (LUT), which can be programmed to implement any desired n-input function. Different sizes of LUTs can be found according the FPGA manufacturer and family. The very first FPGAs featured 3-inputs LUTs, providing a fine-grained architecture for maximizing flexibility. Some FPGA vendors have increased the number of inputs providing coarser grain architectures; in this way these FPGAs allow higher performance for some applications families, while sacrificing flexibility. Currently, most commercial FPGAs feature 4-input LUTs. In the same way, FPGAs' logic cells contain configurable sequential elements: one can find flip-flops with configurable reset signals, edge sensitivity, and enabling options.

2.2 Dynamically reconfigurable FPGAs

FPGAs' programming is performed by means of a configuration bitstream, which is stored in a configuration memory. This bitstream contains the configuration information for every FPGA's internal elements: interconnection matrices, LUT functions, flip-flop's initial state, etc. At present, FPGAs' bitstreams exhibit a considerable complexity, and are generated by design automation tools. A typical design flow for generating a bitstream begins with the description of a circuit by using an HDL language. The HDL code is synthesized for generating a netlist, which constitutes a system description in terms of the FPGA's basic components. The netlist description, along with some design constraints, allows one to perform the placement and routing of the design on the targeted FPGA. Details during the design flow may vary between FPGA vendors, but the basics of the flow remain the same.

When the circuit has been completely placed and routed, a configuration bitstream is generated, which describes the functionality of every programmable element in the FPGA. Before programming the FPGA, it is non-operational (waiting for a bitstream), and the configuration memory is empty. This is said to be a *static configuration* [170], given that for reconfiguring the device with a new bitstream, the configuration memory must be erased and the configuration process must be restarted.

Some FPGAs allow the performing of *partial reconfiguration* [75, 245], where a reduced bitstream reconfigures only a given subset of internal components. Several issues arise when partially reconfiguring an FPGA, the routing being one of the main problems. When a portion of the FPGA is modified one must ensure that the modified section will still be compatible with the unmodified part. For instance, a processor with a reconfigurable coprocessor must ensure that when modifying the coprocessor logic and its respective routing resources, the new coprocessor will still be correctly attached to the original coprocessor interface.

Dynamic partial reconfiguration (DPR) [30, 75, 245] is done while the device is active: certain areas of the device can be reconfigured while other areas remain operational and unaffected by the reprogramming. Unlike a *static partial reconfiguration*, in DPR the system execution is not interrupted during the reconfiguration. However, the reconfiguring section will be unuseable during the reconfiguration time.

Self-reconfigurable systems [7, 14] result as a direct offspring of DPR systems. *Self-reconfigurable systems* can modify their own hardware substrate, in order to provide high performance, high flexibility, and high autonomy. They exhibit high performance due to their hardware nature. Their high flexibility is achieved thanks to the possibility of modifying their hardware at run-time by using DPR. The autonomy is assured by the capability of the platform to modify itself without any external intervention. Self-reconfigurable systems have been mainly used for reconfigurable coprocessors, where a repository of configurations is previously available, and an embedded processor reconfigures the platform with the coprocessor required at a given moment.

2.2.1 Custom reconfigurable platforms

Reconfigurable platforms provide a trade-off between the performance of hardware and the flexibility of software. Having more knowledge about the targeted application, allows the use of a more customized reconfigurable platform, better suited for the task at hand. However, for

a different task than the targeted one the platform may perform poorly.

Several custom platforms have been proposed for supporting reconfigurable systems. Typically, these platforms target an application or a family of them. One can roughly determine the level of flexibility of a reconfigurable system by taking into account two aspects, which are closely related: how application-oriented the architecture is, and its granularity level. In this way, among custom fine-grain architectures, one can find devices for application domains such as: complex bit-oriented computations (Splash [43] and DEC PeRLe-1 [10]), bit level image processing (Garp [59]), or general bit-level computations (Chimaera [58], DPGA [201], and DISC [230]). The bit-level orientation of these applications provides an enhanced flexibility when implementing any other task. On the other hand, we find coarse-grained architectures targeting such domains as: data-parallel processing (Remarc [133] and MorphoSys [182]), DSP applications (PADDI [23]), and Systolic arrays (RaPiD [33]). In these cases, the coarse granularity does not support bit-level computations efficiently, constituting a less flexible solution than their fine fine-grained conterparts.

One can also find bio-inspired reconfigurable devices among the current custom reconfigurable platforms. One of the more recent bio-inspired chips is the POEtic tissue [203], a platform for bio-inspired hardware composed of the three layers of the POE model: phenotype, mapping, and genotype, each one of them supporting each of the three axes of life: phylogenesis (evolution), ontogenesis (development) and epigenesis (learning) (more details on the POE model in chapter 3). Previous work on evolvable architectures has been done by Moreno et al. with FIPSOC [139], a chip integrating digital and analog programmable circuits, with a dynamic multi-context reconfiguration for the digital section, focusing on evolution of parallel cellular machines. Higuchi's group has developed an evolvable LSI chip [80], which includes a genetic algorithm unit, and the ability to process two chromosomes in parallel. Layzell developed the Evolvable Motherboard [101], a diagonal matrix of analogue switches, connected to a set of daughter-boards, which contain the basic components for building a circuit.

In general, custom reconfigurable platforms are application oriented, keeping themselves restrained to a very reduced market participation (or no commercial use at all, as all the devices presented in this section). This small scale production makes them, consequently, very expensive, being exclusively available to some privileged research groups.

2.2.2 Commercial reconfigurable platforms

Among the commercial reconfigurable devices available in the market, this section will concentrate on those offering dynamic reconfigurability features. When we talk about dynamic reconfigurable devices, we are referring in more than 90% of cases to FPGAs. Dynamic reconfigurability requires a fast reconfiguration timing, only offered by SRAM technologies. Among the reconfigurable devices available in the market, this technology is exclusively offered by FPGA architectures. However, the main interest in FPGAs as dynamic reconfigurable platforms is not because of their reconfiguration speed, but because of its high architectural flexibility compared with CPLDs, making FPGAs' architectures the most appropriate platform for dynamically reconfigurable systems.

Commercial FPGAs typically offer more flexibility than their custom counterparts. Again, given the tradeoff between performance and flexibility, in most of the cases the more customized architectures can offer a better performance. However, during recent years the

performance gap between FPGAs and custom devices has been reduced in an impressive way. This relative performance increase of FPGAs can be attributed to the market importance that they have acquired with respect to custom circuits. The number of FPGAs sold these days allows them to benefit from the latest fabrication technologies.

In the market, we find around a dozen FPGA manufacturers, among which the leader, at the time of this writing, is Xilinx with more than 50% of market share, followed by Altera with more than 30%, and Lattice and Actel with approximately 7% each. The remaining market share is divided among different smaller manufacturers: Atmel, QuickLogic, and MathStar, among others.

In general, commercial FPGAs have several architectural aspects in common, and the logic cell array basics are almost the same. However, smaller FPGA vendors' strategies consists in proposing enhancements for some specific applications, trying to conquer a specific portion of the market. For instance, QuickLogic focuses on low power FPGAs [158], Actel offers FPGAs configurable with Flash and Anti-fuse technologies [2], Lattice emphasizes their FPGAs' high speed capabilities [100], and others, like MathStar, propose new architectural paradigms: their field reconfigurable object array (FPOA) supports input clock frequencies of up to 1 GHz, doubling the clock frequency supported by the fastest FPGA from Xilinx [122].

2.2.2.1 Xilinx

Xilinx is the inventor and the world leader in production of FPGAs. Currently, they produce two FPGAs series: the high performance Virtex series, and the low cost Spartan series. It is also important, for the scope of this thesis, to cite the obsolete FPGA family XC6200 [236], given its reconfigurability features.

The XC6200 family constituted a very important platform for the evolvable hardware community. This family offered two very interesting features for evolving hardware: (1) it offered the possibility of being dynamically and partially reconfigured and (2) the logic cells and the routing were implemented as a multiplexer based architecture. Figure 2.6 illustrates a XC6200 functional unit. The multiplexer based architecture guaranteed that any arbitrary configuration bitstream downloaded to the FPGA would be free of internal contentions. Additionally, it reduced the size of the configuration bitstream, compared to matrix switch based interconnections, also reducing the probabilities of having invalid circuits. The most famous work doing evolvable hardware on these devices is reported by Adrian Thompson [206–209], who directly evolved their configuration bitstream.

The last family produced by Xilinx is the Virtex-5, whose architecture offers a choice of four different platforms, each one optimized for different specific features: high-performance logic, serial connectivity, signal processing, and embedded processing. The previous family, still not fully supported by Xilinx development tools, is the Virtex-4, which, in a similar way to the Virtex-5, offers three application-domain-optimized platforms: the LX platform provides a high number of logic cells, the SX platform provides enhanced signal processing resources, and the FX platform provides resources for embedding processing. When developing this thesis (and still at the time of this writing) dynamic partial reconfiguration is still not supported for Virtex-4, but only for Virtex and Virtex-II families. Because of that, the work reported in this thesis mainly uses the Virtex-II family (The low cost families, Spartan-II and Spartan-3, are also used for some implementations), on which this subsection will be focused.

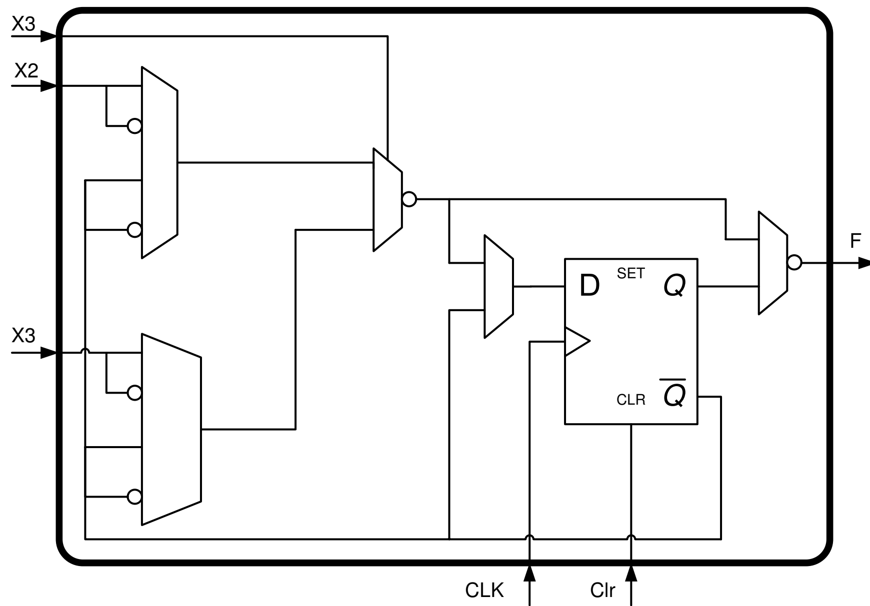


Figure 2.6 *XC6200 functional unit.*

The low cost Spartan series inherit the Virtex series architecture. For instance the latest family, the Spartan-3, basically provides the same logic cells, the same routing, and the same embedded blocks (memories, multipliers, ...) as the Virtex-II. However, the amount of resources, the maximum clock frequency, some architectural features, and the support are not as good as for their high-performance counterpart.

An overview of the Virtex-II architecture can be visualized in figure 2.7. It is basically composed of an array of configurable logic blocks (CLB) surrounded by configurable input-output blocks (IOB), embedded 18-bit hardwired multipliers, synchronous memory blocks called block SelectRAM, and digital clock managers (DCM).

The CLBs for the Virtex-II and Virtex-4 families do not differ much. A Virtex-II CLB is depicted in figure 2.8; it is composed of four slices arranged in two columns of two slices, and two 3-state buffers (which are not included in Spartan-3 architectures). The slices and the 3-state buffers are connected to a switch matrix, allowing them to be connected to any other CLB in the FPGA. Additionally, slices can be directly connected to neighbor CLBs through a fast connection bus, in order to avoid the delay induced by the switch matrix.

A slice is mainly composed of two LUTs and two storage elements. Figure 2.9 illustrates the top half of a Virtex-II slice (the bottom half is very similar). There are two main components: a 4-input look-up-table (LUT) and a D-type storage element. The LUT can be configured to behave as a single- or dual-port RAM, a ROM, a shift register, or a simple LUT for implementing any 4-input combinatorial logic function. The D-type storage element can be configured to behave as a flip-flop or as a latch, and provides several configurable features, such as set, reset, and enable. In addition to these two basic components, the slice architecture provides other interesting features. *Fast carry logic* provides a direct connection with the above and below LUTs without using routing matrices, in order to reduce arithmetic computation delays; it also provides an embedded fast *sum of products* (SOP) computation; and, finally, it provides a high flexibility for interconnecting the different components inside the slice.

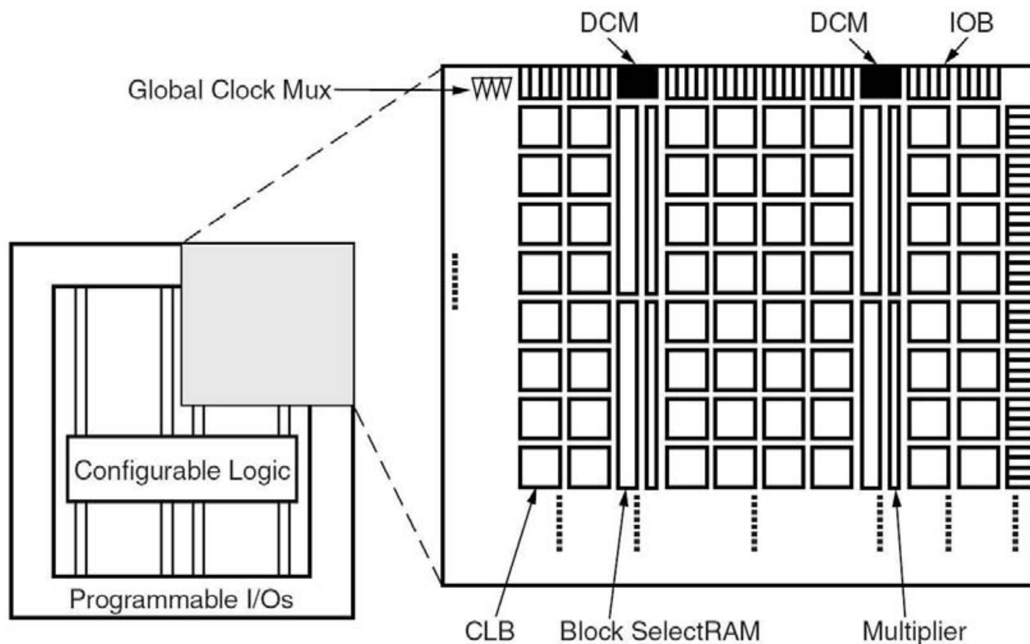


Figure 2.7 *Virtex-II architecture (Taken from [240]).*

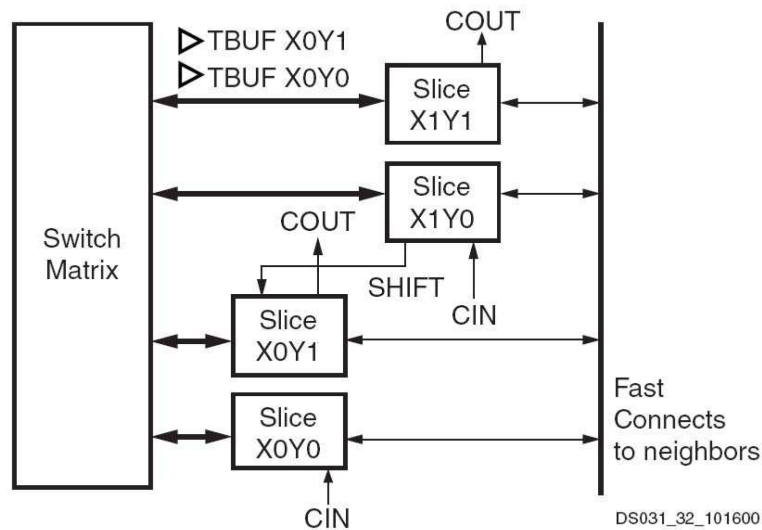


Figure 2.8 *Virtex-II CLB (Taken from [240]).*

The Virtex-5 CLBs differ considerably from their predecessors. A Virtex-5 CLB is composed of two slices arranged in different columns; in this way they are not interconnected to each other by carry chains or any other interconnection lines, but the carry chains connect them directly to the above and below CLBs. Each slice contains four LUTs and four storage elements. To this point, it can just be considered as a different way of presenting the same Virtex-4 and Virtex-II CLB. However, a major change is the size of the LUTs: the Virtex-5 slices use 6-input LUTs, greatly increasing its computational power and its complexity. This modification from the conventional structure provides a coarser architecture granularity, being still consequent to previous empirical studies arguing that LUT sizes from 4 to 6 inputs make

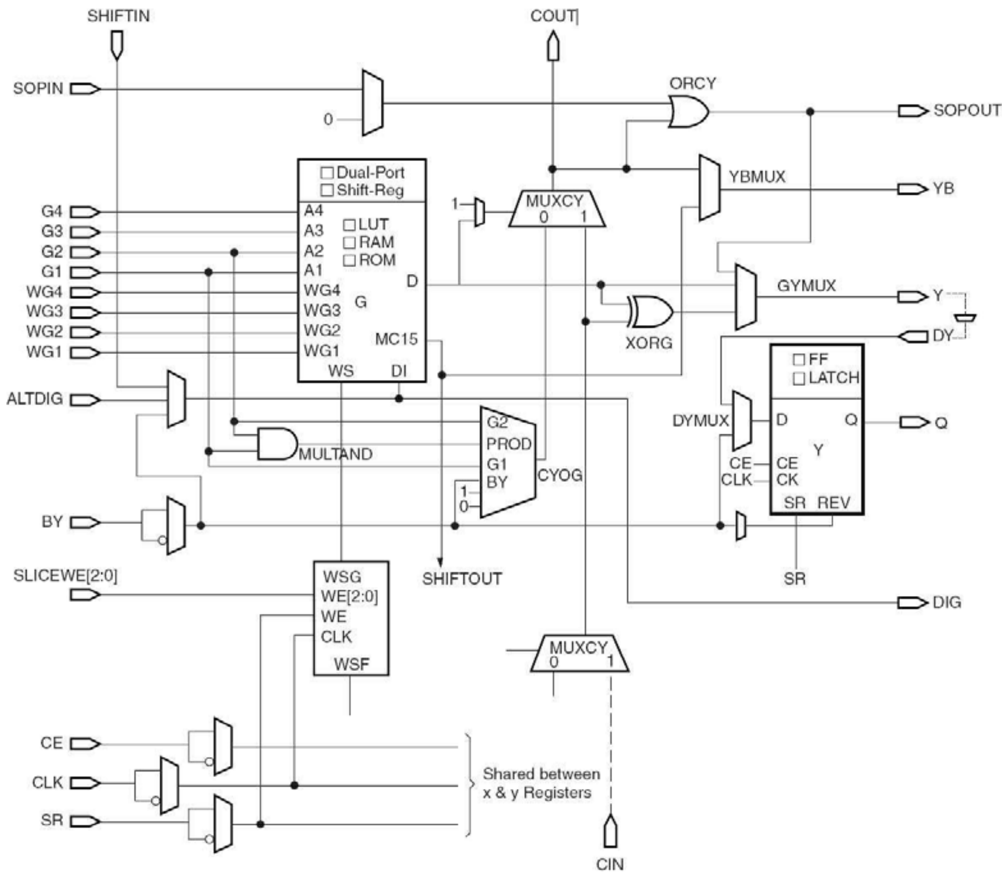


Figure 2.9 *Top half of a Virtex-II slice (Taken from [240]).*

the best trade-off between area and delay [3].

Xilinx also provides two approaches for implementing embedded processing: hardcore and softcore processors. They support three processors: the IBM PowerPC 405 hardcore, the MicroBlaze softcore, and the PicoBlaze softcore processors. The PowerPC hardcore is a high-performance 32-bit RISC processor currently supported by the Virtex-II-Pro family and by the Virtex-4 FX. It integrates a 5-stage pipeline, instruction and data cache memories, a JTAG port, timers, and a memory management unit (MMU). The MicroBlaze softcore [234] is a flexible 32-bit Harvard RISC processor supported by the Virtex and Spartan series. Finally, the PicoBlaze is a compact 8-bit processor provided as a VHDL file. Given its very reduced size, applications can benefit from a very high number of them for massive concurrent processing.

Xilinx FPGAs offer very interesting reconfigurability features: they support dynamic partial self-reconfiguration. Given the importance of dynamic reconfiguration of Xilinx FPGAs for this thesis, it deserves to be more widely covered in section 2.3.

2.2.2.2 Altera

In the same way as their strongest competitor, Altera offers two FPGA series: the high performance Stratix series and the low cost Cyclone series. The Stratix series are composed of several families: Stratix, Stratix-II, Stratix-GX, and Stratix-II-GX. All them share very simi-

lar architectural features. The II versions are basically an upgrade of their predecessors; they provide higher performance and lower power consumption by using a 90nm fabrication technology, and provide a larger and more powerful logic cell. The main difference for the GX versions is the inclusion of high speed embedded communication transceivers of up to 6.375 Gbps.

The Stratix series architecture is composed of a 2-dimensional array of logic cells called logic array blocks (LABs), memory block structures, input-output elements (IOE), and digital signal processing (DSP) blocks. Figure 2.10 depicts the architecture of a Stratix-II FPGA.

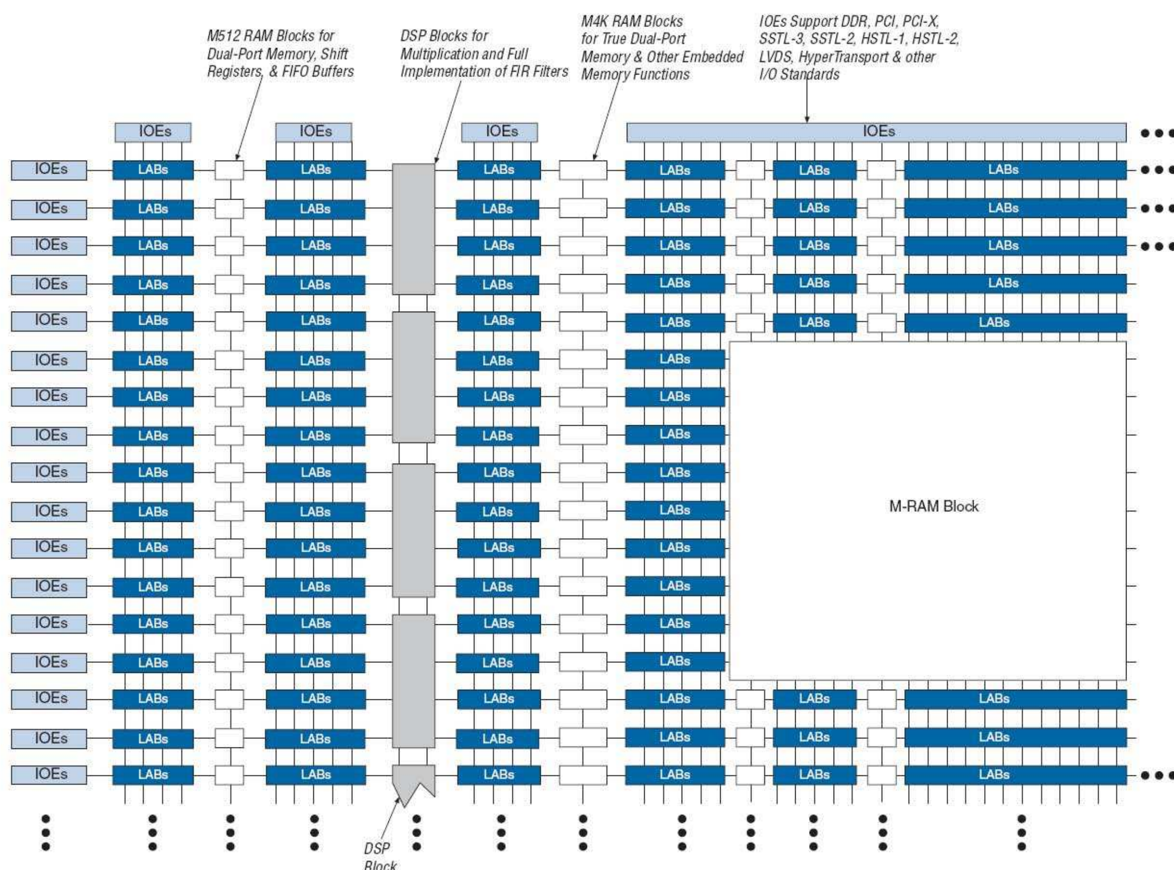


Figure 2.10 *Stratix-II architecture (Taken from [4]).*

The architecture of a LAB differs from the Stratix to the Stratix-II families. In the former, a LAB is composed of 10 logic elements (LE), with each LE basically containing a register, a 4-inputs LUT, some internal signal multiplexing, and carry logic. A LAB, for a Stratix-II, is composed of eight adaptive logic modules (ALM), each ALM being slightly more powerful than 2 of the previously described LEs. Each ALM contains two registers, 2 4-input LUTs, 4 3-input LUTs, carry logic, shared arithmetic chains, and local interconnections.

The Stratix series offer a variety of memory structures: M512 RAM, M4K RAM, and M-RAM blocks. M512 RAM provide dedicated simple dual-port or single-port memory. They are distributed throughout the FPGA, placed in between LABs, and can run at up to 500MHz. M4K RAM blocks provide dedicated true dual-port, simple dual-port , or single-port memory. As M512 blocks, they are placed in between LABs (see figure 2.10), and can run at up to

550MHz. Finally, M-RAM blocks provide dedicated true dual-port, simple dual-port , or single-port memory up to 144 bits wide. They are individually located in the device (see figure 2.10), and can run at up to 420 MHz.

As an embedded processing approach, Altera offers the Nios II, a 32-bits RISC general purpose softcore processor. Altera has also launched, some years ago, the Excalibur devices, a hardcore solution for embedded processing, which is not being supported anymore. They are composed by a 32-bit ARM922T microcontroller and the configurable logic of the APEX 20KE family. Excalibur devices provide interesting reconfigurability features since the FPGA side can be reconfigured independently from the microcontroller execution. The reconfigurable array can be also configured by the ARM922T, constituting a suitable platform for implementing self-reconfigurable systems. However, unlike Xilinx FPGAs, it does not support partial reconfiguration of the logic array, limiting its flexibility.

2.2.2.3 Atmel

Atmel offers two series of FPGAs, the AT6000 and the AT40, both using SRAM technology and similar complexity levels, differing mainly in their logic cell architecture. The AT6000 logic cell use multiplexers and some logic gates for implementing the logic functions and includes a register (with a similar philosophy as the XC6200 from Xilinx); however, this family is not being produced anymore. The AT40 family has been designed with a more standard architecture with 2 3-input LUTs and a register.

Even though the cell's architecture is very simple, they have an interesting feature: the routing. Unlike Xilinx's and Altera's FPGAs, Atmel routing is not implemented with switch matrices but with multiplexers. The inputs to LUTs are multiplexed from its 8 neighbors, while for other multiplexer based routing devices the neighborhood is 4. This feature makes them very well suited for cellular architectures.

The AT6000, AT40K and AT40KAL families have the ability to implement *Cache Logic design*: a section of the FPGA can be reprogrammed without loosing register data, guaranteeing that the unmodified section of the FPGA can continue operating without disruption. This configurability feature can make Atmel FPGA architectures well suited for building modular adaptive systems, or any designs where the datapath can change to increase system performance and provide flexibility.

2.3 Dynamic reconfiguration of Xilinx FPGAs

Xilinx FPGAs support dynamic partial reconfiguration. A critical issue when designing a dynamically reconfigurable system is the design flow used for conceiving it. A typical design flow targets static design, where an unmodified circuit is synthesized and simulated for performing a predefined task. When tackling a problem requiring dynamic reconfigurability, a number of new issues and paradigms arise given the changing nature of the underlying platform implementing the system. How to simulate a system's dynamic partial reconfiguration? How to generate and deal with multiple partial bitstreams? Simulation tools still do not support dynamic modification of the circuit. However, Xilinx proposes two design flows for generating partial reconfiguration bitstreams: *module-based* and *difference-based* [245].

The *module-based* flow allows the designer to split the whole system into modules. For each module, the designer generates a configuration bitstream starting from an HDL description and goes through the synthesis, mapping, placement, and routing procedures, independently of other modules. Each module may be reconfigurable or fixed. A complete initial bitstream must be generated, then partial bitstreams are generated for each reconfigurable module.

Several placement constraints must be met when creating the modules; figure 2.11 depicts an example layout. A module's area is defined by the bottom left and the top right CLBs, always defining a rectangular shape. The module's left bound must always be a multiple of 4, as must the module's width. The module's height must correspond to the full height of the FPGA logic cell array, mainly due to the configuration bitstream format that will be explained in subsection 2.3.2.

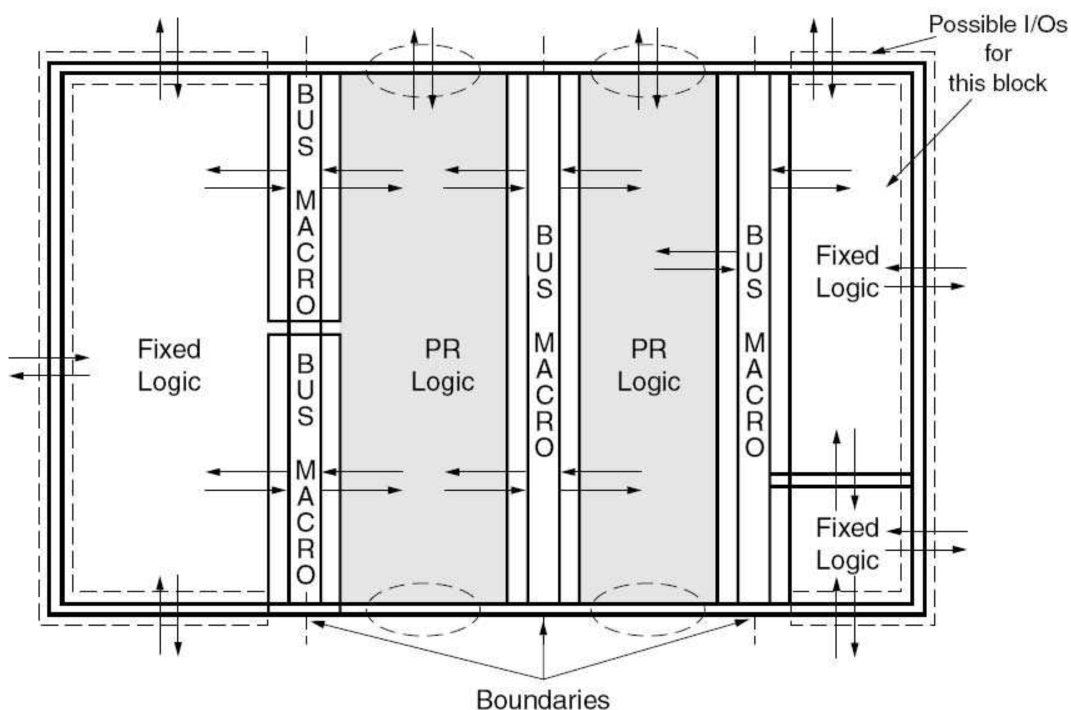


Figure 2.11 *Modular design layout (Taken from [245]).*

Inter-module communication is restricted to neighbor modules and must be done through a *bus macro*. A *bus macro* (figure 2.12) is a hard macro consisting of four lines, each connected to two 3-states buffers, where each buffer is contained in one of the two modules to be connected. Inter-module connection must be done through bus macros for guaranteeing a fixed inter-module channel that remains unaffected during reprogramming.

Access to IOBs also imposes special constraints to the system. As depicted in figure 2.11, IOBs are accessible exclusively to adjacent modules. This constraint also includes global signals, such as the system's global reset, which must be routed throughout the whole system via bus macros. The only exception to this rule is the global clock signal, clocking circuitry has dedicated embedded routing lines whose configuration bitstream is independent from other routing resources. These IOB constraints will be an important issue when designing a PCB

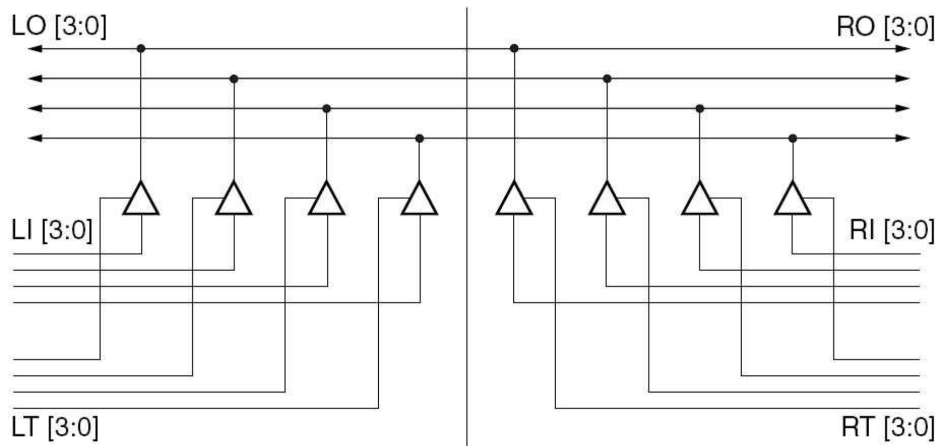


Figure 2.12 *Bus macro implementation (Taken from [245]).*

board layout as will be discussed in Chapters 8 and 9 in the design of the presented reconfigurable platforms.

With the *difference-based* flow the designer must manually perform low-level changes. Using the *FPGA Editor*, a low level design tool, the designer can change the configuration of several components such as: look-up-table equations, internal RAM contents, I/O standards, multiplexers' selections, flip-flop initialization reset values, etc. After editing the changes, a partial bitstream can be generated, containing only the differences between the "before" and the "after" designs.

Lower level partial bitstreams can be generated by using the difference-based flow. Using this technique to modify circuits requires a previous knowledge of the physical placement of the logical components implementing the target function, i.e. the logical function to be modified, in the FPGA. In this case, it is desirable to select where in the FPGA the changing circuitry will be placed. A solution to this problem is proposed in this thesis: cellular systems are designed as an array of hard macros. By using hard macros one can define placement constraints; one can place each hard macro and, knowing LUT positions, one can modify them by using difference-based reconfiguration. Hard macros must be designed by low level specification of the circuit: using the FPGA_editor one can define a system in terms of the FPGA basic components. Every CLB, LUT and flip-flop must be manually placed, and a semi-automatic routing is performed. For instance, section 6.2 presents a hard macro for implementing fuzzy rules, in section 6.3 hard macros are used for implementing cellular automata, and in section 7.2 they are used for implementing random boolean networks.

2.3.1 Self-reconfigurable systems

Even though only these flows are supported by the FPGA vendor, in this thesis I propose a new low-level design flow. Self-reconfigurable platforms generate a special interest in the field of reconfigurable computing, given the autonomy they provide. Virtex-II FPGAs include an *Internal Access Configuration Port* (ICAP), allowing one to read and to write the configuration bitstream from the FPGA. The ICAP allows an on-chip processor to self-reconfigure the FPGA supporting it. Usually, self-reconfigurable platforms modify the system by re-configuring the

FPGA with predefined partial bitstreams [75]. The main drawback of these partial bitstreams is the fact that, using the design flows proposed by Xilinx, they must be pre- placed and routed on a workstation, restricting the number of reconfigurable systems.

An attempt to allow a platform to self-reconfigure with a design description conceived on the fly has been proposed by Xilinx engineers [14]: XPART (Xilinx Partial Reconfiguration Toolkit) is an application program interface (API) for MicroBlaze or PowerPC microprocessors that provides methods to read and modify selected FPGA resources by using the ICAP (Internal Configuration Access Port). Unfortunately, XPART was never released.

Another approach consists in directly modifying the configuration bitstream by knowing in advance which bits in the bitstream must be modified for obtaining a desired circuit. In this way, one can modify logic functions, inter-connectivity, memory contents, and system's initial conditions, on-chip and on-line. The main drawback of this approach is that to do it one must know the bitstream format in advance, and the Virtex-II bitstream format is not documented, so reverse-engineering must be done for obtaining it.

2.3.2 Virtex-II bitstream

Up to now, all described dynamic partial reconfiguration techniques are highly dependant on Xilinx tools, making them restrictive for self-reconfigurable adaptive systems. Directly generating arbitrary configuration bitstreams without using a synthesis and place & route flow is not a very common technique. However, this technique has been used with the XC6200 family and on other custom platforms summarized in 4.3. However, in every case one must maintain a fixed section (i.e. not evolved) in the bitstream. For instance, Thompson in [207], uses an XC6216 with an array of 64×64 logic cells, but the evolved circuit uses just an array of 10×10 logic cells, while keeping fixed input and output. In this case the evolved section of the bitstream is just that section containing the 10×10 array while the sections for IO blocks and the remaining cells are kept constant during the evolution.

Exactly the same principle can be applied for Virtex series, including Virtex, Virtex-II, Virtex-II-Pro and eventually Virtex-4 and Virtex-5: LUT contents can be arbitrarily defined, by keeping a safe predefined fixed routing. By using hard macros, as will be described in subsection 4.3.2, one can describe a computing cell. This computing cell can consist in a neuron, a fuzzy rule, a simple LUT, or any function, including one or several LUTs; it can also include flip-flops for making the design synchronous, or it can just implement combinatorial circuits. LUTs' and multiplexers' configurations can be modified in an arbitrary way; however, routing must remain fixed. Connectivity among components of a computing cell is manually set when designing the hard macro; connectivity among computing cells is defined by an HDL description of the full system. Although routing must remain fixed during evolution, LUTs can be evolved as multiplexers, where the selection is done by the configuration bitstream. An implementation using this principle is described in [213], where the authors present a cellular automata evolution running on a Virtex-E.

For the Virtex family, the XAPP151 [244] describes in a detailed way the configuration bitstream, specifying the functionality of each bit composing the configuration bitstream. However, for the Virtex-II family this documentation is not available and just a limited bitstream description can be found in [241]. A Virtex-II bitstream basically consists in a header and the configuration data.

The header initially contains a synchronization word, and subsequently a set of instructions for reading the FPGA's identification code, erasing the configuration memory, and in general interfacing with the configuration control logic in the FPGA. It supports bitstream addressing for writing and readback, and it allows one to access the different configuration control registers described in [241].

Configuration data in a Virtex-II bitstream has a hierarchical format, and it is initially divided in 3 blocks. Block 0 contains all clocking, IOB, IOI and CLB configuration, block 1 contains BRAM contents, and finally block 2 configures BRAM interconnections.

Then, each block is divided into columns, where each column configures a set of basic components of the FPGA in a vertical manner as described in figure 2.13. From the figure, one can see that in block 0 the first column configures the clock routing, buffering, and management (DCMs), the second and third columns configure the left IOBs with their respective switch matrices interconnections, and from the fourth column we find CLBs' configuration.

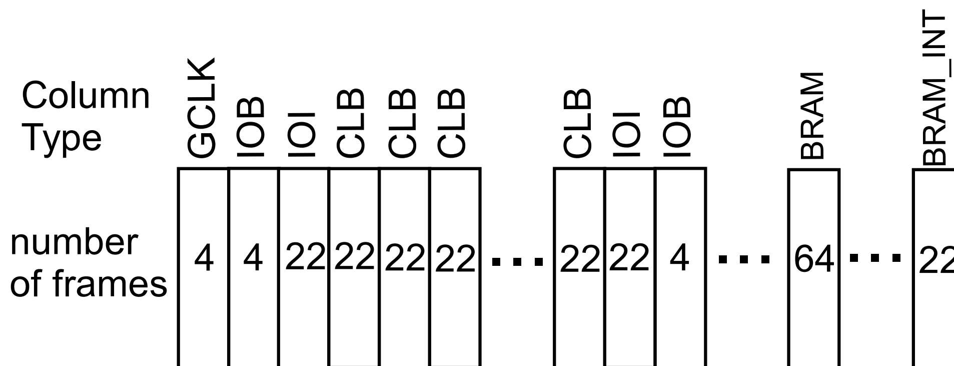


Figure 2.13 *Virtex-II configuration bitstream composition.*

At the same time, each column is composed of vertical frames, with each column type having a different number of frames, as shown in figure 2.13. A frame constitutes the minimum addressable configuration data, consequently constituting the minimum amount of configuration information modifiable when performing partial reconfiguration. This limitation imposes important constraints when designing a reconfigurable system. An example is the module-based design flow, where the height of reconfigurable modules is constrained to the whole device height. In the design flow used in this thesis, this limitation is considered for minimizing the number of frames required when reconfiguring a system.

Going down in the hierarchy of the bitstream format is not possible since it is not documented. Thus there is not information about frame composition, and consequently, no information about LUTs, multiplexers, and routing configuration. However, for this thesis I have worked on reverse engineering the bitstream format for accessing LUTs' and multiplexers' selection configurations.

2.3.2.1 LUT content configuration

The following paragraphs describe how to address LUT contents in a bitstream. In the Virtex-II architecture each CLB has 4 slices arranged 2×2. This arrangement means that each CLB

column has 2 columns of slices. Slices are enumerated in the format X_iY_j, with i from 0 to $2n-1$ beginning from the left (n is the number of CLB columns) and j from 0 to $2m-1$ beginning from the bottom (m is the number of CLB rows). For instance, for an XC2V40 (with array size 8×8) the slice placed at the top left of the component is called slice X_0Y_{15}. Each one of these slices has 2 LUTs, called G-LUT and F-LUT.

Even though the FPGA vendor does not document the configuration format, LUT contents can be localized in the frames of the configuration bitstream by reverse-engineering it. This frame format has been first documented by Sanchez and myself in [221]. As shown in figure 2.13, a CLB column contains 22 frames; the contents for the first slices column LUTs (i.e. with an even X) can be found in the second frame, while for the second slices column (i.e. with an odd X) they are in the third frame. Frame contents are described in Table 2.1. It must be noted also that, as in the Virtex family, LUT configurations are stored inverted (i.e. for a 4-input AND function, LUT contents must be 1000 0000 0000 0000, but are actually stored like 0111 1111 1111 1111 or 7F FF in hex format). Additionally, the bit order is swapped in F-LUTs respective to G-LUTs (i.e. the same AND function in a G-LUT is stored 7F FF in the configuration bitstream, while in a F-LUT function it is stored FF FE).

Based on this description one can determine the position of any LUT content within the bit-stream by applying the following equation:

$$\begin{aligned}
\text{Position} = \quad & \text{Header size} \\
+ \quad & \#\text{GCLK_col_frames} \times \#\text{bytes/frame} \\
+ \quad & \#\text{IOB_col_frames} \times \#\text{bytes/frame} \\
+ \quad & \#\text{IOI_col_frames} \times \#\text{bytes/frame} \\
+ \quad & \#\text{Xcoord_of_CLB_col} \times \#\text{CLB_col_frames} \times \#\text{bytes/frame} \\
+ \quad & \begin{cases} \text{1frame} \times \#\text{bytes/frame} & \text{if slice } X \text{ coord is even} \\ \text{2frames} \times \#\text{bytes/frame} & \text{if slice } X \text{ coord is odd} \end{cases} \\
+ \quad & \text{12bytes} \qquad \text{– IOB config.} \\
+ \quad & \text{5bytes} \times \text{slice_Ycoord (from top)} \\
+ \quad & \begin{cases} \text{0bytes} & \text{if G-LUT} \\ \text{3bytes} & \text{if F-LUT} \end{cases}
\end{aligned} \tag{2.1}$$

Almost all these values are constant for every Virtex-II family device; just the number of bytes per frame (# bytes/frame) depends on the number of CLB rows of the device. The header size is variable, and depends on the configuration options enabled for the bitstream (Details on the header can be found in [241]).

Accessing LUT contents in a partial bitstream is even easier, since one can directly address the target frame by setting the frame address in the bitstream header as described in [241]. Then, the table 2.1 description may be used for directly localizing LUT contents in the frame.

Table 2.1 *Frame description of the LUT's frames. The first 12 bytes configure the IOB, the next 2 bytes configure the G-LUT contents for the top slice, the next byte has an unknown functionality, and the next 2 configure the F-LUT. This sequence is repeated for every slice, and finishes with the bottom IOB configuration. (* Supposing it is the second frame of the first CLB column for an XC2V40.)*

Description	**Size** (# of bytes)
Top IOB	12
Top slice G-LUT (slice X0Y15)	2
–	1
Top slice F-LUT (slice X0Y15)*	2
2nd slice G-LUT (slice X0Y14)*	2
–	1
2nd slice F-LUT (slice X0Y14)*	2
...	...
...	...
Bottom slice F-LUT (slice X0Y0)*	2
Bottom IOB	12

2.3.2.2 Multiplexer configuration

Multiplexers generate a special interest when designing connecting architectures. The main interest stems from the fact that they provide a safe and practical way of playing with arbitrary connectionism systems: they allow the assignment of a single driver for each source (preventing short circuits), and multiple sources for a single driver.

Routing configuration of Virtex-II FPGA is complicated and not documented at all. It is basically composed of switch matrices interconnecting lines through programmable interconnection points (PIPs). Technically, it would be possible to use FPGAs' routing resources to multiplex functions' inputs by activating the correct PIPs. However, reverse-engineering PIP configuration is very complex to do by just comparing some bitstream differences.

LUTs in Virtex-II families can be easily configured as 4-input multiplexers by constraining the possible configurations to:

0000 0000 1111 1111 $\Longrightarrow$ sel = A1
0000 1111 0000 1111 $\Longrightarrow$ sel = A2
0011 0011 0011 0011 $\Longrightarrow$ sel = A3
0101 0101 0101 0101 $\Longrightarrow$ sel = A4

Implementing larger multiplexers requires the use of extra LUTs for multiplexing the new inputs. This is a useful general solution; however, sometimes it can be inefficient. For instance a 5-input multiplexer would use double the LUTs of its 4-input counterpart. In this case it would be more optimal to use the multiplexers present in the FPGA slices for multiplexing the fifth input.

Figure 2.14 depicts the way in which the output of a LUT is connected to the slice output in a Virtex-II slice. By using the same technique used for LUTs, one can control the multiplexers' selection bus from the first and the fourth frame of the corresponding CLB column.

In Virtex-II FPGAs, a CLB is composed of 4 slices arranged as 2 columns $\times$ 2 rows. The

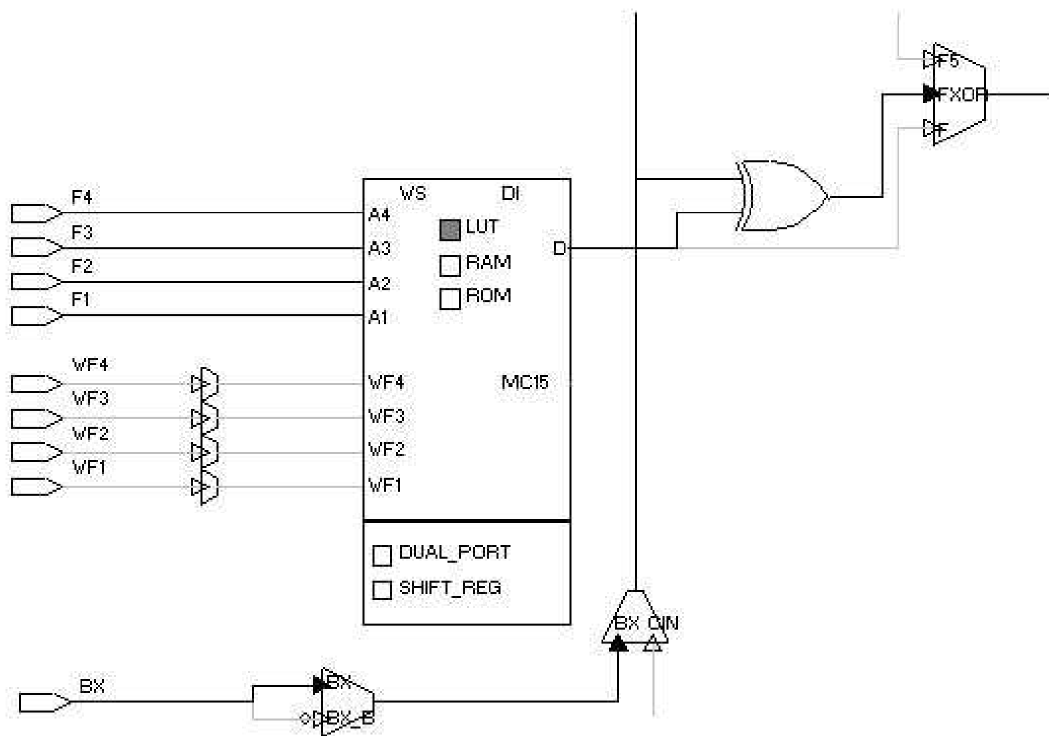


Figure 2.14 *LUT and multiplexer in a Virtex-II slice*

multiplexers' configurations of the first slices' columns is contained in the first CLB frame, while for the second slices' columns they are contained in the fourth CLB frame.

The frame composition is depicted in Table 2.2. The configuration for each multiplexer consists in two bits (the selection lines), each bit contained in a byte (Table 2.2). For G-mux the configuration bits are the second least significant bits in the byte, while for F-mux they are the second most significant bits.

The multiplexer in Figure 2.14 is an F-mux with 3 inputs. The selection table is: $00 \rightarrow F$, $01 \rightarrow F5$, $10 \rightarrow unknown$, $11 \rightarrow FXOR$. For a G-mux the selection table is slightly different for two reasons: it has four inputs, and the order of the configuration bits is inverted. Thus, the selection table is: $00 \rightarrow G$, $01 \rightarrow SOPEXT$, $10 \rightarrow GX$, $11 \rightarrow GXOR$.

By modifying these configuration bits, one can control the multiplexers' selection, enhancing in this way the implementation efficiency of reconfigurable 5-inputs multiplexers. The functionality of the other bits in the frame is still unknown, and they remain unmodified during the system's operation.

For the scope of this thesis, accessing the configuration bitstream, as described in this section, allows modifying circuits in a very flexible way. By defining an initial cellular platform based on an array of hard macros, one can let an on-chip (or off-chip) processor simply modify partial bitstreams just containing LUTs and multiplexers frames. This powerful approach can be greatly improved when combined with the technique proposed in subsection 5.2.2.1, which allows different interconnection grid schemas as well.

Table 2.2 *Frame description of the multiplexers' frame (* Supposing it is the first frame of the first CLB column for an XC2V40.)*

Description	**Size** (# of bytes)
Top IOB	12
Top slice G-mux (slice X0Y15)	2
–	1
Top slice F-mux (slice X0Y15)*	2
2nd slice G-mux (slice X0Y14)*	2
–	1
2nd slice F-mux (slice X0Y14)*	2
...	...
...	...
Bottom slice F-mux (slice X0Y0)*	2
Bottom IOB	12

2.4 Conclusions

Digital electronic devices have experienced tremendous developments during the last decades. These developments have been mainly done in three axes: performance, power consumption, and flexibility.

Performance enhancements have been mainly achieved by two methods: the increasing of operation clock frequency and the design of efficient architectures. At the same time, clock frequency improvements have been achieved by different technological and methodological efforts. Higher transistor integration scales, along with specialized design techniques such as pipelining, branch prediction, cache memories, parallelism, etc., have allowed computers to increase execution performance every year in a quite impressive way.

Increasing clock frequency implies increasing power consumption, becoming a critical issue to the point of dissipating more energy per area than a nuclear reactor! Power consumption issues are currently tackled from several levels: from low-power transistor to low-power software design, including low-power logic gates and architectures.

Finally, flexibility enhancements are the ones which have allowed electronic devices to become omnipresent in our world. Microprocessors, which constitute the core of every computer or embedded system, have achieved their success thanks to their flexibility in the form of *programmability*. It is thanks to programmability that a single mass-produced device can perform a huge variety of applications at a very low cost. The main drawback of such approach is that this programmability has a direct impact on the system performance, since it only supports a sequential set of instructions executed one after the other, independent of the nature of the task at hand. A Microprocessor is tied to a predefined hardware architecture which does not allow programmability at architectural level, but only at instruction-set level.

Reconfigurable logic devices offer an alternative to these limitations exhibited by microprocessors. They offer a wider flexibility in the form of *configurability*. They can support any hardware architecture (within certain complexity limits defined by the device size) by config-

uring the device with a configuration bitstream, allowing one to parallelize the execution of concurrent processes inherent to the task at hand. Additionally, as presented in this chapter, this architecture can be dynamic by modifying it on the fly. Moreover, current reconfigurable devices support self-reconfiguration, where the system itself can modify a section of the architecture implemented on the device.

These devices constitute a powerful and flexible platform for supporting the *dynamically reconfigurable bio-inspired hardware* systems presented in this thesis.

Chapter 3

Bio-inspired Systems

> The problems that exist in the world today cannot be solved by the level of thinking that created them.
>
> Albert Einstein

Living organisms, from bacteria to giant sequoias, including animals such as insects and humans, have successfully survived on earth for billions of years. If one were to propose but one key to explain such a success, it would certainly be *adaptation*. In contrast with nature, adaptation has been very elusive to human technology. The most relevant examples of adaptive systems are not among human's creations, but among nature's. Biological organisms show a striking capacity to adapt to changing circumstances, thus ensuring their continued functionality.

Nature has always stimulated the imagination of humans, but it is only very recently that technology is allowing the physical implementation of bio-inspired systems. They are man-made systems whose architectures and emergent behaviors resemble the structure and behavior of biological organisms [99]. Artificial neural networks (ANNs), evolutionary algorithms (EAs), and fuzzy logic are some representatives of a new, different approach to artificial intelligence. Names like "computational intelligence", "soft computing", "bio-inspired systems", or "natural computing", among others, are used to denominate the domain involving these and other related techniques. Whatever the name, these techniques exhibit the following features: (1) their role models, to different extents, are natural processes such as evolution, learning, development, or reasoning; (2) they are intended to be tolerant of imprecision, uncertainty, partial truth, and approximation; (3) they deal mainly with numerical information processing using little or no explicit knowledge representation.

How to model life? How to integrate all these bio-inspired techniques to a single model? How to merge these techniques in order to create an entity able to mimic living beings? These are open questions that are still far from being completely answered. There exist several research fields deeply studying and proposing computational models of specific aspects of biological systems. Neurocomputing, evolutionary computation, and fault-tolerant systems

are some examples of them. However, modelling life implies including them all in a single model, for which the POE model proposes a well structured framework, which is also well suited to the implementation of real systems.

3.1 The POE model

If one considers life on Earth since its very beginning, then the following three levels of organization can be distinguished [169]: (1) Phylogeny, concerning the temporal evolution of a certain genetic material in individuals and species, (2) Epigenesis, concerning the learning process during an individual's lifetime, and (3) Ontogeny, concerning the developmental process of multicellular organisms.

Analogous to nature, the space of artificial bio-inspired systems can be partitioned along these three axes: phylogeny, ontogeny, and epigenesis; we refer to this as the POE model [169, 189]. The distinction between the axes cannot be easily drawn where nature is concerned. We therefore define each of the above axes within the framework of the POE model as follows: the phylogenetic axis involves *evolution*, the ontogenetic axis involves the *development* of a single individual from its own genetic material, essentially without environmental interactions, and the epigenetic axis involves *learning* through environmental interactions that take place after formation of the individual. As an example, consider the following three paradigms, whose hardware implementations can be positioned along the POE axes: (P) EAs are the simplified artificial counterpart of phylogeny in nature, (O) self-replicating and self-repairing cellular automata are based on the concept of ontogeny, where a single mother cell gives rise, through multiple divisions, to a multi-cellular organism, and (E) ANNs embody the epigenetic process, where the system's synaptic weights change through interactions with the environment. Within the domains collectively referred to as soft computing [132], which often involves the solution of ill-defined problems coupled with the need for continual adaptation or evolution, the above paradigms yield impressive results, frequently improving upon those of traditional methods.

This chapter presents an overview of bio-inspired techniques, contextualizing them in the framework of the POE model. Among these techniques, a special emphasize is put on the ones concerned in this thesis.

3.2 Phylogeny

The first level concerns the temporal evolution of the genetic program, the hallmark of which is the evolution of species, or *phylogeny*. The multiplication of living organisms is based upon the reproduction of the program, subject to an extremely low error rate at the individual level, so as to ensure that the species of the offspring remain unchanged. Mutation (asexual reproduction) or mutation along with recombination (sexual reproduction) gives rise to the emergence of new organisms. The phylogenetic mechanisms are fundamentally nondeterministic, with the mutation and recombination rate providing a major source of diversity. This diversity is indispensable for the survival of living species, for their continuous adaptation to a changing environment, and for the appearance of new species.

EAs, presented in subsection 3.2.1, can be considered to contain every phylogenetic algorithm. Genetic algorithms (GA) and particle swarm optimization (PSO), presented in the two

subsequent subsections, are examples of EAs. Subsection 3.2.4 presents an EA specifically conceived for cellular architectures, more precisely for cellular automata. Finally, subsection 3.2.5 presents Fuzzy CoCo, a coevolutionary approach for evolving Fuzzy Systems.

3.2.1 Evolutionary algorithms

The idea of applying the biological principle of natural evolution to artificial systems, introduced more than three decades ago, has seen impressive growth in the past few years. Usually grouped under the term *evolutionary algorithms* (EAs) or *evolutionary computation*, we find the domains of GAs, evolution strategies, evolutionary programming, and genetic programming [8,36,91,130]. EAs can be also considered as a family of stochastic global optimization algorithms, mainly differing from their deterministic counterparts [153] in lower knowledge requirements of the problem at hand and in the absence of mathematical proofs of convergence given their stochastic nature. For highly non-linear search spaces, EAs have exhibited faster convergence than deterministic methods, given their population-based approach. In most of the cases, the applications solved by EAs can also be tackled with deterministic optimization approaches.

Evolutionary computation makes use of a metaphor of natural evolution according to which a problem plays the role of an environment wherein lives a population of individuals, each representing a possible solution to the problem. The degree of adaptation of each individual to its environment is expressed by an adequacy measure known as the fitness function. The phenotype of each individual, i.e., the candidate solution itself, is generally encoded in some manner into its genome (genotype). EAs potentially produce progressively better solutions to the problem. This is possible thanks to the constant introduction of new "genetic" material into the population, by applying so-called genetic operators which are the computational equivalents of natural evolutionary mechanisms.

The archetypal EA proceeds as follows: An initial population of individuals, $P(0)$, is generated at random or heuristically. At every evolutionary step t, known as a generation, individuals in the current population, $P(t)$, are decoded and evaluated according to some predefined quality criterion, referred to as the fitness. Then, a subset of individuals, $P'(t)$ (known as the mating pool) is selected to reproduce, according to their fitness. Thus, ”good” individuals -i.e. the ones exhibiting high fitness- stand a better chance of ”reproducing”, while low-fitness ones are more likely to disappear.

As they combine elements of directed and stochastic search, evolutionary techniques exhibit a number of advantages over other search methods. First, they usually need a smaller amount of knowledge and fewer assumptions about the characteristics of the search space. Second, they are less prone to get stuck in local optima. Finally, they strike a good balance between exploitation of the best solutions, and exploration of the search space.

EAs are common at present, having been successfully applied to numerous problems from different domains as diverse as optimization, circuit design, disease diagnosis assistance, precision agriculture, self-organizing systems, automatic programming, machine learning, economics, immune systems, ecology, population genetics, studies of evolution and learning, and social systems [91].

3.2.2 Genetic algorithms

Maybe the most representative EA is the genetic algorithm (GA) [227]. A GA (Figure 3.1) is an iterative procedure applied to a constant-size population of individuals. Each individual represents a possible solution to the given problem, and eventually an individual is chosen as the best solution found.

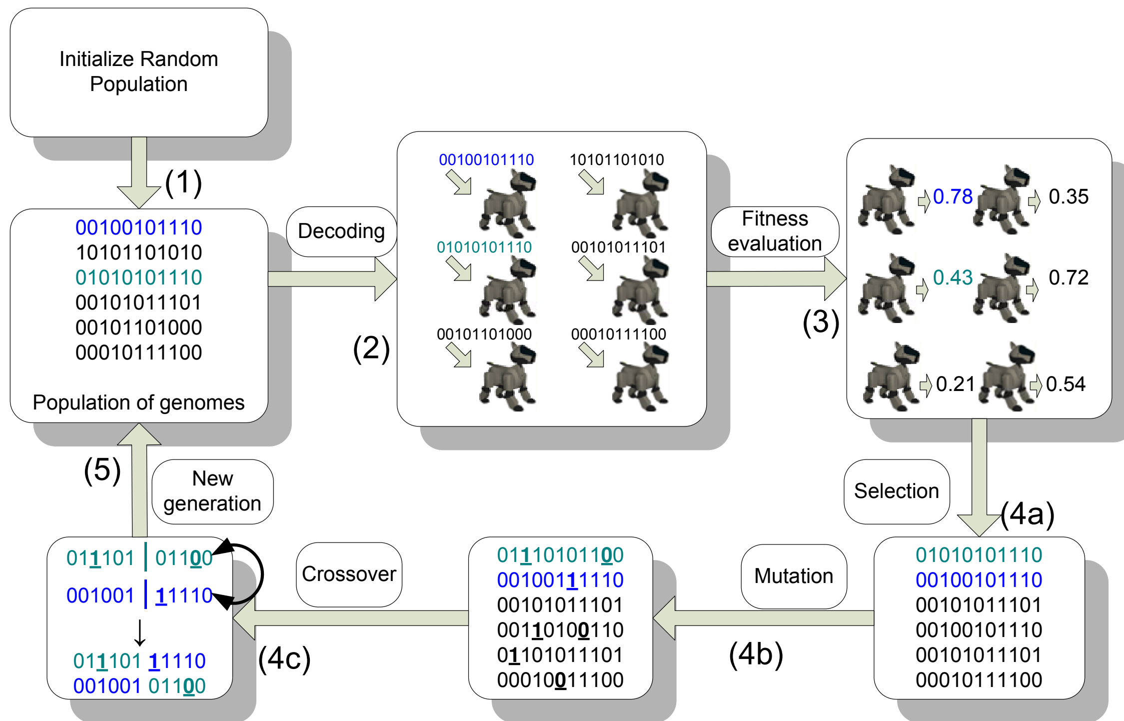


Figure 3.1 *Genetic algorithm. A population is randomly initialized, for being further evolved by applying genetic operators on it.*

Each individual is represented by a finite string of symbols from a given alphabet, known as the genome. Each genome is mapped to a phenotype consisting in a solution to the problem at hand -e.g. a robot controller for the example of figure 3.1. The individual receives a score (fitness) depending on the performance exhibited during its evaluation. The process of going from the genome to a fitness value can be seen as an n-dimensional function (where n is the genome size), and the set of all possible solutions can be seen as an n-dimensional search space. A GA can be summarized in the following steps:

1. Initialization: an initial population of individuals is created by defining a set of genomes in a random or heuristic manner.

2. Decoding: phenotypes for the individuals in the current population are generated by decoding (mapping) the genotypes.

3. Fitness evaluation: individuals are evaluated according to some predefined quality criterion, referred to as the fitness, or fitness function.

4. Genetic operators: genetically inspired operators are applied to the current population
   (a) selection: individuals are selected into a mating pool for reproduction according to their fitness. By using stochastic or deterministic selection mechanisms, the fittest individuals will have more chances to transmit their genetic material to the next generation.
   (b) mutation: symbols in the genome have a probability of being randomly modified
   (c) crossover: two genomes are selected for splitting and swapping at a random position.
5. If a predefined convergence condition has not been met, go back to step 2 for evaluating a new generation. Otherwise, deliver the best individual ever evaluated.

The basic components of GAs are always the same: a population of individuals, a decoding mechanism from a genotype to a phenotype, a fitness evaluation, genetic operators, and an iterative process. However, GAs allows several variants: there exist several methods for defining each one of the above described steps. By running a large enough number of generations, the GA will eventually find an acceptable solution, i.e., one with high fitness.

3.2.3 Particle swarm optimization

Particle Swarm Optimization (PSO) is a stochastic, bio-inspired, population-based optimization method first introduced by Kennedy and Eberhart [87]. The algorithm is founded on the social behavior of certain species and evolutionary psychology, which suggests that sociocognitive individuals must be influenced by their past behavior and the success of their neighbors. Even though it does not inspire directly from natural evolution, it is very often classified as an EA since it involves a population of individuals, whose description is being modified in a stochastic way, by interaction with the other individuals in the population.

PSO has become widely used in engineering and computer science as a problem-solving method and it is now a competitive alternative to other stochastic optimization methods such as GAs and evolution strategies, specially in the optimization of continuous functions. Indeed, one of the first and most successful applications of PSO was to the weight learning on feedforward [34] and product unit [224] neural nets.

In PSO, an n-dimensional search space is explored using a swarm of M particles, seeking to minimize an objective function f. The particles are interconnected according to a given topology. The neighborhood $N(j)$ of the j-th particle is defined as the set of particles connected to it. Two topologies have been traditionally used in the literature: the *lbest* topology and the *gbest* topology. In the *lbest* topology, particles are organized in a circular array, the neighborhood of a particle comprising its adjacent neighbors with or without the particle itself. In the *gbest* topology, all the particles are connected together, so that the neighborhood of every particle is the whole swarm. The type of the topology defines the way information will be exchanged among the particles and the robustness of the algorithm [126].

Three kinds of information characterize each particle in the swarm in a given time step t: its position $\mathbf{x}_j^t = (x_{j1}^t, x_{j2}^t, \ldots, x_{jn}^t)$, its velocity $\mathbf{v}_j^t = (v_{j1}^t, v_{j2}^t, \ldots, v_{jn}^t)$ and its *personal best* (*pbest*) $\mathbf{p}_j^t = (p_{j1}^t, p_{j2}^t, \ldots, p_{jn}^t)$, the best position it has found so far.

At time step $t+1$, each particle calculates its new velocity using a given *velocity update rule*. Traditionally, this update rule takes into account:

1. the particle's velocity at time step t,
2. its personal best, $\mathbf{p}_j^t$ and
3. its *neighborhood best*, the best position found so far by the particle's neighbors.

The neighborhood best is defined for each particle j as:

$$\mathbf{p}_{b(j)}^t = argmin_{l \in N(j)} \left(f(\mathbf{p}_l^t) \right) \tag{3.1}$$

In the case of a *gbest* topology, where the neighborhood of each particle is the population itself ($N(j) = \{1, 2, \ldots, M\}$), the neighborhood best is the same for all the particles and is called the *global best* (*gbest*).

The *inertia weight* update rule [179] modifies the particle's velocity according to:

$$\begin{aligned} \mathbf{v}_j^{t+1} &= w \cdot \mathbf{v}_{ji}^t + \mathbf{U}\left[0, \varphi_1\right] \cdot (\mathbf{p}_j^t - \mathbf{x}_j^t) \\ &+ \mathbf{U}\left[0, \varphi_2\right] \cdot (\mathbf{p}_{b(j)d}^t - \mathbf{x}_j^t), \end{aligned} \tag{3.2}$$

where w is the inertia weight, $\mathbf{U}\left[lower, upper\right]$ is a vector of uniformly distributed random values between $lower$ and $upper$, and φ_1 and φ_2 are acceleration constants usually set to 2. The velocities are usually clamped by means of a damping function $\Gamma(\cdot)$, that is implemented component-wise as follows:

$$\Gamma\left(v_{id}\right) = \begin{cases} v_{max} & \text{if } v_{ji} > v_{max} \\ -v_{max} & \text{if } v_{ji} < -v_{max} \\ v_{ji} & \text{otherwise} \end{cases} \tag{3.3}$$

Loosely speaking, the particle updates its velocity according to a *psycho-social* compromise among the following *behavioral rules*:

1. keep your previous direction,
2. move in the direction of the best position found by you,
3. move in the direction of the best position found by any particle in your neighborhood.

The parameters w, φ_1 and φ_2 control the relative importance of each of these "intentions".

After having calculated its new velocity by applying Eq. 3.2 and 3.3, each particle updates its position by:

$$\mathbf{x}_j^{t+1} = \mathbf{x}_j^t + \mathbf{v}_j^{t+1}. \tag{3.4}$$

The net effect of the application of the updates given by Eq. 3.2 and Eq. 3.4 is the acceleration of the particle towards a random weighted average of its previous best position and its neighborhood's best position [87]. One of the main issues regarding the application of PSO (and any other stochastic optimization heuristic) in a real problem is the so called exploration-exploitation dilemma or how to balance global and local search. Global search (exploration)

is important in early stages of the search in order to escape from local minima and premature suboptimal convergence, while local search (exploitation) is necessary in the later stages in order to fine-tune promising near-optimal solutions found in the global search phase. One of the first approaches to deal with this problem was the introduction of a linear decaying schedule for the inertia weight parameter [180]. In fact, a value of this parameter close to 1 causes the particles to oscillate with grater amplitude, thus forcing them to explore more widely, whereas a relatively small value forces them to focus their search on a narrower region. In [180], a linear decrease from an initial value of 0.9 to a final value of 0.4 was determined to be a suitable schedule for this parameter, allowing more global search at the beginning and more local search at the end of the run.

The use of inertia weight also guarantees the converegence of the system using a wide range of v_{max}, which means that v_{max} is not a critical parameter if an inertia weight is used, giving robustness to the system [87].

Several variations to the original PSO algorithm have been proposed in the literature in order to improve the performance of the standard PSO algorithm, specially to adequately balance global and local search and thus to avoid premature convergence and stagnation. Typically, these algorithmic improvements come at the expense of increasing the computational cost of the original algorithm, making it difficult to implement in embedded applications.

3.2.4 Cellular programming

Cellular Programming is a distributed EA targeting cellular systems. Evolutionary operations are locally performed in each cell of the system, by sharing chromosomes with its neighbors. It has basically been used for finding rules in non-uniform cellular automata.

Cellular automata (CA) are discrete time dynamical systems, consisting in an array of identical computing cells [210, 233]. A cell is defined by a set of discrete states, and a rule for determining the transitions between states. In the array, states are synchronously updated according to the rule, which is a function of the current state of the cell itself and the states of the surrounding neighbors.

Several dimensionalities of cellular arrays are found in literature, mainly for $n = 1, 2$, and 3 (n-dimensional arrays). Among these, 1 and 2 -dimensional arrays arouse a special interest for hardware implementation given the 2-d nature of current integrated circuit technologies. However, further nanotechnological developments should provide 3-d structures for transistor arrays. CA features makes them very suitable for hardware implementations, mainly the simplicity of their basic components, their inherent massive parallelism, and the locality of interactions among basic components (the concept of neighborhood).

Non-uniform CA differ from their uniform counterpart in their state transition rule diversity. The fact that uniform CA constitute a sub-set of non-uniform CA makes non-uniform CA a more general and powerful platform featuring universal computation capabilities [105, 183]. In the same way, this power improvement implies an important drawback: it becomes very difficult to design the set of rules for a CA to solve a particular problem. That's the reason why evolutionary techniques have been used for finding non-uniform CA state transfer rules [20, 131, 138, 185]. Several EAs have been used for non-uniform CA: mainly GAs [131] and cellular programming [20].

In Cellular Programming (algorithm 1), each cell's state transfer rule is coded as a bit-

string, usually known as a genome. This genome implements a rule for computing the next state as a function of the current states of the cell and its neighbors. Each genome is, for instance, composed of 8 bits for CA with neighborhood radius $T = 1$. Instead of using a population of CA as GAs, the cellular programming approach involves a single, non-uniform CA. This fact implies that the final solution would not be an individual selected from a population (like GAs), but the population itself composed of a set of the best individuals.

Algorithm 1 Cellular Programming

```
for each cell i in CA do
    Initialize rule table of cell i
    f_i = 0 (fitness value)
end for
c = 0 (initial configurations counter)
while not done do
    generate a random initial configuration
    run CA on initial configuration for M time steps
    for each cell i do
        compute cumulative fitness
    end for
    c = c + 1
    if c mod C = 0 then
        for each cell i do
            compute nf_i(c) (number of fitter neighbors)
            if nf_i(c) = 0 then
                rule i is left unchanged
            else if nf_i(c) = 1 then
                replace rule i with the fitter neighboring rule, followed by mutation
            else if nf_i(c) = 2 then
                replace rule i with the crossover of the two fitter neighboring rules,
                followed by mutation
            else if nf_i(c) > 2 then
                replace rule i with the crossover of two randomly chosen fitter neighboring
                rules, followed by mutation
            end if
            f_i = 0
        end for
    end if
end while
```

When running the algorithm, initial cell rules are initialized at random. Then, initial states are also randomly initialized; one must let the CA run for M iterations, and then, one must repeat it for a number C of different initial states. There is not a global fitness, as in GAs, but a local fitness for each automaton. Each cell's fitness is accumulated for the C state initializations, computed as a performance measure according to the behavior desired. After computing the fitness, the genetic operators (reproduction, crossover, and mutation) are

applied to genomes. In this algorithm, these evolutionary operators act in a local manner, by limiting the reproduction and crossover operators to use genomes from neighbor cells. The algorithm is driven by nf_i (the number of fitter neighbors of cell i) as indicated in the pseudo-code of algorithm 1.

3.2.5 Fuzzy CoCo

Fuzzy CoCo [150] is a Cooperative Coevolutionary approach to fuzzy modelling, wherein two coevolving species are defined: database (membership functions, MFs hereafter) and rule base. It is based in a family of EAs called coevolutionary algorithms, which take inspiration from the way that several species cooperate or compete against themselves for survival, sharing the same environment. In the algorithm, several evolutions are run in parallel on several populations of different species. Two different species usually have different genome lengths, and use a different genotype-phenotype mapping (at least in artificial evolution). So the coevolutionary case can be seen as a modular system, where each of the modules is evolving separately, and each is, at the same time, interacting with the other modules. Two main approaches are proposed for these algorithms: cooperative coevolution, where the fitness is based on the ability of a species to cooperate with the others, and competitive coevolution, where the fitness rewards the species able to dominate the others.

Fuzzy CoCo applies the principles of cooperative coevolution for evolving fuzzy systems. Fuzzy systems [94] are based on fuzzy logic where, unlike boolean logic, a statement is not constrained to be true or false but it can deal with partial truth. Fuzzy logic provides, in this way, a more accurate model of human reasoning, since it allows representing human-like descriptions of the world: the water can be very cold, cold, warm, hot, or very hot, instead of just measuring the temperate in degrees Celsius. In fuzzy logic, this partial truth is modelled by membership functions (MFs). MFs determine how true or how false an input statement can be, e.g. how hot is the temperature input which values 3°C. These MFs constitute the fuzzification layer in a fuzzy system, since they convert the inputs from real to fuzzy values, and they constitute the first coevolved species in Fuzzy CoCo.

The second species is the inference layer, which defines a set of fuzzy rules having as input the fuzzified values delivered by the fuzzification layer. Individuals of this second species define a set of rules of the form:

if (weather **is** sunny) **and** (temperature **is** hot) **then** (window **is** *open*),

In the same manner that partial truth applies to the fuzzy inputs it applies to the fuzzy outputs. If the membership functions for the premises of the above rule are partially true, the output will be also partially true, and the window will be partially opened.

Finally, a deffuzzification layer is required at the output of the fuzzy system, in order to convert the fuzzy values to real (also called crisp) values (in this case a position for a motor controller, which opens and closes the window).

The two EAs used to control the evolution are instances of a simple GA [227]. GAs apply fitness-proportionate selection to choose the mating pool, and apply an elitist strategy with an elitism rate E_r to allow a given proportion of the best individuals to survive into the next generation. Standard crossover and mutation operators are applied with probabilities P_c and P_m, respectively.

An individual undergoing fitness evaluation establishes cooperation with one or more

representatives of the other species, i.e., it is combined with individuals from the other species to construct fuzzy systems. The fitness value assigned to the individual depends on the performance of the fuzzy systems it participated in. Representatives, or *cooperators*, are selected both fitness-proportionally and randomly from the last generation since they have already been assigned a fitness value. In Fuzzy CoCo, a number of N_{cf} cooperators are selected according to their fitness and N_{cr} cooperators are selected randomly from the population.

3.3 Ontogeny

Upon the appearance of multi-cellular organisms, a second level of biological organization manifests itself. This level constitutes the developmental process of multi-cellular organisms, best known as *ontogeny*. The successive divisions of the mother cell, the zygote, into newly formed cells each possessing a copy of the original genome, is followed by a specialization of the daughter cells in accordance with their surroundings, i.e. their position within the ensemble. This latter phase is known as cellular differentiation. The ontogenetic process is essentially deterministic: an error in a single base within the genome can provoke an ontogenetic sequence that results in notable, possibly lethal, malformations.

Ontogeny comprises several mechanisms of high interest for inclusion in human-designed systems. Self-replication and self-reparation are two key characteristics of living beings that are still far from being implemented in engineered systems with an efficiency comparable to nature. However, some key factors from multicellular beings have been identified for use in the design of ontogenic machines: a cell's function depends upon its relative position, the physical neighborhood is relevant for chemical interactions between cells, time scales are determinant during cellular reproduction, and the fundamental role played by protein's regulation and cell's differentiation, which is driven by regulatory and differentiation genes.

Research projects as *Embryonics* [118] (embryonic electronics) and *POEtic* [202–205] have studied the issues related to hardware implementations of such mechanisms.

The Embryonics project take inspiration from the genome interpretation done by each cell composing living beings. This project aims to build robust integrated circuits endowed with two fundamental properties of living beings: self-repair and self-replication. For achieving this, they propose a hardware system with several levels of organization. The lowest level is a molecule consisting in a multiplexer. The next level consists in a cell, represented by a set of molecules forming a processor with its memory. Then, a set of cells forms an organism, or in hardware, a multiprocessor system. Finally, this organism can itself replicate, generating in this way a population of organisms.

In the POEtic project, the ontogenetic axis is represented by a different approach. They propose a hardware system composed of layers, where each layer implements each one of the three axes of life. The ontogenetic layer implements cellular differentiation, growth, and cellular self-reparation. The differentiation process is performed by a user-defined differentiation algorithm.

3.4 Epigenesis

The ontogenetic program is limited in the amount of information that can be stored, thereby rendering the complete specification of the organism impossible. A well-known example is that of the human brain with some 10^{11} neurons and 10^{14} synapses, far too large a number to be completely specified in the four-character genome with an approximate length of 3×10^9. Therefore, upon reaching a certain level of complexity, there must emerge a different process that permits the individual to integrate the vast quantity of interactions with the outside world. This process is known as *epigenesis* and primarily includes the nervous system, the immune system, and the endocrine system. These systems are characterized by the possession of a basic structure that is entirely defined by the genome (the innate part), which is then subjected to modification through lifetime interactions of the individual with the environment (the acquired part). The epigenetic processes can be grouped under the heading of *learning* systems and, in bio-inspired systems, it is mainly represented by the domain of ANNs.

3.4.1 Artificial neural networks

Artificial neural networks (ANNs) are massively parallel distributed computing units made up of very simple basic elements. They provide the feature of storing experiential knowledge making it available for future use. ANNs takes inspiration from animals' brains in several aspects: they benefit from a massively parallel cellular architecture, a learning process allows acquiring a certain knowledge, and this knowledge is stored in the form of synaptic weights interconnecting neurons. Among other computation features, ANNs provide nonlinearity (an ANN made up of nonlinear neurons has a natural ability to compute nonlinear input-output functions), they are universal approximators (ANNs can approximate input-output functions to any desired degree of accuracy, given an adequate computational complexity), they are adaptable (adjustable synaptic weights and network topology can adapt to its operating environment and track statistical variations), they are fault tolerant (an ANN has the potential to be fault-tolerant, or capable of robust performance, in the sense its performance degrades gradually under adverse operating conditions), and they can mimic real neurons (neurobiologists look to neural networks as a research tool for the interpretation of neurobiological phenomena. By the same token, engineers look to the human brain for new ideas to solve difficult problems) [60].

In other words, from an implementation point of view, an ANN is a system that maps a function from an input vector to an output vector. It consists of a set of simple units which are called artificial neurons. Each neuron has an internal state which depends on its own input vector. From this state the neuron maps an output that is sent to other units through parallel connections. Each connection has a synaptic weight that multiplies the signal travelling through it. So, the final output of the network is a function of the inputs and the synaptic weights of the ANN.

The first neuron model for computing purposes was proposed by McCulloch and Pitts [125] in 1943, consisting in a binary neuron where the addition of the weighted inputs, plus a bias value, generate a binary value as output: the neuron is either active, or not. Since then, several neuron models have been proposed, almost all of them featuring a weighted sum of inputs and an activation function. Variations to this schema include different neural coding

techniques, discrete and continuous delay modelling, neural models oriented to a certain network topology, etc.

In the literature one can find a large number of neural models, ranging from simplistic implementation-oriented to the more complex bio-mimetic models. Maybe the most known neuron model, and one of the most simplistic, is the perceptron, which consists in a weighted sum expressed as:

$$\eta_i(t) = \sum_j w_{ij} x_j(t) + \beta_i \tag{3.5}$$

where $\eta_i(t)$ is the weighted sum for neuron i at time t, $x_j(t)$ is the input value coming from neuron j, w_{ij} is the weight value for the synapse connecting neuron j to neuron i, and β_i is the bias value for the neuron i. For generating the neuron's output, this weighted sum is evaluated by an activation function, which for the perceptron can be, for instance, a sigmoid function (which can be seen as a smooth approach to the step function used by the McCulloch and Pitts model) of the form:

$$y_i(t) = \frac{1}{1 + e^{\frac{-\eta_i(t)}{T}}} \tag{3.6}$$

where T is the slope of the sigmoid function. The sigmoid function as activation function, unlike the step function, is derivable, providing also the possibility of deriving the error function. Deriving the error function is important for supervised learning algorithms, it provides information about the direction and the magnitude of the corrections that must be done to synaptic weights.

In general, learning deals with adjusting synaptic weights, but some algorithms modify also the network architecture -i.e. the network connectionism or the neuron model. Three main types of learning algorithms are identified: supervised, unsupervised and reinforcement learning. The algorithm to be used for a given task is highly dependant upon the previous information about the task, and it is closely related to the task's nature.

Supervised learning is characterized by the presence of an external teacher having knowledge about the environment, represented by a set of examples of input-output pairs. However, the ANN does not know the environment. By trial end error, the teacher performs modifications to the network in order to minimize the error function as depicted in figure 3.2, defining the error as the difference between the *desired* and the *actual* outputs. After being trained, in the absence of the teacher, the ANN should be able to continue providing correct outputs for the previously learned inputs as well as for new unknown inputs, providing knowledge generalization. The most common supervised learning algorithms are the least mean square (LMS) algorithm and the back-propagation algorithm [62] (which is a generalization of the LMS for multi-layer networks). Supervised learning is often used for data classification and non-linear control.

In *reinforcement learning* [199], modifications are done based on a critic's score, which indicates how well the ANN performs, without explicit knowledge about the desired solution. Instead of minimizing an error function, as in supervised learning, reinforcement learning aims to maximize a reinforcement signal. It is, by its own nature, an on-line algorithm that learns from the environment based on its own experience; when it performs well it receive a reward

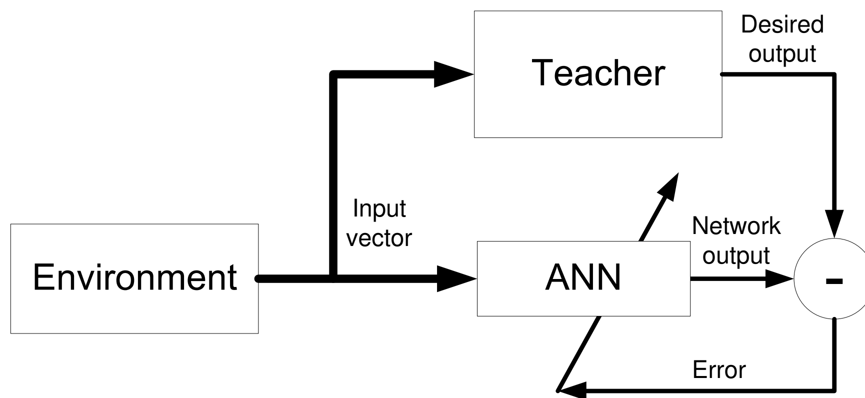


Figure 3.2 *Supervised learning.*

from the critic, and when not it can receive a punishment. Accumulations of rewards and punishments are taken into account when computing the overall reinforcement signal. The most representative applications can be robot navigation and games' strategies (e.g. backgammon, chess).

In *unsupervised learning* there is no information about the task to be performed, synaptic modifications depend on correlations among input data, so the network is intended to identify these correlations without knowing them in advance. It is used for clustering, pattern recognition, and reconstruction of corrupted data, among others. Hebbian learning is the most known unsupervised learning algorithm. In Hebbian learning, synaptic weights are updated solely by local interactions among neurons, and no global or external factors (any teacher or critic) influence this update. It is based on the fundamental principle formulated by Hebb [61] in 1949:

"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency, as one of the cells firing B, is increased".

The choice of a type of learning algorithm is very correlated to the problem at hand, and to the amount of information known in advance. Real implementations can include hybrids of learning techniques for exploiting the features provided by each one of them. For instance, an initial off-line supervised learning can be performed in a robot controller for learning a certain maze solution, followed by the use of an on-line reinforcement learning at run-time for sensor tuning.

3.4.2 Spiking neurons

The human brain contains more than 10^{11} neurons connected in an intricate network. In every volume of cortex, thousands of spikes are emitted each millisecond. Issues like the information contained in such a spatio-temporal pattern of pulses, the code used by the neurons to transmit information, or the decoding of the signal by receptive neurons have a fundamental importance in the problem of neuronal coding. They are, however, still not fully resolved.

Inter-neuronal signal propagation is performed in the form of short voltage pulses called spikes. These spikes travel along the neuron's axon in order to be distributed to several receptive neurons (also called post-synaptic neurons) where they generate post-synaptic potentials. When, in a neuron, several post-synaptic potentials are superposed in a short time window, the neuron's membrane potential may reach a certain threshold value that will fire the neuron, generating a spike transmitted to other neurons, and making the neuron enter into a refractory period, in which new post-synaptic potentials are inhibited.

Most neuron models, such as perceptron or radial basis functions, use continuous values as inputs and outputs, processed using logistic, gaussian or other continuous functions [60]. In contrast, biological neurons process pulses: as a neuron receives input pulses at its dendrites, its membrane potential increases according to a post-synaptic response. When the membrane potential reaches a certain threshold value, the neuron fires, and generates an output pulse through the axon. Biological neurons are extremely complex biophysical and biochemical entities. Before designing a model it is therefore necessary to develop an intuition for what is important and what can be safely neglected.

Among spiking neuron models one can find several levels of abstraction according to the level of biological plausibility; the more biological plausible are usually the more computational expensive. The *Hodgkin and Huxley model* (H&H) [71] describes the generation of action potentials according to the behavior of ion channels and ion current flow in terms of differential equations resulting in input spike responses as shown in figure 3.3. It basically considers three ion channels: sodium (Na), potassium (K), and an unspecific leakage channel (L).

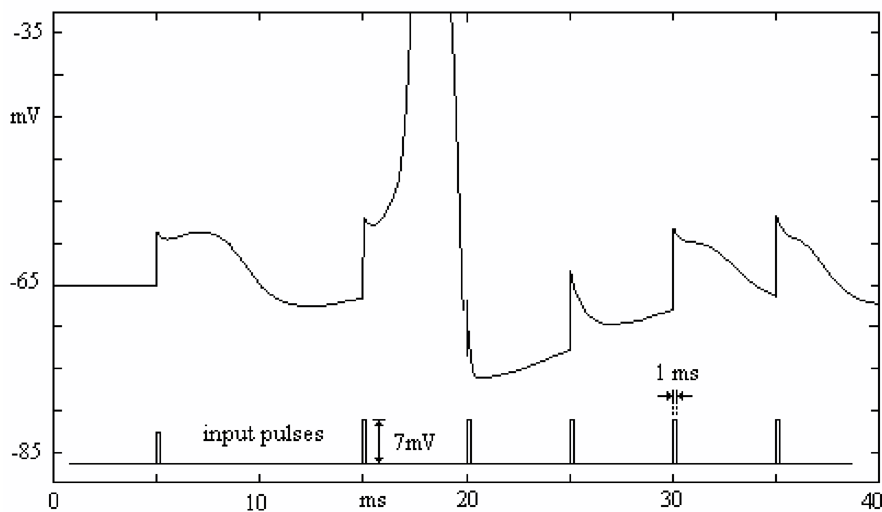


Figure 3.3 *Hodgkin and Huxley model response. The neuron receives input spikes at* $t = 5, 15, \ldots 35$. *The first spike is not strong enough to fire the neuron (i.e. to reach the threshold potential) and generates a postsynaptic response. The spike at time 15 is stronger, and produces a firing; after which, the neuron enters in a refractory period, where the actions of the input spikes are negligible.*

Given the (H&H) complexity, it is not well suited for computational purposes. This is the reason why other simplified approaches are needed. In formal spiking neuron models, spikes are fully characterized by their firing time, allowing simplifications of the neuron dynamics

modelling. Typical examples of spiking neuron models are Integrate-and-fire (I&F) and Spike Response Model (SRM) [40]. These models are suitable for software implementation, where kernels or differential equations represent the neuron response.

The *I&F model* [40, 114], based on a current integrator, models the neuron's membrane potential as the potential of a circuit consisting of a resistance and a capacitor in parallel. The current flowing through this circuit is described by the equation:

$$I(t) = \frac{u(t)}{R} + C\frac{du}{dt} \tag{3.7}$$

where $I(t)$ is the inserted current at time t, $u(t)$ is the membrane potential at time t, and R and C are the constant resistance and capacitance values in the circuit respectively.By introducing the time constant $\tau_m = RC$, it is considered a leaky integrator, and the equation can be rewritten as:

$$\tau_m \frac{du}{dt} = -u(t) + RI(t) \tag{3.8}$$

This equation models the dynamics of the response of the membrane potential to current inputs in the form of spikes. However, for modelling the firing, a firing condition must be introduced by comparing the membrane potential with a threshold potential γ. A firing is, thus, characterized by the condition $u(t^{(f)}) = \gamma$, where $t^{(f)}$ is the firing time. In this way, after a firing at time 0 for instance, the membrane potential can be expressed as:

$$u(t) = RI(t)(1 - \exp(-\frac{t - t(0)}{\tau_m})) \tag{3.9}$$

The *SRM model* [40, 114], instead of using differential equations, express the membrane potential as a sum of kernel functions, where each kernel is the postsynaptic response for an input spike. This model express the membrane potential as:

$$u_i(t) = \sum_{t_i^{(f)} \in \mathcal{F}_i} \eta_i(t - t_i^{(f)}) + \sum_{j \in \Gamma_i} \sum_{t_j^{(f)} \in \mathcal{F}_j} w_{ij}\epsilon_{ij}(t - t_j^{(f)}) \tag{3.10}$$

where $u_i(t)$ is the membrane potential at time t for neuron i, $t_i^{(f)}$ is the firing time of neuron i, $\mathcal{F}_i$ is the set of all firing times of neuron i, Γ_i is the set of presynaptic neurons to neuron i, η_i is a kernel function describing the response of neuron i to its own spikes, w_{ij} represent the synaptic efficacy for the synapse connecting neuron j to neuron i, and the ϵ_{ij} kernels describe the postsynaptic response in neuron i to the presynaptic spikes from neuron j.

SRM is a general model, and kernel descriptions can differ between specific models. I will present some typical examples of kernels. η is a kernel modelling the response of a neuron to its own firing. As illustrated in the H&H model in figure 3.3, after the firing of a neuron it enters into a refractory period, so η is usually a negative kernel since it must guarantee a certain time between two firings, and it must prevent the neuron from firing immediately by providing a refractory period. A typical expression for $\eta_i(s)$ is:

$$\eta_i(s) = -\gamma \exp(-\frac{s}{\tau})\mathcal{H}(s) \tag{3.11}$$

where τ is a time constant, and $\mathcal{H}(s)$ is a step function which for $s < 0$ equals $\mathcal{H}(s) = 0$, otherwise $\mathcal{H}(s) = 1$. The kernel response is illustrated in figure 3.4.a.

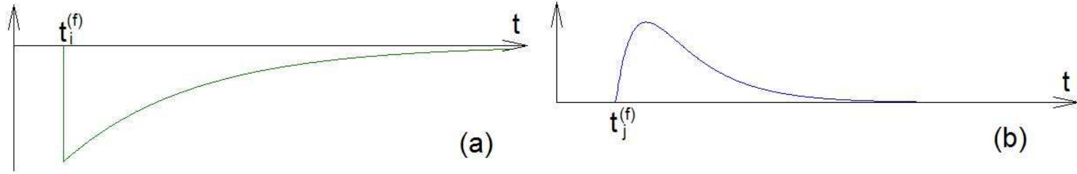


Figure 3.4 *Examples of spike response model (SRM) kernels. (a) Kernel function η_i describing the response of neuron a to its own spikes (equation 3.11). (b) Kernel function ϵ_{ij} describing the postsynaptic response in a neuron following the presynaptic spikes from another neuron (equation 3.12).*

The kernel ϵ_{ij} models the response to presynaptic spikes. It can be positive or negative depending on whether the synapse is excitatory or inhibitory, respectively. A typical example for $\epsilon_{ij}(s)$ can be expressed as:

$$\epsilon_{ij}(s) = [\exp(-\frac{s}{\tau_m}) - \exp(-\frac{s}{\tau_s})]\mathcal{H}(s) \tag{3.12}$$

where τ_m and τ_s are time constants. This expression does not consider the axonal delay, and it must also be noted that it does not contain a scale factor since the amplitude is usually determined by the synaptic weight w_{ij} introduced in equation 3.10. The kernel response is illustrated in figure 3.4.b.

Spiking-neuron models process discrete values representing the presence or absence of spikes; this fact allows a simple connectivity structure at the network level and a striking simplicity at the neuron level. However, implementing models like SRM and I&F on digital hardware is very inefficient, wasting many hardware resources and exhibiting a large latency due to the implementation of kernels and numeric integrations. This is why hardware-oriented models are necessary to achieve fast architectures at a reasonable cost in chip area. Several hardware architectures have been proposed for overcoming this issue. In this thesis, a hardware-oriented spiking neuron model is presented in section 6.1.

How synaptic efficacy is modified in real neurons, remains an active research area still plenty of unanswered questions. However, some adaptation mechanisms have been identified by neuroscientists.

The most widely studied synaptic plasticity mechanism has been the one proposed by Hebb in 1949 [61], where the efficacy of the synapse interconnecting two cells is strengthened when there exists a correlation between the firing of both neurons. The observation reported by Hebb has evolved to a more detailed model of the synapse efficiency dynamics, called Spike-Time Dependent Plasticity (STDP) [40].

The STDP model is based on the time relation of the pre- and post-synaptic firings. If a pre-synaptic spike occurs before a post-synaptic spike, the synaptic efficacy is strengthened. If, on the other hand, it is the post-synaptic spike which occurs before the pre-synaptic one, the synaptic efficacy is weakened. Additionally, the amplitude of the change made to the synapse is inversely proportional to the time between pre- and post-synaptic spikes.

To better understand this concept, and for facilitating its modelling, the concept of learning windows $W(s)$ is introduced. The learning window consists in a function determining the

synaptic modification to be performed as a function of both firing times. Mathematically, a biologically plausible simplified model of the learning window can be expressed as:

$$W(s) = \begin{cases} A_+ \exp[s/\tau_1] & for \quad s < 0 \\ A_- \exp[-s/\tau_2] & for \quad s > 0 \end{cases} \tag{3.13}$$

where τ_1 and τ_2 are time constants, and A_+ and A_- are parameters determining the magnitude of the synaptic modifications, which must be positive and negative respectively. This window definition is the most widely accepted STDP model. However, there are several models proposed. In this thesis, a simplified hardware-oriented model is presented in section 6.1, and its adaptation capabilities are also shown by training a network using the proposed neuron model.

3.5 Hybrid POE approaches

The POE model proposes some guidelines for implementing each one of the above described axes; however, as in biology, the three axes influence each other in a non-obvious way, making it sometimes difficult to discriminate whether an artificial mechanism belongs to one axis or to another. There are also some systems that can benefit from two or from all three axes at the same time, implementing hybrid POE approaches. This thesis focuses on the P and E axes, the ontogenetic axis being out of its scope.

The *PE hybrid* comprises two basic forms of adaptation: at individual and at species level. Adaptation refers to a system's ability to undergo modifications according to changing circumstances, thus ensuring its continued functionality. In this context, learning and evolution are two fundamental forms of adaptation. Evolutionary ANNs are the best example of PE systems, they refer to a special class of ANNs in which evolution is applied as another form of adaptation in substitution of, or in addition to, learning. EAs are applied to the ANN at several levels, depending on the feature to be evolved. These three levels are:

- *Evolution of connection weights*. In this strategy evolution replaces learning algorithms in the task of minimizing the neural network error function. Global search, conducted by evolution, allows overcoming of the main drawback presented by gradient-descent-based algorithms which often get trapped in local minima. It is also useful for problems in which an error-gradient is difficult to compute or estimate. This approach has been widely used, as reflected by the numerous references presented by Yao in [246]. This approach cannot, however, be considered as an example for hybrid PE systems, since the epigenetic axis is simply replaced by the phylogenetic axis.

- *Evolution of architectures*. The architecture of an ANN refers to its topological structure and to the neuron model used. Architecture design is crucial since an undersized network may not be able to perform a given task due to its limited capability, while an oversized one may overlearn noise in the training data and exhibit poor generalization ability. Constructive and destructive algorithms for automatic design of architectures are susceptible to becoming trapped at structural local optima. Research on architectural evolution of neural networks has concentrated mainly on the design of connectivity

[1, 76, 77, 246]. This approach can be considered within the framework of PE, when the topological evolution is accompanied by a learning algorithm.

- *Evolution of learning rules*. The design of training algorithms used to adjust connection weights depends on the type of architecture under investigation. It is desirable to develop an automatic and systematic way to adapt the learning rule to an architecture and to the task to be performed. Research into the evolution of learning rules is important not only in providing an automatic way of optimizing learning rules and in modelling the relationship between learning and evolution, but also in modelling the creative process since newly evolved learning rules can deal with a complex and dynamic environment. Representative advances of this research are [35, 145]. This case is the most evident intersection between the phylogenetic and epigenetic axes, since what is being evolved is the learning mechanism itself.

In addition to these three levels, one can apply EAs to other aspects of ANNs. They can be also used, for instance, for input feature selection. However, in this case the EA is not operating on the ANN itself, but on the data being used for training.

The *PO hybrid* is characterized by systems that can evolve a genome, which may then develop such that it may generate an individual. It implies building a system where a genome can map a solution in an undirected manner, by allowing the genes to express in order to construct an organism. It must be also noted that the search space for the organism would be constrained to a certain set of possible solutions according the ontogenetic mechanism, in the same way that the human genome, formed by 3×10^9 bases, cannot fully describe the human brain, comprising 10^{11} neurons and 10^{14} synapses. This fact forces ontogeny designers to build efficient ontogenetic mechanisms for allowing phylogeny to converge to acceptable solutions. An example of a PO system is the CAM-brain machine of de Garis [29]; he evolves a genome describing the mechanism in which an ANN structure is developed. A common problem faced in EAs is the exponential search space increase when increasing the genome size, Haddow et al. [54] have proposed a PO approach that shrinks the genome size for evolving and developing L-systems.

The *OE hybrid* is composed of systems able to learn and to develop simultaneously. These two axes are the most related. Unlike phylogeny, the O and E axes are constantly interacting during the individual's lifetime, making it sometimes difficult to differentiate the limits among them. In artificial OE systems, one can consider the E axis, the one in charge of updating parameters by means of an incremental learning algorithm, while the O axis is in charge of adapting the topology by growing or pruning the artificial organism. An example of such a system is the work of Perez-Uribe [152], where an ANN implemented in an FPGA features both: a learning algorithm and a mechanism for growing or pruning the network according to the learning results.

Finally, a system containing the three axes, a *POE hybrid*, is a system that must be able to evolve, develop, and learn. Up to now, there has been reported only one hardware platform able to contain these three levels of organization in a single system, it is the POEtic tissue [203, 205]. The POEtic tissue is a self-contained, flexible, and physical substrate, designed to interact with the environment through spatially distributed sensors and actuators, to develop and adapt its functionality through a process of evolution, growth, and learning to a

dynamic and partially unpredictable environment, and to self-repair parts damaged by ageing or environmental factors in order to remain viable and perform the same functionality. It provides, thus, all the mechanisms required for supporting the three axes of life.

3.6 Conclusions

Bio-inspired systems have succeed in mimicking, at least in a very rough way, some very specific aspects of living beings. However, we are still far from fully characterizing and reproducing realistic biological behavior. Some researchers argue that we are just missing the computational power for putting all these principles together. On the other hand, the complexity exhibited by living beings suggests that it is not just a matter of putting a lot of neurons together, but that we are still missing a lot of the underlying mechanisms that allow them to learn, interact, and perceive the environment.

This chapter has presented, in a non-exhaustive way, some of the mechanisms that have been identified by biologists, as being used by living beings for evolving, learning, and developing. However, nature's complexity is still large enough to prevent us from designing human-comparable intelligent machines.

Nevertheless, bio-inspiration can be used for purposes other than reproducing living beings. By extracting some of the biologic principles presented above, one can think about systems exhibiting desired behaviors similar to living beings, such as adaptability, fault tolerance, and generalization. By including these principles in human-designed systems we can improve their utilization lifetime, reuse them for other tasks not foreseen at design-time, and provide them with a certain autonomy for allowing them to take decisions at a certain level.

The POE model provides a framework for integrating several of these features in a single model. By modelling the three axis of life (phylogeny, ontogeny, and epigenesis), the POE model allows the conception of further POE systems exhibiting several of the above mentioned desired characteristics.

Chapter 4

Evolvable FPGAs

A man with a new idea is a crank, until the idea succeeds.

Mark Twain

The space of bio-inspired hardware systems can be also partitioned along the POE axes: phylogeny, ontogeny, and epigenesis. We therefore define bio-inspired hardware for each of the POE axes as follows: the phylogenetic axis involves *evolvable hardware*, the ontogenetic axis involves *hardware implementations of self-replicating and self-repairing cellular automata*, and the epigenetic axis mainly involves *neural hardware architectures*.

The next pages will focus on the phylogenetic axis of hardware bio-inspired systems, most known as evolvable hardware (EHW). Though the scope of EHW includes the evolution of diverse hardware substrates ranging from analog circuits to antennas design, this chapter focuses on evolution of digital circuits by using reconfigurable computing devices, more precisely, FPGAs.

4.1 Evolvable hardware: an introduction

In the case of living beings, adaptation due to evolution is performed through modifications in the DNA (Deoxyribonucleic acid), which constitutes the encoding of every living being on earth. DNA is a double-stranded molecule composed of two sugar-phosphate chains linked together by the base pairs Adenine, Cytocine, Guanine, and Thymine, constituting a string of symbols from a quaternary alphabet (A, C, G, T). On the other hand, reconfigurable logic devices are configured by a string of symbols (the configuration bitstream) from a binary alphabet (0, 1). This string determines the function implemented by each of the programmable components and the connectionism of each of the switch matrices. Under this description, a rough analogy naturally arises between the DNA and a configuration bitstream, and between a living being and a circuit (Figure 4.1). In both cases there is a mapping from a string represen-

tation to an entity that will perform one or more actions: growing, moving, reproducing, etc. for living beings, or computing a function for circuits.

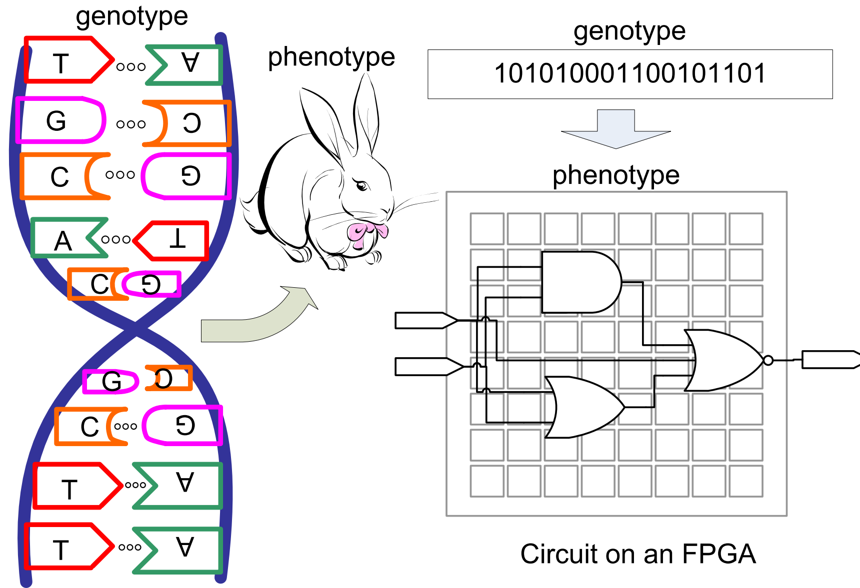


Figure 4.1 *Analogy between living beings and digital circuits.*

This analogy between living beings and digital circuits suggests the possibility of applying the principles of artificial evolution to the field of circuit design (Figure 4.2). Designing analog and digital electrical circuits is, by tradition, a hard engineering task, vulnerable to human errors, and no one can guarantee the optimality of a solution for large circuits. Design automation has become a challenge for tool designers, and given the increasing complexity of circuits, higher abstraction levels of description are needed. EHW arises as a promising solution to tackle this problem: from a given behavior specification of a circuit, an EA will search for a bitstream describing a circuit able to satisfy the specification.

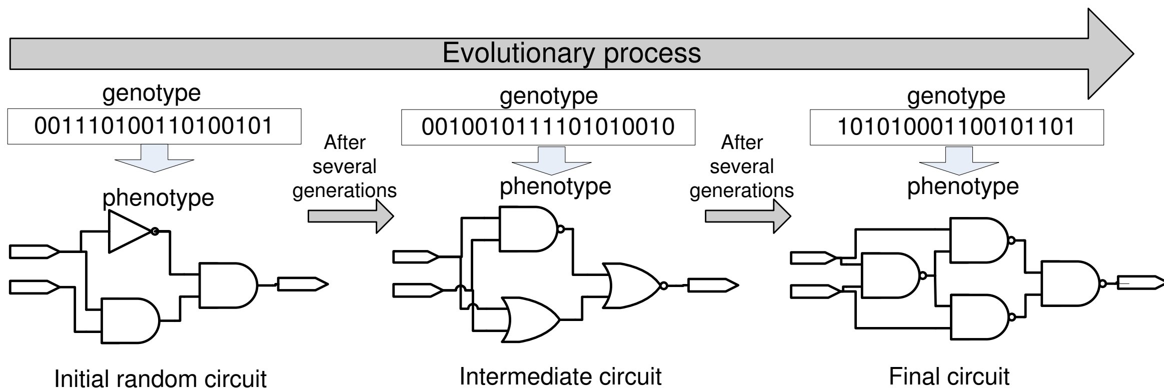


Figure 4.2 *Evolutionary design of digital circuits.*

If one carefully examines the work carried out to date under the heading EHW, it becomes evident that this mostly involves the application of EAs to the synthesis of digital systems

[67, 106, 110, 128, 140, 171, 188, 195, 196, 214, 251]. From this perspective, EHW is simply a subdomain of artificial evolution, where the final goal is the synthesis of an electronic circuit. The work of Koza [91], which includes the application of genetic programming to the evolution of a three-variable multiplexer and a two-bit adder, may be considered an early precursor along this line. It should be noted that at the time the main goal was that of demonstrating the capabilities of the genetic programming methodology, rather than designing actual circuits. We argue that the term *evolutionary circuit design* would be more descriptive of such work than that of EHW (see also [247]). For now, we shall remain with the latter (popular) term; however, we shall return to the issue of clarifying definitions in section 4.2.

EHW, taken as a design methodology, offers a major advantage over classical methods. The designer's job is reduced to constructing the evolutionary setup, which involves specifying the circuit requirements, the basic elements, a decoding mechanism, and the testing scheme used to assign fitness (this latter phase is often the most difficult). If the set-up has been well designed, evolution may then (automatically) generate a circuit performing the desired functionality. Currently, most evolved digital designs are suboptimal with respect to traditional methodologies; however, improved results are regularly demonstrated.

There are two main critical aspects to define when setting up a system to be evolved: how to map a phenotype from a genotype? How to compute the fitness of a circuit? These two issues are very critical and can make the difference between a successful and an unsuccessful evolution. When examining work carried out to date, one can derive a classification of current EHW, in accordance with the genome encoding (i.e., the circuit description), and the calculation of a circuit's fitness.

4.1.1 Genome encoding

- *High-level languages*. Using a high-level functional language to encode the evolving population implies an additional step to obtain the final circuit implementation: the chosen individual must be synthesized. In [91], the evolved solution is a program describing the (desired) multiplexer or adder rather than an interconnection diagram of logic elements (the actual hardware representation). Mermoud et al. [127] use fuzzy rules as evolvable components; and in [143] and [218] the authors propose the evolution of ANN topologies at neuron and layer level. Hemmi et al. [63] used a high-level HDL to represent the genomes. Koza et al. [92] used the rewriting operator, in addition to crossover and mutation, to enable the formation of a hierarchical structure.

- *Low-level languages*. The idea of directly incorporating the bit string representing the configuration of a programmable circuit within the genome was expressed early on by Atmar [6] and more recently by Higuchi et al. [69] and de Garis [28]. As a first step, one must choose a set of basic logic gates (e.g., AND, OR, and NOT) and suitably codify them, along with the interconnections between gates, to produce the genome encoding. An example of this approach is offered in [65]: Higuchi et al. used a low-level bit string representation of the system's logic diagram to describe small-scale PALs, where the circuit is restricted to a logical sum of products. The limitations of PAL circuits have been overcome to a large extent by the introduction of FPGAs, as used initially by Thompson [206, 207], and then by several others research groups.

The use of a low-level circuit description that requires no further transformation is an important step forward since this potentially enables placing the genome directly in the actual circuit, thus paving the way toward true EHW (we shall elaborate upon this point in section 4.2). However, FPGAs had presented two major problems: 1) the genome's length was on the order of tens of thousands of bits, rendering evolution practically impossible using current technology, and 2) within the circuit space, consisting of all representable circuits, a large number were invalid.

With the introduction of the Xilinx 6200 [235] family of FPGAs, these problems were reduced. As with previous FPGA families, there is a direct correspondence between the bit string of a cell and the actual logic circuit; however, for the 6200 case, it always leads to a viable system (i.e., with no short circuits). Moreover, as opposed to previous FPGAs where one had to configure the entire system, the 6200 family permitted the separate configuration of each cell, a markedly faster and more flexible process. Thompson [206] employed this latter characteristic to reduce the genome size, without, however, introducing real-time, partial system reconfigurations. Unfortunately, the production of this FPGA family was discontinued some years later; however, the results achieved by directly evolving its bitstream allowed an important visibility for the EHW community, and made possible the growth of this research field.

4.1.2 Fitness computation

- *Extrinsic evolvable hardware*. The use of a high-level language for the genome representation means that one has to transform the encoded system to evaluate its fitness. This is usually carried out by simulation, and only the final solution found by evolution is actually implemented in hardware.

- *Intrinsic evolvable hardware*. As noted above, the low-level genome representation enables a direct configuration (and reconfiguration) of the circuit, thus entailing the possibility of using real hardware during the evolutionary process.

4.2 Evolvable hardware: a taxonomy

Taxonomy is the science of classifying living organisms into groups by several criteria: common ancestor, structure, origin, etc. The main goal of establishing a taxonomy is to identify an inherent order in nature. Taxonomic classifications allow organizing species and understanding how they are related to each other. In EHW, the phylogenetic axis admits four taxonomic subdivisions (Figure 4.3) according to the level of bio-inspiration: extrinsic, intrinsic, complete, and open-ended evolution.

4.2.1 Extrinsic evolution

At the bottom of this axis, we find what is in essence evolutionary circuit design, where all operations are carried out in software, with the resulting solution possibly loaded into a real circuit. Though a potentially useful design methodology, this falls completely within the realm

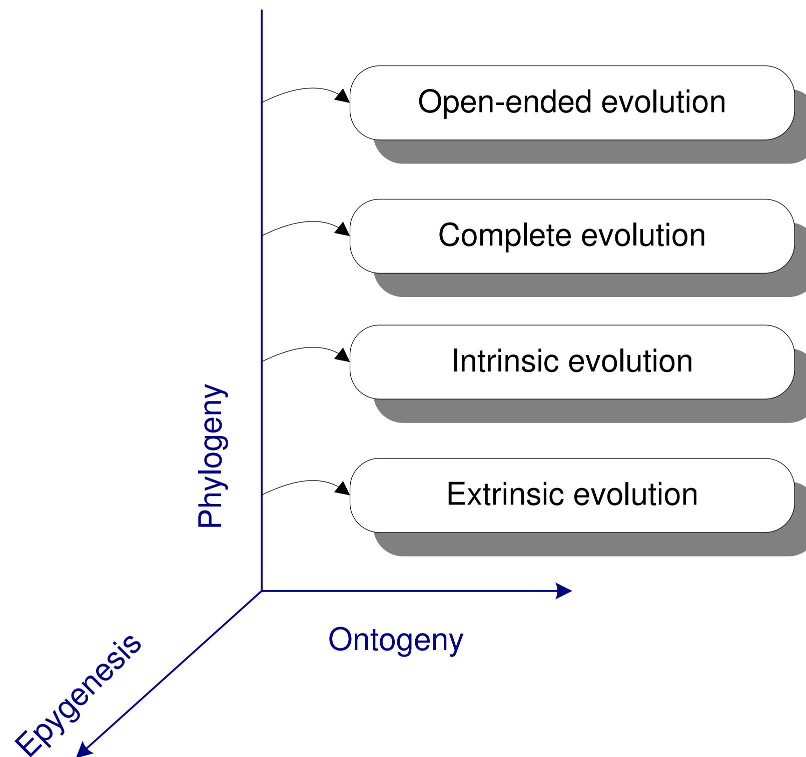


Figure 4.3 *Divisions of phylogenetic hardware.*

of traditional evolutionary techniques. This category is also well known as extrinsic EHW. This approach has typically targeted the synthesis of circuits: from a desired behaviour specification, an EA finds a schematic of a circuit implementing a function that satisfies the specification. This category supports different abstraction levels of description: from gate to HDL level; however, it is not well suited for evolving circuits at bitstream level. Evolution has also been used in other extrinsic aspects of circuit design such as placement and routing [84, 225], and scheduling and allocation problems [112].

4.2.2 Intrinsic evolution

Moving upward along the axis, one finds research in which a real circuit is used during the evolutionary process for fitness computation, though most operations are still carried out offline, in software. This category has been called intrinsic evolution. Examples are [72,127,207,208], where fitness calculation is carried out on a real circuit. Thompson et al. [207] evolved a hardware controller for a two-wheeled autonomous mobile robot for the task of obstacle avoidance. He also evolved an FPGA circuit consisting of 10 x 10 cells, to discriminate between square waves of 1 kHz and 10 kHz presented as inputs. A single real circuit was available with a sequential evaluation of every individual taking place on that circuit, at each evolutionary generation. An interesting aspect of these works concerns the unconstrained use of hardware. While conventional (human) design requires constraints to be applied to the circuit's spatial structure and dynamical behaviour, evolution can do away with these. The circuits evolved

by [207] and [208] had no spatial structure enforced (e.g., limitations upon recurrent connections), no impositions upon modularity, nor any dynamical constraints such as a synchronizing clock or handshaking between modules. Unconstrained circuit design can better exploit the dynamics of the circuit supporting it; however, these circuits exhibit two main drawbacks: (1) impossibility to reproduce a solution: the same bitstream does not behave in the same manner in two different devices, and (2) a high sensibility to external conditions: slight temperature changes can modify the circuit behaviour.

Another example that can be situated within this subdivision of the phylogenetic axis is the works of Murakawa et al. [142] and Iwata et al. [81]. One of the major obstacles which they wished to overcome is that of large genome size (defining the FPGAs configuration). Toward this end they proposed two solutions: 1) variable-length chromosome GA (VGA), where the genome does not directly represent the configuration bit string but rather codifies the possible logical operations and interconnections [81]. A decoder is therefore necessary to translate the genome into an FPGA configuration string. This decoder is, however, much simpler than the compilation tools associated with high-level hardware description languages (such as VHDL); therefore this solution reduces the genome's size without incurring a high computational cost. 2) Evolution at the function level, where the basic units are not elementary logic gates (e.g., AND, OR, and NOT), but rather higher-level functions (e.g., sine-wave generator, multiplier) [142]. Since no such commercial FPGA currently exists, they proposed a novel architecture, dubbed F^2PGA (function-based FPGA). One can combine both solutions, using VGA encoding with an F^2PGA architecture.

It is important to note that while experiments of the above type have been referred to by some as intrinsic evolution, there is a prominent extrinsic aspect since the population is stored in an external computer, which also controls the evolutionary process.

4.2.3 Complete evolution

Still further along the phylogenetic axis, one finds systems in which all operations (selection, crossover, mutation), as well as fitness evaluation, are carried out intrinsically, in hardware. This category has been called complete evolution by Haddow and Tufte [51]. The main motivation is to attain adaptive systems that are able to accomplish difficult tasks, possibly involving real-time behavior in a complex, dynamic environment. The major aspect missing, compared with biological evolution, concerns the fact that evolution is not open ended, i.e., there is a predefined goal and no dynamic environment to speak of. In this category we find two subdivisions: *centralized* and *population-oriented*.

The main characteristic of the *centralized* approach is the existence of a single evolvable circuit and a single evolvable algorithm computation. The centralized approach implements an on-chip genetic machine: a hardwired EA. This description comprises also the implementations where the EA is executed in an on-chip processor. This approach has a special interest since it greatly enhances the autonomy of the circuit, allowing the EHW to adapt to a changing environment during its lifetime. An example of a hardwired EA can be found in [51], where a hardware implementation of a GA, the GA pipeline, evolves a robot controller. Other hardwired EA examples can be found in [120, 176]. Implementations of EAs in general purpose processors can benefit from a more general framework, a more user-friendly interface for implementing chromosome manipulations, fitness evaluations, memory access, etc., and enhance

the possibilities of immediately using the evolving circuit for useful computations. Glette and Torresen [41] report the implementation of a GA on an embedded Power PC processor in a Virtex-II-Pro FPGA, which evolves a circuit in the same FPGA. In this thesis, in sections 6.3 and 7.2 there is reported the implementation of a self-evolvable platform, where the EA is executed on a Xilinx MicroBlaze processor.

A hardware implementation of the full population, and not only of one individual (as was the case for previous categories), is the distinctive feature of the *population-oriented* approach. An example of this approach is the work of Goeke et al. [42], where an evolving cellular system was implemented in which evolution takes place completely on-chip. It is based on the cellular automata model, a discrete dynamical system that performs computations in a distributed fashion on a spatially extended grid. A cellular automaton consists of an array of cells, each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps according to a local, identical interaction rule [210, 233]. The state of a cell at the next time step is determined by the current states of a surrounding neighbourhood of cells. This transition is usually specified in the form of a rule table, delineating the cell's next state for each possible neighbourhood configuration. The cellular array (grid) is n-dimensional, where n = 1,2,3 is used in practice. Nonuniform cellular automata have also been considered in which the local update rule need not be identical for all grid cells [15].

Based on the cellular programming EA of Sipper [186], Goeke et al. [42] implemented an evolving, one-dimensional, nonuniform cellular automaton. The main feature of the cellular programming algorithm is the fact that genetic operators are computed in a distributed way: each automaton modifies its own rule based on its own and its neighbours' fitness. Each of the system's 56 binary-state cells contains a genome that represents its rule table. These genomes are initialized at random, thereupon to be subjected to evolution. The environment imposed on the system specifies the resolution of a global synchronization task: upon presentation of a random initial configuration of cellular states, the system must reach, after a bounded number of time steps, a configuration whereupon the states of the cells oscillate between all zeros and all ones on successive time steps. This may be compared to a swarm of fireflies, thousands of which may flash on and off in unison, having started from totally uncoordinated flickerings. Each insect has its own rhythm, which changes only through local interactions with its neighbours' lights. Due to the local connectivity of the system, this global behaviour involving the entire grid comprises a difficult task. Nonetheless, applying the evolutionary process of [186], the system evolves (i.e., the genomes change) such that the task is solved. This evolving cellular system exhibits complete on-chip evolution, all operators being carried out in hardware in a population-based approach with no reference to an external computer.

In this thesis, in section 9.3, a population-based implementation is also presented. A hardware-oriented particle swarm optimization algorithm is implemented by using a physical existence of each particle.

4.2.4 Open-ended evolution

The last subdivision, situated at the top of the phylogenetic axis, involves a population of hardware entities evolving in an open-ended environment. When the fitness criterion is imposed by the user in accordance with the task to be solved (currently the rule with artificial evolution techniques), one attains a form of guided, or directed, evolution. This is to be contrasted

with open-ended evolution occurring in nature, which admits no externally imposed fitness criterion, but rather an implicit, emergent, dynamical one (that could arguably be summed up as reproducibility). Open-ended undirected evolution is the only form of evolution known to produce such devices as eyes, wings, and nervous systems and to give rise to the formation of species. Undirectedness may have to be applied to artificial evolution if we want to observe the emergence of completely novel systems.

We argue that only the last category can be truly considered EHW, a goal which still eludes us at present. We point out that a more correct term would probably be evolving hardware. A natural application area for such systems is within the field of autonomous robots, which involves machines capable of operating in unknown environments without human intervention [19]. Specifically, the field of collective robotics exhibits a population of individuals interacting in a common environment: they can learn to cooperate or to compete for achieving their goal [19], exhibiting a high level of emergence as a first step to open-endedness. The field of modular robotics, a subtype of collective robotics, offers also a promising open-ended real environment. In chapter 8 of this thesis, a modular robot well suited for evolving distributed hardware is presented: YaMoR, a modular robot composed of mechanically homogeneous modules [134, 135]. Each module contains an FPGA-based system allowing wireless FPGA configuration and on-board self-reconfiguration. Another interesting example would be what is called Hard-Tierra, involving the hardware implementation (e.g., using FPGA circuits) of the Tierra world, which consists of an open-ended environment of evolving computer programs [160]. The idea of Hard-Tierra is important since it demonstrates that open-endedness does not necessarily imply a real, biological environment.

4.3 Evolvable hardware digital platforms

The hardware substrate supporting the evolution is one of the most important initial decisions to make when evolving hardware. The hardware architecture is closely related with the type of solution being evolved. Hardware platforms have, in most cases, a cellular structure composed of uniform or non-uniform components. In some cases, one can evolve the components' functionality; in others the connectivity; or, in the most powerful platforms, both. FPGAs fit well for this third category: they are composed of configurable logic elements interconnected by configurable switch matrices. FPGA's configuration is contained in a configuration bitstream, which contains every function and switch position to be configured for implementing a given design. Currently, FPGAs allow processing partial bitstreams, reconfiguring just a sector of the FPGA while the remaining logic stays unaffected.

When evolving a circuit on an FPGA, one can consider the FPGA logic cell as the basic building block of the circuit to be evolved. One can evolve thus the logic cells configuration and the whole FPGA connectionism by simply considering the configuration bitstream as the genome. However, doing that implies a huge search space to explore and could prevent the EA from finding a solution. A common technique to constrain the search space is to define a basic block as a set of logic cells. In this way each basic block could be an arithmetic operation, an artificial neuron, a fuzzy rule, or a more complex cell in general. Another option is to constrain the connectionism: with layered architectures, constraining connectionism to a certain neighborhood, or just defining a fixed connectionism.

The most basic hardware requirement when evolving hardware is to have a set of high- or low-level evolvable components: the basic elements from which the evolved circuits will be built (transistors, logic gates, arithmetic functions, functional cells ...) and an evolvable substrate supporting them: a flexible hardware platform allowing arbitrary configurations mapped from a genome. FPGAs constitute the perfect hardware substrate, given its connectivity and functional flexibility. This evolvable substrate can be implemented using two main techniques: (1) by using the flexibility provided by the FPGAs configuration logic or (2) by building a virtual reconfigurable substrate on top of the FPGA logic.

The first approach consists in directly generating the configuration bitstream of the FPGA. In this way, one can make a better use of FPGA resources: logic functions are directly mapped in the FPGAs LUTs, and connections are directly mapped to routing switch matrices and multiplexers, incurring the cost of dealing with very low level circuit descriptions [207, 208, 221]. The second approach consists in building a virtual reconfigurable circuit [174] on top of the actual one. In this way the designer can also define his own configuration bitstream and can determine which features of the circuit to evolve. This approach has been widely used by several groups [42, 53, 127, 172, 174, 176, 187, 191, 218, 223, 226, 253], exhibiting enhanced flexibility and ease of implementation, incurring the cost of an inefficient use of logic resources.

Different custom chips have been proposed for this purpose with very interesting results: the main interest in using a custom architecture is that commercial FPGAs are designed for general purpose applications, so they would not necessarily meet the desired requirements for evolvable architectures. Custom evolvable chips usually provide dynamic and partial reconfiguration, dispose of multi-context configuration memories, and can be configured with random bitstreams. The commercial options' main advantage is the absence of non-recurrent engineering, as with any general purpose architecture, incurring the cost of reduced flexibility and performance.

Different chips and platforms have been developed providing the flexibility necessary for evolving analog, digital and mixed circuits; some of them have been designed specifically targeting EHW, while others have just found in EHW another application field. Among these platforms, one can find different levels of granularity, different types of reconfiguration including dynamic and static reconfigurations, the possibility of loading partial configuration bitstreams, and the utilization of context memories.

4.3.1 Xilinx 6200 family

The obsolete Xilinx 6200 family [236] deserves a special mention when referring to EHW platforms. For several years, the 6200 family constituted the perfect platform for intrinsic EHW: any arbitrary bitstream was possible to download without risking contentions, given its multiplexer-based architecture (figure 2.6). Additionally, this FPGA family allowed dynamic reconfiguration, making it more flexible for adaptive algorithms in a general sense.

The most known work using these devices may be that of Adrian Thompson [206–209]. In [208], Thompson et al. evolved a hardware controller for a two-wheeled autonomous mobile robot that was required to display simple wall-avoidance behaviour in an empty rectangular arena. A standard GA was used, with a population of 30 individuals, each one consisting of a 60-bit representation of a dynamic state machine. Thompson [207] evolved an FPGA

circuit, consisting of 10 x 10 cells, to discriminate between square waves of 1 kHz and 10 kHz presented as inputs. Again, a standard GA was employed, with a population of 50 individuals, each one a string of 1800 bits (18 configuration bits per cell), representing a possible circuit. In both cases, a single real circuit was available (one robot in the first experiment and a single FPGA board in the second), with a sequential evaluation of every individual taking place on that circuit, at each evolutionary generation.

The Xilinx 6200 family represents a very important initial stepping stone for the EHW field. It has been also used by other groups for implementing several types of applications. Among these applications we can find cooperative robot controllers [103], sorting networks [93], and image processing algorithms [31].

4.3.2 Evolution on commercial FPGAs

After the disappearance of the 6200 family, many research groups turned to the Xilinx 4000 family. However, this family had an important drawback for evolving hardware: it was not partially reconfigurable, and no arbitrary bitstreams were allowed. When Virtex families appeared, they exhibited two well appreciated features for the EHW community: partial and dynamic reconfiguration. However, not all the evolutionary-friendly features from the 6200 were kept: the connection mechanism does not support arbitrary bitstreams, making devices susceptible to damage by internal short-circuits.

More recent work on evolvable circuits on commercial FPGAs has focused on Virtex and Virtex-II architectures from Xilinx [241], and will certainly extend to Virtex-4 and Virtex-5 in the near future. Two main approaches have been used for evolving Virtex circuits: by using virtual reconfigurable circuits [173], and by partially reconfiguring the FPGA.

4.3.2.1 Virtual reconfiguration

Two solutions were used in order to replace the discontinued 6200 family: implementing an ASIC evolvable circuit (only achievable by some privileged groups, summarized in subsection 4.3.3) and building a reconfigurable circuit on top of another reconfigurable circuit -i.e. a virtual reconfigurable device [174]. The idea of a virtual reconfigurable circuit is depicted in Figure 4.4, where a reconfigurable neuron cell constitutes the basic logic cell of the virtual reconfigurable device.

In the beginning, the most intuitive method was to reconstruct the 6200 architecture. At the University of York a virtual 6200 CLB was implemented in Virtex FPGAs [18, 73]. Slorach et al. [191] also used virtual 6200 cells in Xilinx XC4010 and Altera EPF6010A FPGAs. They evolved configuration bitstreams which did not configure the FPGA itself, but the virtual 6200 CLBs. Afterward, other research groups proposed different reconfigurable architectures with enhanced reconfigurability features. Several virtual reconfigurable devices have been proposed, with the goal of keeping a flexible and easily reconfigurable architecture [32,52,53,172,175,176,191,226,253]. Sekanina et al. [175] proposed a virtual reconfigurable cell called functional block (FB), and used an array of them for image compression. Durbeck and Macias [32] implemented an 8 x 8 Cell Matrix using a Xilinx Spartan-II FPGA.

This approach provides the possibility of designing any desired architecture. In most of the cases the architecture consists in a fine-grained cellular array where a general-purpose

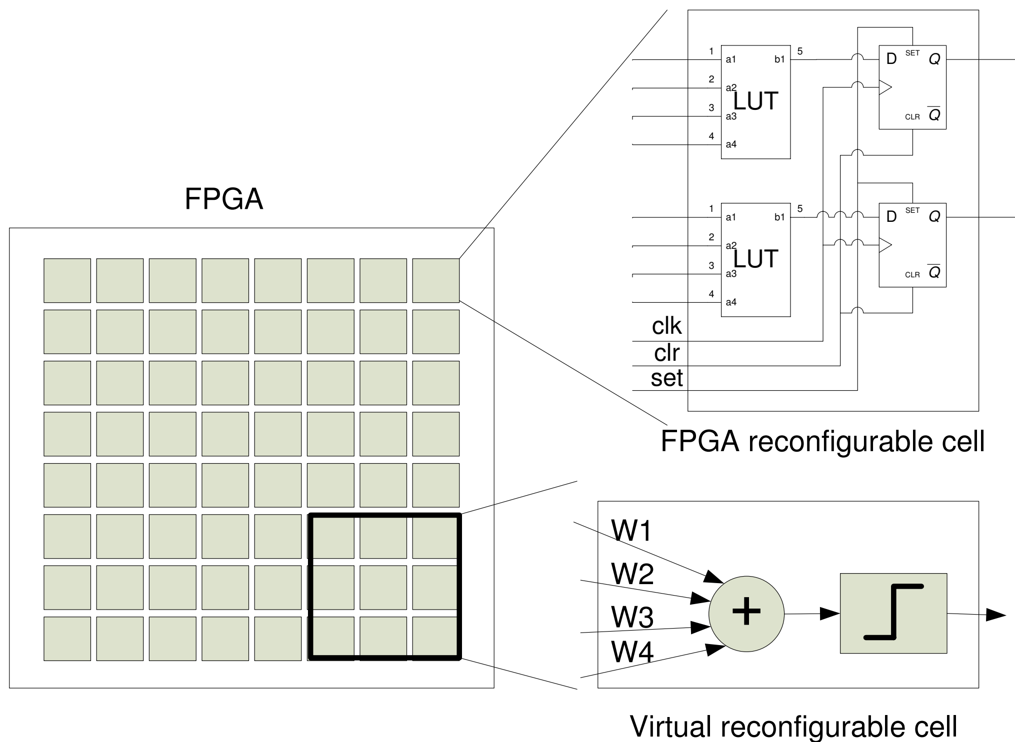


Figure 4.4 *Virtual reconfigurable circuit: a reconfigurable neuron.*

evolvable architecture is proposed. However, more problem-oriented architectures use coarse-grained reconfigurable architectures, where just some features of the architecture are evolved.

The main drawback of this approach is the efficiency in the logic resource utilization. The silicon overhead when implementing a circuit in an FPGA is 40 times the area required to implement the same circuit in an ASIC [96]. Additionally, when implementing a virtual reconfigurable substrate this overhead can be hugely increased. An example is the implementation of the random boolean network presented in subsection 7.2.3, where the overhead of the virtual reconfigurable architecture with respect to the implementation on the actual FPGA is $\times 4.5$, increasing the total overhead to $\times 200$.

4.3.2.2 Dynamic partial reconfiguration

In addition to the work performed on Xilinx 6200 families, other commercial platforms have been partially reconfigured for evolving circuits. The main interest has been focused on Xilinx Virtex families. When evolving circuits by partially reconfiguring Virtex architectures, one must take care not to generate invalid bitstreams (i.e. bitstreams causing internal contentions), and different approaches have been proposed for dealing with that problem.

Haddow and Tufte proposed a 2-d array of Sblocks [52], each of them containing a flip-flop, a 5-input LUT, and some routing resources. Even though the Sblock array is a virtual reconfigurable device, the functionality is reconfigured by partially reconfiguring a Virtex FPGA. They use a partial bitstream for reconfiguring just LUT contents.

At the University of York, Jbits [48] has been used for evolving circuits. Jbits is a Java API

for describing circuits and manipulating configuration bitstreams. Jbits allows one to safely generate partial bitstreams, allowing the modification of internal modules in the FPGA design. At York, LUT contents have been mapped from a genome for evolving simple combinatorial functions [72], fault tolerant circuits [18], and robot controllers for obstacle avoidance [215]. Also using Jbits, Levi and Guccione from Xilinx have developed a tool called GeneticFPGA [104], which from a chromosome translates a configuration bitstream, making it easy to generate legal bitstreams. Even though Jbits provides interesting features for EHW, it has several limitations, such as the impossibility to run on an embedded platform (for on-chip evolution), dependance on supported FPGA families and supported boards, incompatibility with other hardware description languages (HDLs), and a limited support from Xilinx, mainly reflected in insufficient documentation.

In [221], I proposed three methodologies for evolving hardware by partially reconfiguring Virtex and Virtex-II families in a dynamic way (these methodologies may be used also in other Xilinx families as the design tools support their partial dynamic reconfiguration), without using Jbits [48]. Each methodology considers a different level of abstraction and granularity of the basic component used in the evolved circuit. The *modular evolution* methodology is a coarse-grained high level solution, well suited for architecture exploration. The *node evolution* and the *bitstream evolution* methodologies, which are closely related, constitute a fine-grained low level solution, well suited for fine tuning.

- **Modular evolution**

  The modular evolution methodology considers a coarse grain reconfigurable substrate, where the basic blocks are defined at a high functional level. The methodology uses the module based partial reconfiguration design flow described in subsection 2.3. The main consequence of the features of DPR depicted in figure 2.11 is a modular structure, where each module communicates solely with its neighbor modules through a bus macro (Figure 2.11). This structure matches well with modular architectures, such as layered neural networks, fuzzy systems, multi-stage filtering, etc. These systems require a high degree of adaptability, and can benefit greatly from architecture exploration. However, some design constraints must be respected: inputs and outputs of the full network must be previously fixed, as well as the number of layers and their interconnectivity (number and direction of connections). While each layer can have any kind of internal connectivity, connections among modules are fixed through bus macros and restricted to neighbor layers.

  Evolving ANN topologies by using this method is presented in section 7.1. For each module, there exists a pool of different possible configurations. Each configuration may contain a layer topology -i.e. a certain number of neurons with a given connectivity. Each module can be configured with different layer topologies, provided that they offer the same external view (i.e. the same inputs and outputs). Several generic layer configurations are generated to obtain a library of layers, which may be used for different applications.

  An EA is responsible for determining which configuration bitstream is downloaded to the FPGA. The EA considers a full network as an individual. For each application the EA may find the combination of layers that best solves the problem. Input and output

fixed modules contain the required logic to code and decode external signals and to evaluate the fitness of the individual depending on the application.

Two ways of generating bitstreams can be identified by using this methodology: (1) by letting the EA modify HDL or the netlist descriptions of the system, or (2) by pre-placing and routing all the possible modules to be used. The first option, letting the EA modify HDL or netlist specifications, would definitely result in prohibitive execution times: a full placement and routing process should be executed for each individual, which is typically a very heavy computing task. The second option, pre- placing & routing modules, is more accurate for EHW. Under this approach one can see each module as a coarse-grained configurable block that can be configured with a set of predefined components. The EA would select the best combination of components to solve the problem. This methodology fits well for a global coarse search; however, for fine tuning, another adaptation technique must be used.

- **Node evolution**

  The node evolution methodology considers a finer grain reconfigurable substrate than the modular evolution. The basic blocks are defined at a low functional level. This methodology uses the difference based partial reconfiguration design flow described in section 2.3. Using this methodology to modify circuits requires a previous knowledge of the physical placement of the logical components implementing the target function (i.e. the logical function to be evolved) in the FPGA. By using hard macros one can define placement constraints; one can place each hard macro and, knowing LUT positions, one can modify them by using the difference-based design flow. Hard macros must be designed by low level specification of a system: using the FPGA_editor one can define a system in terms of the FPGA's basic components. Every CLB, LUT, multiplexer, and flip-flop must be manually placed, and a semi-automatic routing must be performed.

  Cooperative coevolution of fuzzy systems using this methodology is described in section 6.2, where two hard macros are defined: a parameter macro and a fuzzy rule macro. The function of a parameter macro is just storing a constant parameter. After specifying placement constraints for this macro one can access and modify its contents automatically by using the FPGA editor. In the same way, the fuzzy rule macro can be automatically configured to implement a fuzzy-OR or a fuzzy-AND function (different from their Boolean counterparts).

  For using this methodology, the first step is to define an initial HDL description of the system. This description must include as black boxes the hard macros to be evolved. The hard macros must be designed before the placement and routing process. Placement constraints must be specified for the hard macros, taking care not to overlap them. After placing and routing the design, one must check that hard macros have been placed as desired. The system is now ready to be evolved: an EA written in your favorite programming language will map LUT configuration contents from a chromosome and will run a script for modifying the LUT contents in the FPGA_editor. The result is a partial bitstream, containing just the LUT modifications, which will be generated and downloaded to the FPGA.

  This methodology provides the possibility of fine tuning systems, incurring the cost of

not allowing topological or connectionism modifications. It is well suited for evolving systems with cellular structures, such as neural networks, fuzzy system rules, or cellular automata, among others, with the main drawback of a dependence on Xilinx tools for modifying LUT contents and generating the bitstream. Even though the placement and routing process need not be executed for every individual, it is still not suited for on-system evolution.

- **Bitstream evolution**

  The previously described evolving methodologies are highly dependant on Xilinx tools, making them restrictive for on-chip evolution. The bitstream evolution methodology constitutes the finer grain reconfigurable approach, where modifications are performed at configuration bitstream level. The methodology directly manipulates the configuration bitstream by using the bitstream description of subsection 2.3.2.

  Directly evolving the configuration bitstream has been a very common technique. It has been widely used with the XC6200 family and on other custom platforms summarized in section 4.3.3. However, in every case one must maintain a fixed section -i.e. not evolved- in the bitstream. For instance, Thompson in [207], uses an XC6216 with an array of 64x64 logic cells, but the evolved circuit uses just an array of 10x10 logic cells, while keeping fixed inputs and outputs. In this case the evolved section of the bitstream is just that containing the 10x10 array, while the sections for IO blocks and the remaining cells are kept constant during the evolution.

  Exactly the same principle can be applied for Virtex families, including Virtex-II, Virtex-II-Pro and eventually Virtex-4 and Virtex-5: LUTs' and multiplexers' configurations can be evolved, while keeping a fixed routing. By using hard macros, as described for node evolution, one can describe a computing cell. This computing cell can implement a neuron, a fuzzy rule, a simple LUT, or any function, including one or several LUTs; it also can include flip-flops for making the design synchronous, or it can just implement combinatorial circuits. LUTs' and multiplexers' configurations can be modified in an arbitrary way; however, routing must remain fixed. Connectivity among components of a computing cell is manually set when designing the hard macro; connectivity among computing cells is defined by an HDL description of the full system. Although routing must remain fixed during evolution, LUTs can be evolved as multiplexers, where the selection is done by the configuration bitstream.

  For the Virtex family, the XAPP151 [244] describes in a detailed way the configuration bitstream, specifying the position of LUT contents in the bitstream. However, for the Virtex II family this documentation is not available, and just a limited bitstream description can be found in [241]. In this thesis, the bitstream format description presented in subsection 2.3.2 is used for evolving cellular automata in section 6.3 and random boolean networks in section 7.2.

4.3.3 Custom evolvable FPGAs

Custom evolvable devices offer enhanced reconfiguration capabilities. They provide the required flexibility for allowing an EA to test arbitrary configuration bitstreams. Some of them

additionally provide other reconfigurability features, such as multi-context configuration memories, partial reconfigurability, and dynamic routing, among others. Maybe their main drawback is their cost: their design and construction is very expensive, since the topic remains a research area, and they are not produced in large scale.

One of the more recent evolvable chips is the POEtic tissue [203, 205], a computational substrate optimized for the implementation of digital systems inspired by the POE model. The POEtic tissue is a self-contained, flexible, and physical substrate designed to interact with the environment through spatially distributed sensors and actuators, to develop and adapt its functionality through a process of evolution, growth, and learning to a dynamic and partially unpredictable environment, and to self-repair parts damaged by ageing or environmental factors in order to remain viable and perform the same function.

Higuchi's group has developed an evolvable LSI chip [80], which includes a GA unit, and the ability to process two chromosomes in parallel. Higuchi's group is famous for the large number of applications implemented in their chips [66,68], ranging from prosthetic hand controllers [85,86] to data compression [167,168], including robot navigation controllers [88] and low power integrated circuits [200] among others.

This chapter has mainly focused on evolution for digital devices; however, several platforms have been also proposed for analog and mixed signal circuit evolution. At the Jet Propulsion Laboratory (Caltech), a field programmable transistor array (FPTA) [198] has been developed, which is the basis of the stand-alone board-level evolvable system (SABLES) [197]. Layzell [102] proposed the evolvable motherboard: a diagonal matrix of analogue switches, connected to up to 6 plug-in daughterboards, which contain the desired basic elements for evolution. Also targeting analogue circuits, Zebulum et al. presented PAMA [252], a programmable analogue multiplexer array.

Other platforms, such as PIG [117], FIPSOC [139], MorphoSys [111], DREAM [9], and Palmo [55], were not initially designed for bio-inspired systems. However, the flexibility and performance offered by their architecture fits well with the EHW requirements.

4.4 Conclusions and future directions

EHW has shown to be effective for finding solutions for real world applications [66, 68]. Additionally, some solutions have been shown to perform better than their engineered counterparts [66, 177, 200]. On the other hand, EHW performs poorly, in general, for generating solutions at system level: microprocessor architectures, for example, are not among evolution results. As a matter of fact, evolution works better when targeting complex cellular architectures: cellular automata, neural networks, or gate arrays.

Examining work carried out to date, we find many common characteristics that span most current systems, often differing from biological evolution (though this difference is not necessarily disparaging):

- Evolution pursues a predefined goal: the design of an electronic circuit is subject to precise specifications. Upon finding the desired circuit, the evolutionary process terminates.

- In very few cases does the population have material existence. At best, in what has been called intrinsic and complete evolution, there is typically a single circuit available, onto

which individuals from the population are loaded one at a time, to evaluate their fitness.

- The absence of a real population in which individuals coexist simultaneously entails notable difficulties in the realization of interactions between "organisms." This usually results in a completely independent fitness calculation, contrary to nature which exhibits a co-evolutionary scenario.

- The different phases of evolution are carried out sequentially, controlled by a central unit.

These limitations suggest that the simple application of EAs to hardware design is not enough, and future trends in EHW must not be limited to exploration of EHW architectures and substrates; there is also a lot of work to do at algorithmic level. Man-made adaptable systems are still far from exhibiting an adaptation comparable to living beings, and even though we are still far from attaining circuits of equivalent complexity, limitations are not just a matter of magnitude. Only by modeling together the 3 axis of life (phylogeny, ontogeny, and epigenesis) we will be able to build systems featuring nature-like adaptation.

Chapter 5

Bio-Inspired Adaptation Components

> Just because something doesn't do what you planned it to do doesn't mean it's useless.
>
> Thomas Alva Edison

In the POE model, one can identify two entities on which adaptation can be performed: at species level, and at individual level. Adaptation at species level, also known as evolution, refers to the capability of a given species to adapt to an environment by means of natural selection and reproduction. Adaptation at individual level, also known as development and learning, refers to the cellular development and the behavioral changes in an individual, performed during its lifetime while interacting with an environment. Artificially, adaptation refers to modifications performed to a system in order to allow it to execute a given task.

In both artificial and biological systems, adaptation implies modification. These modifications are presented in several forms depending on the substrate being modified and the mechanism driving the modification. One can roughly consider (maybe very roughly!) that epigenesis mainly involves parametric modifications which concern exclusively an individual's adaptation (such as synaptic efficacies learning in neural systems), while morphological modifications concern both entities (individuals and species), and are mainly driven by phylogeny and ontogeny.

Although artificial adaptation has been widely studied, in contrast with nature, adaptation has been very elusive to human technology. A very representative example of artificial adaptability is that of ANNs one of the most common machine learning techniques. In this case, adaptability refers to the modification performed to an ANN in order to allow it to execute a given task. Several types of adaptability methods can be identified according to the modification done. The most common methods modify the synaptic weights [61, 62] and/or the topology [161, 246].

The single synaptic-weight modification is the most widely used approach, as it provides a smooth search space, making it well suited for gradient-descent algorithms. On the other hand, the single topology modification provides a highly rugged landscape of the search space

(i.e. small changes on the network may result in very different performances). Even though topological adaptation techniques substantially explore the space of computational capabilities of the network, it is very difficult to converge to a solution.

A hybrid of both methods can achieve better performance, since the weight-adaptation method contributes to smoothing the search space, while keeping the advantages of topological exploration. Growing [152], pruning [161], and evolutionary algorithms (EAs) [246] are adaptive methods widely used to modify an ANN topology that will, in association with weight modification, eventually converge to better solutions than the ones found without the hybrid. This thesis thus presents a hybrid adaptation framework, where a structural adaptation is performed by modifying the system's topology, allowing a wider exploration of the system's computational capabilities. The evaluation of these capabilities is further done by a parametric adaptation (synaptic weight learning in the case of ANNs), finding in this way a solution for the problem at hand.

This thesis considers these adaptation aspects for being specifically implemented in partially reconfigurable devices. For doing this this thesis propose a methodology containing two main components: a *computation engine* and an *adaptation mechanism*. The computation engine implements the function constituting the final system solution, while the adaptation mechanism is the technique that modifies the computation engine in order to find the target system. These two concepts are further developed in the next sections.

5.1 Computation engine

Universal computation refers to the ability of a machine to simulate any arbitrary computation on a conventional computation model. The complexity of the required machine can vary according to the complexity of the desired computational task. The computational task is highly dependent upon the number of inputs and outputs, and upon the non-linearities of the desired solution.

Among the most famous universal computers one can find boolean functions and Turing machines. However, in the literature there is a very large number of machines claiming universal computability and, among then, one can find a number of bio-inspired machines. Some of these are ANNs [74,115,156,181], fuzzy systems [90,250], cellular automata [105,183,233], and random boolean networks.

In this thesis, the *computation engine* is the hardware implementation of a given function, and it can be implemented in the form of one of the aforementioned universal computers, as well as in the form of non-universal ones. The computation engines have a physical existence in hardware, and even though universal computability is not mandatory, it is a very appreciated feature when searching a solution since it guarantees that a solution actually exists. In this thesis, these computation engines are presented in the form of neural networks, fuzzy systems, cellular automata, and random boolean networks.

Several works have been done for proving the universality of neural networks. Pollack [156] proved that a recurrent net model, which he called a ”neuring machine” for ”neural Turing”, is universal. Siegelmann and Sontag [181] also showed the existence of a finite neural network, composed of sigmoidal neurons, able to simulate a universal Turing machine. Moving to more biologically plausible neron models we find the work of Maass and Markram

[115], who have studied the computational power of different types of spiking neural networks, proving their universal computing abilities.

For fuzzy systems, Kosko [90] proved that they can uniformly approximate any real continuous function on a compact domain to any degree of accuracy. Latter, Ying and Chen [250] studied the necessary conditions for allowing fuzzy systems to behave as universal approximators.

Cellular systems also arouse a great interest as universal computing machines, given their scalability and their friendliness toward hardware implementation. Wolfram has deeply studied the computational capabilities of cellular automata [231, 233] as well as their universality [232], achieving a significant characterization of their computing properties. Maybe his most relevant result is the proof that CA with rule 110 is a universal Turing machine [233]. For the non-uniform cellular automata case, Sipper has shown that two-dimensional, 2-state, 5-neighbor, quasi-uniform CAs can attain universal computation [184]. In the same way, random boolean networks, being a super-set of non-uniform cellular automata, inherit the properties of them for allowing a more complete computing framework.

5.2 Adaptation mechanism

An important issue for all of the previously presented computation engines is how to find a solution for a specific problem. The proof of their universal computability guarantees that a solution exists, but how to find it is an open issue. How to find the correct configuration of NAND gates (accepted as universal approximators) for implementing a specific boolean function? How to determine the synaptic weights of a recurrent spiking neural network for discriminating between two pattern classes? How to connect a random boolean network and which rules to use for achieving the desired dynamics? These issues must be addressed by an adaptation mechanism.

The adaptation mechanism provides the possibility to modify the function described by the computation engine. Two types of adaptation are allowed: *structural* and *parametric*. The first type takes advantage of the reconfigurable characteristic of FPGAs and intends to modify the architecture of the computation engine by means of partially reconfiguring the FPGA. The second type does not modify the structure, but the values of some registers for parameter tuning and enabling (or disabling) some module functions without involving structural changes.

5.2.1 Parametric adaptation

Parametric adaptation involves relatively minor system modifications. A typical example is a neural network synaptic weight, simply consisting in a value stored in a register. Referring to the previously mentioned bio-inspired systems, it can also involve the modification of a fuzzy inference rule, or the modification of a cellular automata rule.

From the hardware system point of view, in this thesis I propose two methods for performing these changes. The first method, the traditional one, is to define a register at system level whose content can be directly updated by the computation engine itself. The second method also involves a register, but in this case the content update is done through the configuration port of the FPGA. This second approach allows a reduced amount of logic in the

implementation, since it exploits the underlying configuration logic of the device for accessing the register.

The most important issue when modifying these parameters is the criteria used for doing it. According to the computation engine used, there are different algorithms that allow one to tune parameters in an efficient way. Among these parameter-tuning algorithms one can find *engine specific* and *general purpose* algorithms.

A typical example of *engine specific* algorithms is the synaptic weight adaptation in ANNs. For instance, the back-propagation learning algorithm [62] exhibits a good efficiency for perceptron-based neural networks. However, the same algorithm may not be well suited for other artificial neuron models, for example, supervised synaptic-weight tuning is an issue that has not been fully solved on spiking neuron models. Another example, maybe less specific than back-propagation, is the hebbian learning. Hebbian learning modifies the synaptic weight w_{ij}, by considering the simultaneity of the firing times of the pre- and post-synaptic neurons i and j.

On the other hand we find *general purpose* algorithms. These algorithms are supposed to work in any set of parameters, independently of the computation engine used. These algorithms are known as global optimization algorithms and they include a very wide range of approaches. The most naive algorithms are the exhaustive search (where every possible combination of parameters is evaluated) or random search (where the chosen solution is the best of a randomly generated set of parameters). These techniques do not use any information about previously evaluated parameters in order to generate further parameters. Among the heuristic algorithms one finds two large families: (1) deterministic algorithms [153] are based on deterministic search techniques; these algorithms guarantee convergence under some specific conditions meting; and (2) stochastic algorithms, among which we find the bio-inspired search algorithms presented in chapter 3.

5.2.2 Structural adaptation

Structural adaptation involves relatively major system modifications. A typical example is that of ANN topology exploration. Referring to the previously mentioned bio-inspired systems it can also involve the number and the order of fuzzy inference rules, the size of a CA array, or the connectivity of random boolean networks. The goal in this thesis is to take advantage of the reconfigurability of FPGAs for dynamically modifying the architecture of the computation engine by means of partial reconfiguration.

From the hardware system point of view, in this thesis I propose two methods for performing these changes. The first method, consists in a modular exploration, determining a coarse-grained partial reconfiguration by exploiting the modular design flow proposed by Xilinx explained in section 2.3. The second method, instead of modifying large modules, modifies only connections in a more fine-grained manner. Both cases use the FPGA's partial reconfiguration property in order to perform the desired architectural modifications.

5.2.2.1 Coarse-grained topology adaptation

The modular approach consists in having pre-designed generic modules, which will be used through the adaptation process. By using the partial reconfiguration modular design flow

[245], one must build a set of generic architectures that would be useful for finding a correct solution. An adaptation algorithm will determine which is the best ensemble of modules for fitting the desired solution.

This adaptation technique mainly targets layered architectures, given the placement constraints imposed by the modular design flow. Among the computation engines that can benefit from this method, one can list layered ANN, fuzzy systems, multistage filters, and polynomial approximators.

As an example, in section 7.1, the case of layered spiking neural networks is considered. For each layer, there exists a pool of different possible configurations. Each configuration describes a layer topology (i.e. a certain number of neurons with a given connectivity). Several generic layer configurations are generated to obtain a library of layers, which can be used for different problems. A GA determines which combination of layers fits best for a given application.

5.2.2.2 Fine-grained topology adaptation

The connection-oriented approach allows one to perform small modifications in the architecture's topology. It allows one to change the source of a given net by dynamically reconfiguring the hardware supporting it. The hardware substrate supporting it consists in an interconnected multiplexer array, forming a connection matrix.

In section 7.2, the partial reconfiguration of this reconfigurable matrix is used in order to generate random boolean network connections. However, the same connection matrix can be used for randomly connected ANNs which have been shown to perform very well in several tasks. Jaeger and Haas [82] have shown the suitability of the so-called echo state networks (randomly connected recurrent ANNs) to predict chaotic time series, improving accuracy by a factor of 2400 over previous techniques. Maass et al. [116] present their liquid state machines (randomly connected integrate and fire neurons), able to classify noise-corrupted spoken words.

Hardwired random connectionism is very expensive when the full configurability features must be provided by the architecture supporting it. The reconfigurable matrix presented in section 7.2 exploits the reconfigurability capabilities already present in the FPGA at a very low level, providing in this way maximum connection flexibility with minimum hardware resources.

5.3 Conclusions

This chapter introduced a methodology based on two concepts for tackling the specification of bio-inspired hardware systems: computation engine and adaptation mechanism. The separation of these two concepts, which in software implementations are sometimes not very well differentiated, allows one to tackle the bio-inspired hardware design with a hardware-oriented approach.

It is important in adaptive hardware implementations to differentiate between the actual hardware constituting the solution to the problem at hand, and the underlying mechanisms allowing it to adapt. It is usually the computation unit which has more timing constraints, while the adaptation mechanism is often less critical in terms of execution speed.

The next two chapters will show different examples of computation engines where parametric and structural adaptations are used. Chapter 6 presents three computation engines, each with its respective parametric adaptation. They are: synaptic-weight adaptation for spiking neurons, fuzzy rules adaptation for fuzzy systems, and rules adaptation for non-uniform cellular automata. Chapter 7 shows two examples of structural adaptation: the coarse-grained and the fine-grained approaches for topology exploration.

Chapter 6

Parametric Adaptation

> Men are born ignorant, not stupid; they are made stupid by education.
>
> Bertrand Russell

Learning in living beings involves several physiological processes that result in a set of synaptic efficacy modifications, constituting one of the key components of neurobiological research. The underlying mechanisms allowing these efficacies to be learned in order to achieve a specific goal, for example to remember a face or to coordinate movements for playing drums, remain an unsolved issue being widely studied by neuroscientists. This synaptic efficacy adaptation exhibited in the brain is an example of parametric adaptation in living beings.

In a similar manner, parametric adaptation in bio-inspired hardware is related to minor system modifications. It basically consists in parameter tuning of the computation engine by modifying, for instance, a register value or the truth table of a combinatorial function, while keeping the system topology unchanged. The most typical example is that of ANNs: the computation engine -the neural network- implements a function solving the problem at hand; however, it is the adaptation mechanism -the learning algorithm- which modifies the synaptic weights in order to allow the ANN to implement the desired function.

When bringing these two components to reconfigurable hardware, one must provide the system implementation with a certain plasticity, in order to allow the system to adapt. This plasticity can be provided in the form of a memory or a set of registers storing the synaptic weights to be adapted, which constitutes a classical approach for providing flexibility or programmability in such a type of hardware systems. Another way of providing this plasticity is by allowing the system to be partially reconfigured only in the sections involving the utilization of these parameters.

In this chapter, there are presented three computation engines with their respective parametric adaptations, including both types of plasticity implementations described in the previous paragraph. In section 6.1, a hardware-oriented spiking neural network is presented with its synaptic weight adaptation performed by using two techniques: a GA, and hebbian learning.

In this case, the synaptic weight update is done in a classical way: they are stored in distributed memories, and the learning algorithm updates them by rewriting the memory values. In section 6.2, the computation engine consists in a fuzzy classifier, and the adaptation mechanism, driven by a Fuzzy CoCo algorithm [150], reconfigures the fuzzification membership functions and the fuzzy rules. Finally, section 6.3 presents reconfigurable cellular automata, where automata rules are reconfigured, driven by a cellular programming algorithm [186].

6.1 A hardware-oriented spiking neuron model

Most neuron models, such as perceptrons or radial basis functions, use continuous values as inputs and outputs, processed using logistic, gaussian or other continuous functions [60, 152]. In contrast, biological neurons process spikes: as a neuron receives input spikes by its dendrites, its membrane potential increases following a post-synaptic response. When the membrane potential reaches a certain threshold value, the neuron fires, generating an output pulse through the axon. The best known biological model is the Hodgkin and Huxley model (H&H) [71], which is based on ion current activities through the neuron membrane.

However, the most biologically plausible models are not the best suited for computational implementations. This is the reason why other simplified approaches are needed [113]. The leaky integrate and fire (LI&F) model [40,114] is based on a current integrator, modelled as the potential of a circuit composed of a resistance and a capacitor in parallel. The spike response model (SRM) [40] expresses the membrane potential in terms of kernel functions instead of differential equations as in LI&F.

Spiking-neuron models process discrete values representing the presence or absence of spikes; this fact allows a simple connectionism structure at the network level and a striking simplicity at the neuron level. However, implementing models like SRM0 and LI&F on digital hardware is highly inefficient, wasting many hardware resources and exhibiting a large latency due to the implementation of kernels and numeric integrations. This is why a functional hardware-oriented model is necessary to achieve fast architectures at a reasonable chip area cost.

In this section, a hardware-oriented spiking neuron model is presented along with a characterization of the computing capabilities of a network. This work has been published in [219]. Additionally, this section presents a hardware-oriented hebbian learning model, which has been published in [220]

6.1.1 Hardware-oriented spiking neuron model

The simplified integrate-and-fire model presented in this section, as other standard spiking models, uses the following five concepts: (1) membrane potential, (2) resting potential, (3) threshold potential, (4) postsynaptic response, and (5) after-spike response (see figure 6.1). A spike is represented by a pulse. The model is implemented as a Moore finite state machine. Two states, operational and refractory, are allowed.

During the operational state, the membrane potential is increased (or decreased) each time a pulse is received by an excitatory (or inhibitory) synapse, then decreases (or increases) with a constant slope until the arrival to the resting value. If a pulse arrives when a previous

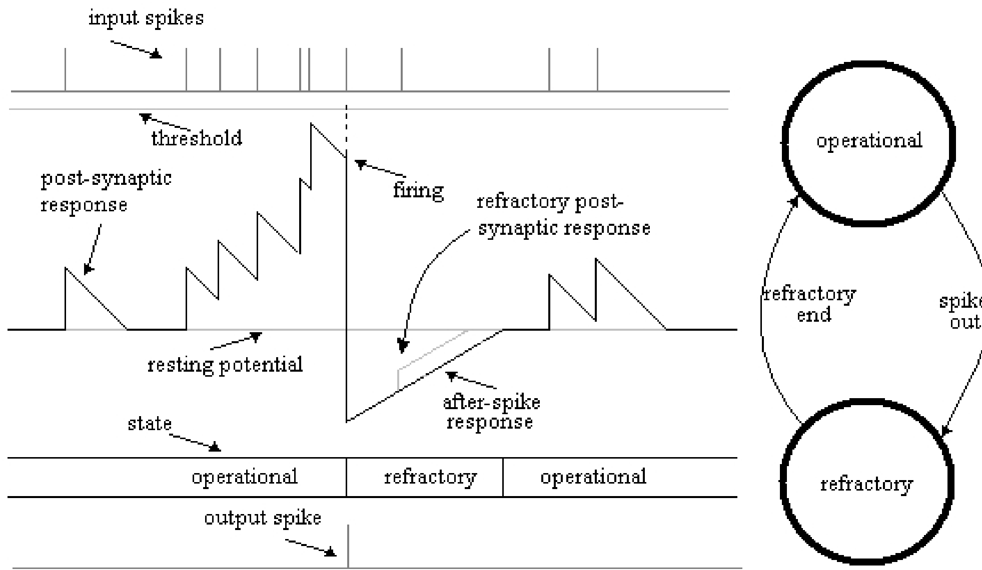


Figure 6.1 *Response of the model to a train of input spikes, and the Moore state machine that describes such response.*

postsynaptic potential is still active, its action is added to the previous one. Membrane potential dynamics are described by:

$$
\begin{aligned}
u(t) = u(t-1) - \quad & K(u(t-1)) + \textstyle\sum_{i=1}^{n} w_i s_i(t-1) \\
\text{with} \quad K(u(t)) = & \begin{cases} k_1 & \text{if } u(t) > U_{rest} \\ -k_2 & \text{otherwise} \end{cases}
\end{aligned}
\tag{6.1}
$$

Where $u(t)$ is the membrane potential at time t, U_{rest} is the constant resting potential, n is the number of inputs to the neuron, w_i is the synaptic weight for input i, $s_i(t)$ is the input spike at input i and at time t, and k_1 and k_2 are positive constants that determine the decreasing and increasing slopes respectively.

When the firing condition is fulfilled (i.e., potential = threshold), the neuron fires, and the potential takes on a hyperpolarization value called after-spike potential. The neuron then passes to the refractory state.

After firing, the neuron enters a refractory period during which it recovers from the after-spike potential to the resting potential. Two kinds of refractoriness are allowed: absolute and partial. Under absolute refractoriness, input spikes are ignored, and the membrane potential is given by:

$$u(t) = u(t-1) + k_2 \tag{6.2}$$

Under partial refractoriness, the effect of input spikes are attenuated by a constant factor. The membrane potential in this case would be expressed as:

$$u(t) = u(t-1) + k_2 + \frac{\sum_{i=1}^{n} w_i s_i(t-1)}{a} \tag{6.3}$$

Where a is a constant positive integer, and determines the attenuation factor. The refractory state determines the time needed by a neuron to recover from firing. This time is completed when the membrane potential reaches the resting potential, and the neuron comes back to the operational state.

The proposed model simplifies some features with respect to SRM0 and LI&F, in particular the post-synaptic response. The way in which several input spikes are processed affects the system dynamics: under the presence of 2 simultaneous input spikes, SRM0 performs a linear superposition of post-synaptic responses, while the proposed model, in a way similar to LI&F, adds the synaptic weights to the membrane potential. Even though the proposed model is less biologically plausible than SRM0 and LI&F, it is still functionally similar.

Other simplistic types of post-synaptic responses could be also considered such as the ones presented by Maass [113]. A type-A neuron uses a post-synaptic response as the one of figure 6.2.(a), which could provide lower computing capabilities and lower resources requirements for the FPGA implementation. In the same way, the post-synaptic response for a type-B neuron (Figure 6.2.(b)) could improve computation, while requiring more logic resources.

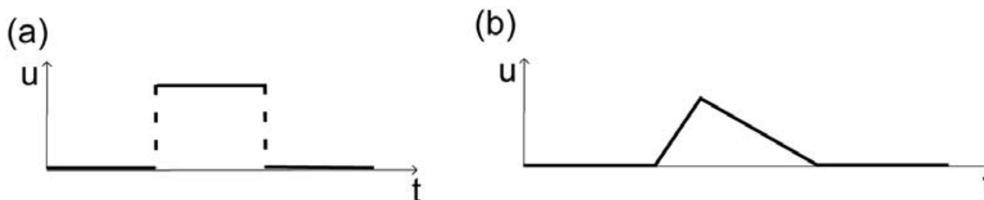


Figure 6.2 *Post-synaptic potential responses for neurons (a) type-A and (b) type-B.*

6.1.2 Computing capabilities

Three experiments have been performed in order to measure the flexibility and the representational power of the model. The goal was to test the functionality and capabilities of a network of neurons, initially with a static problem (an XOR gate), then with a simple dynamic problem (a temporal 3-pattern recognizer), and then with a more complex dynamic problem (a temporal 10-pattern number recognizer). A generic network topology is proposed for all the problems (see figure 6.3). It consists in a network with 3 layers: input, hidden, and output, with recurrent connections allowed only in the hidden layer. To provide the input to the network, the logical values '1' and '0' of the patterns are represented by a train of three spikes and by the absence of spikes respectively. Due to the propagation of spikes throughout the network, the classification spikes arrive a certain time after the presentation of the full pattern.

The patterns for the temporal pattern recognition problem are: $+$, $\times$, and $\diamond$, drawn on a grid of 5 x 5 pixels (Figure 6.4). A pattern is presented to the network one row after another. The topology of the network consists of an input layer of 5 neurons (one for each column), a recurrent hidden layer with 10 neurons, and an output layer with 3 neurons, one for each pattern (see Figure 6.3).

For the number-recognition problem, ten numbers are represented with a grid of 4 columns and 5 rows (see Figure 6.5). The network has 8 input neurons, 30 hidden neurons, and 10 output neurons (the negation of the pattern is also used as input, as it increases the

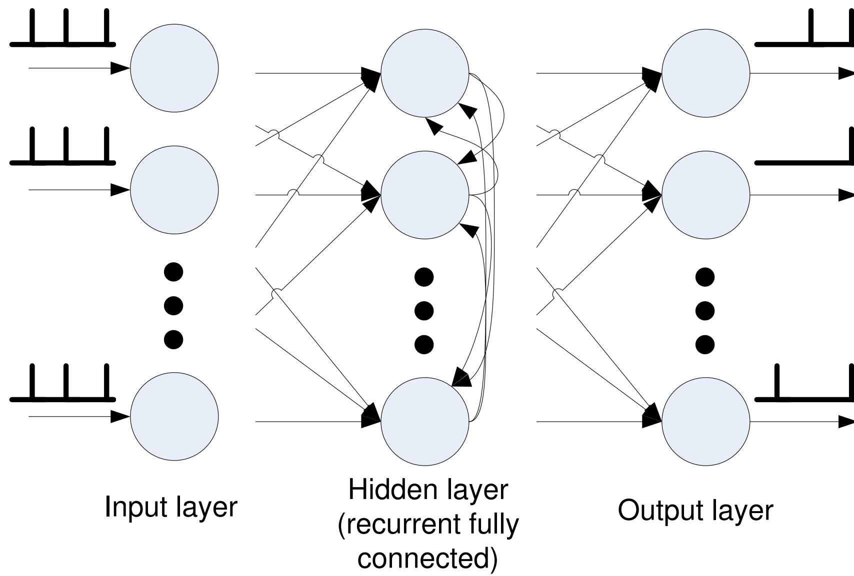


Figure 6.3 *Topology of the network for pattern recognition.*

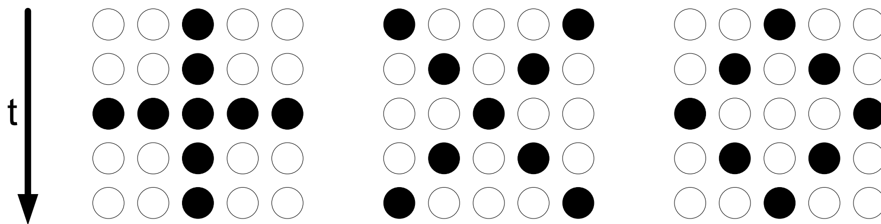


Figure 6.4 *Training patterns. Patterns are presented as the time flows. Each row is presented at a sample period of n iterations.*

amount of useful information). In order to reduce cases where several output neurons fire with the same input, a competitive strategy was used for inhibiting reading outputs after a first output fires (i.e., the pattern is considered as classified).

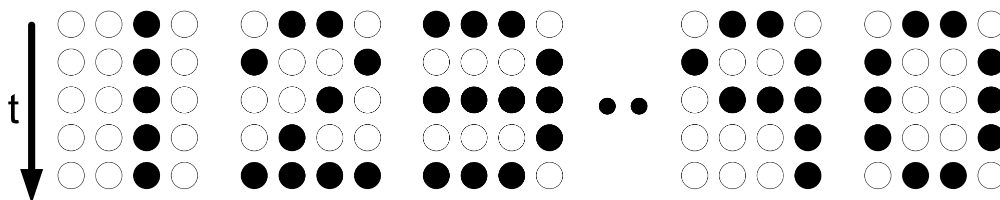


Figure 6.5 *Number patterns on a grid 4 x 5.*

6.1.2.1 Genetic weight determination

A simple GA [227] was used for evolving the network synaptic weights. For each problem the basic genome encodes up to three groups of weights: input-to-hidden, hidden-to-hidden and

hidden-to-output. The number of neurons in the involved layers thus defines the number of weights.

For the pattern-recognition problem the genome encodes 180 synapse weights taking values from -255 to 256 using 7-bit resolution for a total genome length of 1260 bits. Crossover probability: 0.9, mutation probability: 0.001, 200 individuals and 500 generations. The fitness evaluates the classification error and penalizes high weight values. The classification error is given by both undesired firing neurons and not-firing neurons expected to fire. The fitness function function is given by:

$$fitness = 9 - classification\ error - average\ normalized\ weights \tag{6.4}$$

For the number-recognition problem the genome encodes, beside the weights, some neuron parameters: the resting potential, the threshold potential and the postsynaptic slope. The weights take on values from -127 to 128 with 6-bit resolution; the potential values go from 0 to 255 with 8-bit resolution, and the slope from 1 to 9 with 3-bit resolution. The total length of the genome is thus 8659 bits.

For this latter problem, a two-stage incremental evolution was used to search for the solution. The first stage uses an ”easy” fitness criterion to perform coarse tuning; its goal is to find several individuals with acceptable performance. The second stage uses a ”harder” fitness criterion to finely tune the parameters of a set of the best individuals previously found which are used as initial population. The ”easy” fitness computes two scores: the number of patterns correctly classified without considering if the classification was ambiguous or not (see table 6.1) and a second score obtained by comparing the non-fired outputs in both the desired and the obtained classification vectors. It is called ”easy” because the fitness difference between good individuals is small, guaranteeing a smoother landscape. The fitness function is given by:

$$\begin{aligned} fitness = \quad & sum(correctly\ classified\ patterns)/10 \\ & +sum(correctly\ not\ classified\ patterns)/90 \end{aligned} \tag{6.5}$$

The scale factors $1/10$ and $1/90$ normalize the expression, leading to a maximum fitness of 2.

The ”harder” fitness criterion has a more rugged landscape, due to a larger difference between good individuals; it penalizes the ambiguous classifications more strongly, while the previous strategy was very indulgent with them. The fitness evaluates each pattern and assigns a score according to the percentage of good classifications (1 point), ambiguous classifications (0.5 points), undetected classification (0.5 points), and misclassifications (0 points), and penalizes the number of output spikes generated, in order to reduce the classification of the same pattern on several classes. The fitness is given by:

$$\begin{aligned} fitness = \quad & (accurate\ classifications \times 1) + (ambiguous\ classifications \times 0.5) \\ & +(undetected\ patterns \times 0.5) - (total\ fired\ outputs/50) \end{aligned} \tag{6.6}$$

Table 6.1 *Criteria used to describe the quality of classifications.*

Criterion	Description	Output example
Accurate classification	The desired output vector is obtained (i.e., the classification error is 0)	1 0 0 (desired) 1 0 0 (obtained)
Ambiguous classification	The desired output is activated, but another one fires too.	1 0 0 (desired) 1 0 1 (obtained)
Undetected pattern	There is no classification.	1 0 0 (desired) 0 0 0 (obtained)
Misclassification	The pattern is classified in at least one wrong class, but not in the desired one.	1 0 0 (desired) 0 1 0 (obtained)

The maximum fitness, which would be obtained with 10 accurate classifications and 10 fired outputs, is 9.8.

Each evolutionary stage was carried out using different parameters. For the first one, the crossover probability was set to 0.9, the mutation probability to 0.0001, an the elitism to 2, 200 individuals were used, and 500 generations were run. For the second one, the crossover probability was set to 0.1, the mutation probability to 0.001, the elitism to 1, 10 individuals were used, and 500 generations were run. The second one uses atypical values for the probabilities of the genetic operators and for the size of the population since it is intended to perform mutation-driven tuning for some good individuals.

6.1.2.2 Results

Several evolutionary runs were carried out for each GA. For the pattern recognition problem the GA always finds a solution that correctly classifies the three patterns. The best individual has a fitness of 8.54, meaning that the classification error is 0 and the mean of $|weights|$ is 117, the minimum value is -255, and the maximum is 248, which leads to suspect that the weight optimization is not performing very well. The evolution easily loses the best individual, and so because of that, elitism was introduced in the number-recognition problem. For the sensitivity test the results showed, for 10 runs on 300 noisy patterns: an average accurate classification of 170.3, 80.2 undetected patterns, 36.7 ambiguous classifications, and 12.8 misclassifications (It has to be considered that the sensitivity test is done on an already trained pattern without any generalization method, and the training does not take into account any test or validation set).

For the number-recognition problem the first stage finds an individual that accurately classifies 3 patterns, the remaining 7 patterns are ambiguously classified. The achieved fitness is 1.844, which means that all patterns are correctly classified due to all expected neurons having fired, but there were also 14 unexpected output firings (over 100 possible firings).

At the second stage, the best individual found using the first criterion exhibits a fitness of 6.02 (3 *accurately classified* $\times$ 1 + 7 *ambiguous classification* $\times$ 0.5 . 24 *output spikes* / 50). After the fine tuning at the second stage the obtained results displayed an accurate classification for 6 patterns (1, 3, 4, 6, 7, 0) and undetected patterns for the remaining 4, leading to a fitness of 7.88 (6 *accurately classified* $\times$ 1 + 4 *undetected pattern* $\times$ 0.5 - 6 (*output spikes* / 50).

This indicates that the network may not have the complexity required to completely solve the problem.

6.1.3 Learning

Weight learning is an issue that has not been fully solved on spiking neuron models. Several learning rules have been explored by researchers, Synaptic Time Dependant Plasticity (STDP), introduced in subsection 3.4.2, being one of the most studied [40, 114]. In general, hebbian learning modifies the synaptic weight w_{ij}, considering the simultaneity of the firing times of the pre- and post-synaptic neurons i and j. In this way, when two neurons fire within the same time-window, the synapse connecting them is strengthened. Herein it will be described a simplified implementation of hebbian learning oriented toward digital hardware. Two functions are added to the neuron model: *active-window* and *learning*.

In the proposed model, the active-window function determines whether the learning window of a given neuron is active or not (Figure 6.6), maintaining a value 1 during a certain time period after the generation of a spike by a neuron n_i. In this way, the active-window function is given by:

$$W_i(t) = step(t_i^f) - step(t_i^f + s) \tag{6.7}$$

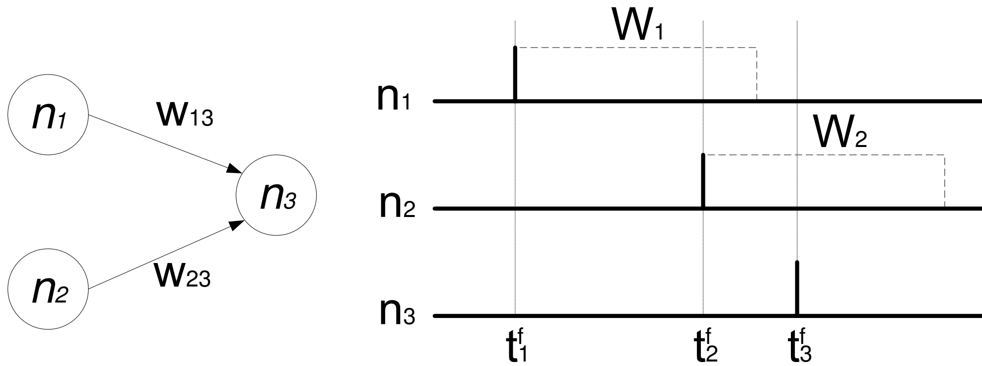


Figure 6.6 *Hebbian learning window. When neuron n_3 fires at t_3^f, the learning window of neurons n_1 and n_2 are disabled and enabled respectively. At time t_3^f Synaptic weight w_{13} is decreased by the learning algorithm, while w_{23} is increased.*

Where t_i^f is the firing time of n_i and s is the size of the learning window. This window allows the receptor neuron (n_j) to determine the synaptic weight modification (Δw_{ij}) that must be done.

The learning function modifies the synaptic weights of the neuron, performing the hebbian learning (Figure 6.6). Given a neuron n_i with k inputs, when a firing is performed by n_i the learning rule modifies the synaptic weights w_{ij} (with $j = 1, 2, ...k$) as follows:

$$\begin{aligned} w_{ij}(t) &= w_{ij}(t-1) + \Delta w_{ij}(t) \\ \text{with } \Delta w_{ij}(t) &= \alpha W_j(t) - \beta \end{aligned} \tag{6.8}$$

Where α is the learning rate and β is the decay rate. Both α and β are positive constants such that $\alpha \gg \beta$.

These two functions, active-window and learning, increase the amount of interneuron connectivity as they imply, respectively, one extra output and k extra inputs for a neuron implementation(Figure 6.7.(a)).

6.1.4 The proposed neuron model on hardware

Several hardware implementations of spiking neurons have been supported by analog and digital circuits [114, 163, 165, 211]. Analog electronic neurons achieve postsynaptic responses very similar to their biological counterparts; however, analog circuits are difficult to set up and debug. On the other hand, digital spiking neurons are often less biologically plausible given their discrete nature, but are easier to set up, debug, scale, and learn, among other features. Additionally these models can be rapidly prototyped and tested thanks to configurable logic devices such as FPGAs.

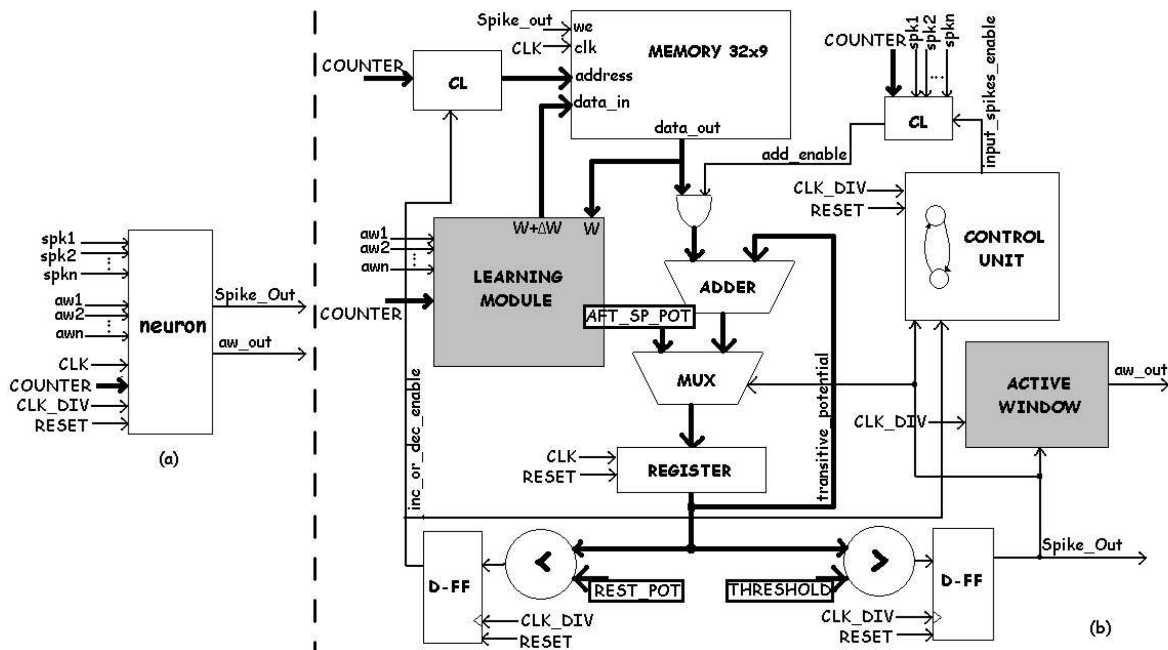


Figure 6.7 *Proposed hardware neuron (a) External view. (b) Neuron architecture.*

The hardware implementation of the neuron model is illustrated in Figure 6.7. The neuron is basically composed of: (1) a control unit, (2) a memory containing parameters, (3) some logic resources to compute the membrane potential, (4) two modules performing the learning, and (5) logic resources to interface input and output spikes. The control unit is a finite state machine with two states: operational and refractory (figure 6.1). An absolute refractoriness is implemented in the proposed neuron. The computing of a time slice (iteration) is given by a pulse at the input *clk_div*, and takes a certain number of clock cycles depending on the number of inputs to the neuron. The synaptic weights are stored in a memory, which is swept by a counter. Under the presence of an input spike, its respective weight is enabled for addition to

the membrane potential. Likewise, the decreasing and increasing slopes (for the post-synaptic and after-spike responses respectively) are contained in the memory.

Although the number of inputs to the neuron is parameterizable, increasing the number of inputs implies raising both the area cost and the latency of the system. Indeed, the area cost depends to a high degree on the memory size, which itself depends on the number of inputs to the neuron (e.g. the 32x9-neuron on Figure 6.1 has a memory size of 32x9 bits, where the 32 positions correspond to 30 input weights and the increasing and decreasing slopes; 9 bits is the arbitrarily chosen data-bus size). The time required for computing a time slice is equivalent to the number of inputs plus one, i.e. 30 inputs plus either the increasing or the decreasing slope.

The dark blocks on Figure 6.7, active-window and learning module, perform the learning in the neuron. The active-window block consists in a counter triggered when an output spike is generated, and stopped when a certain value defining the learning window is reached. The output *aw_out* (active-window out) reads logic-1 if the counter is active and logic-0 otherwise.

The learning module performs the synaptic weight learning described in the previous subsection. This module computes the change to be applied to the weights (ΔW), keeping them bounded. At each clock cycle the module computes the new weight for the synapse pointed at by the *COUNTER* signal; however, these new weights are stored only if an output spike is generated by the current neuron.

6.1.5 Experimental setup and results

The experimental setup consists of two parts: a Matlab simulation for a spiking neural network, and its respective validation on an FPGA.

6.1.5.1 Network description and simulation

A frequency discriminator is implemented in order to test the capability of the learning network to unsupervisedly solve a problem with dynamic characteristics.

Using the 30-input neuron described in subsection 6.1.4, a layered neural network with 3 layers is implemented, fulfilling the constraints required for the coarse-grained on-line evolution implementation described in subsection 5.2.2.1. Each layer contains 10 neurons and is internally fully-connected. Additionally, layers provide outputs to the preceding and the following layers, and receive outputs from them, as described in figure 7.1. For the sake of modularity, each neuron has 30 inputs: 10 from its own layer, 10 from the preceding one, and 10 from the next one.

To present the patterns to the network, the encoding module takes into account the following considerations: (1) 9 inputs are used at layer 1 to introduce the pattern, (2) the patterns consist in two sinusoidal waveforms with different periods, (3) the waveforms are normalized and discretized to 9 levels, (4) every 3 time slices (iterations) a spike is generated at the input corresponding to the value of the discretized signal (Figure 6.9).

The simulation setup takes into account the constraints imposed by the hardware implementation. Table 6.2 presents the parameter setup for the neuron model and for the learning modules. Initial weights are integer numbers generated randomly from 0 to 127.

Different combinations of two signals are presented as shown in figure 6.9. In order to help the unsupervised learning (described in subsection 6.1.3) to separate the signals, they are

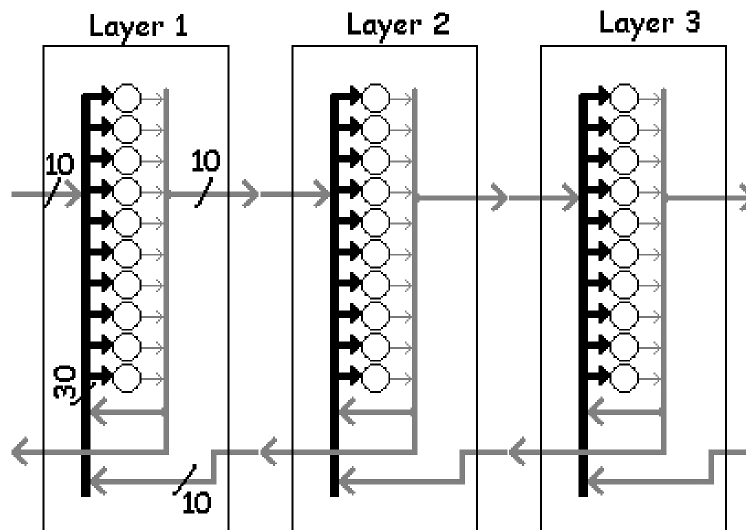


Figure 6.8 *Layout of the network implemented on hardware.*

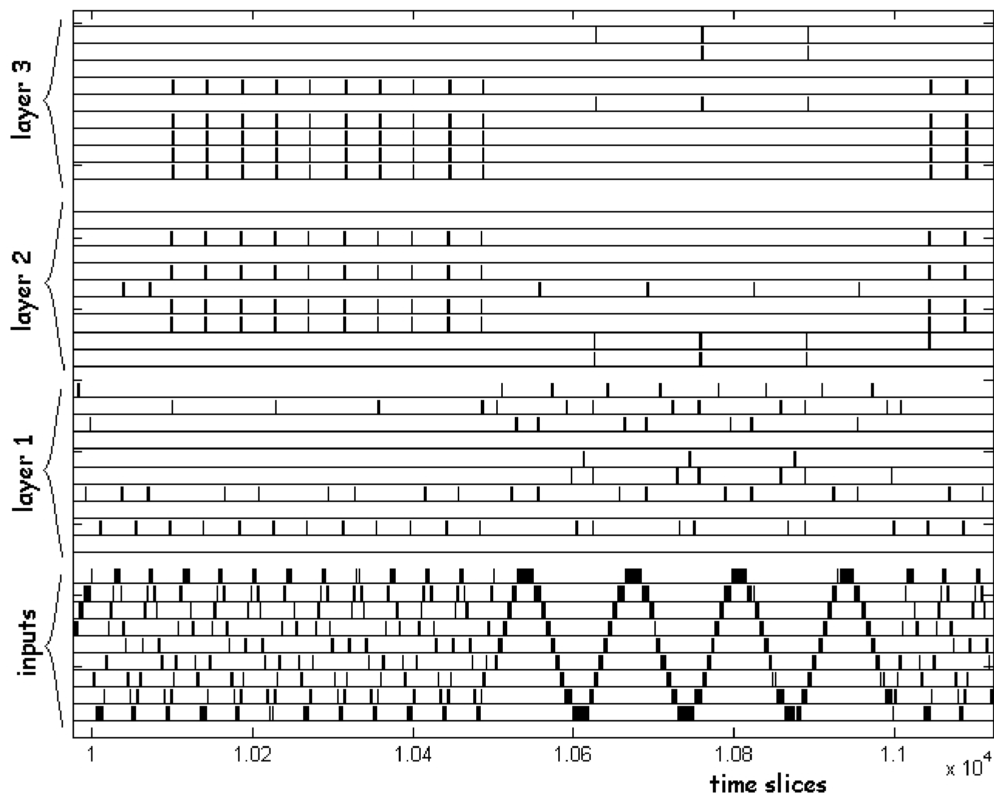


Figure 6.9 *Neural activity on a learned frequency discriminator. The lowest nine lines are the input spikes to the network: two waveforms with periods of 43 and 133 time slices are presented. The next 10 lines show the neuron activity at layer 1, and the following lines show the activity of layers 2 and 3. After 10.000 time slices a clear separation can be observed at the output layer where some neurons fire under the presence of only one of the waveforms.*

Table 6.2 *Set-up parameters for each neuron and for the hebbian learning.*

Neuron Parameters		Learning Parameters	
Resting potential	32	Learning Rate	6
Threshold potential	128	Decay rate	4
After-spike potential	18	Weight upper bound	127
Increasing slope	1	Weight lower bound	-32
Decreasing slope	1	Learning window size	16
Potential lower bound	-128		

presented as follows: during the first 6000 time slices the signal is swapped every 500 time slices, leaving between them an interval of 100 time slices, where no input spike is presented. Then, this interval between the signals is removed, and the signal frequency is then switched every 500 time slices.

Several combinations of signals with different periods are presented to the network. Five tries are allowed for each combination. Some of the signals are correctly separated at least once, while others are not, as shown in Table 6.3. It must be noted that the ranges of periods that are separable is highly dependent upon the way in which data are presented to the network (encoding module). In this case, a spike is generated every 3 time slices; however, if higher (or lower) frequencies are expected to be processed, spikes must be generated at higher (or lower) rates. The period range is also affected by the dynamic characteristic of the neuron: i.e., the after-spike potential and the increasing and decreasing slopes. They determine the membrane-potential response after input and output spikes, playing a fundamental role in the dynamic response of the full network.

Table 6.3 *Signal periods presented to the network. Period units are time slices. (The symbol ”$\surd$” indicates a successful discrimination, and the symbol ”X” indicates an unsuccessful discrimination).*

Period 1	Period 2	Discrimination
40	100	$\surd$
43	133	$\surd$
47	73	X
47	91	$\surd$
50	100	$\surd$
73	150	X
73	190	$\surd$
101	133	$\surd$
115	190	X
133	170	X
133	190	X

6.1.5.2 The network on hardware

The same neural network described above is implemented on a relatively small FPGA to validate the network execution. The network was implemented in a Spartan-II XC2S200 FPGA from Xilinx with a maximum capacity of 200.000 logic gates. This FPGA has a matrix of 28 x 42 CLBs (configurable logic blocks), each of them composed of 2 slices, which contain the logic where the functions are implemented, for a total of 2352 slices. The XC2S200 is the largest device from the low-cost FPGA family Spartan-II. Other FPGA families such as Virtex-II offer up to 40 times more logic resources.

The 30-input neuron described in subsection 6.1.4 was implemented with a data bus of 11 bits, however the memory maintains its width of 9 bits. The width of the data bus is larger than the memory width, preventing transitory overflow for arithmetic operations. Synthesis results about different implementations of these neurons can be found in [220]. The area requirement is very small compared to other more biologically-plausible implementations (e.g. Ros et al [165] use 7331 Virtex-E CLB slices for 2 neurons). However, making a quantitative comparison in terms of performance with this or with other implementations would be impossible, given the absence of a standard test-bench for measuring performance. Several criteria might be taken into account in addition to the minimum error achieved in different possible problems, such as: execution speed, learning speed, size of the neuron, generalization ability, possibility of learning on-chip or off-chip, possibility of learning on-line or off-line, biological plausibility, etc.

Table 6.4 presents the synthesis results for a neuron, a layer, and the whole network with and without modular design. Note that a layer of 10 neurons take fewer slices than 10 independent neurons thanks to synthesis optimization. That should also apply to the whole network. However, when the network is modular it is not possible to simplify the implementation, given that each layer has clearly-defined boundaries in the circuit, and cannot be merged with neighbour modules.

Table 6.4 . *Synthesis results for a neuron, a layer, and a network.*

Unit synthesized	Number of CLB slices	FPGA percentage
A neuron (30 inputs)	53	2.25 %
A layer (10 neurons)	500	21.26 %
A network (3 layers, without modules)	1273	54.21 %
A network (3 layers, modular design)	1500	63.78 %

To test the design, the sequence of input spikes (i.e., after the encoding stage) is stored in a memory block. The hardware network, both in simulation and on-chip, exhibits similar behaviour to that of its Matlab counterpart: clear frequency discrimination is obtained at the output of the network, as some outputs generate spikes responding only to a given input frequency.

The system achieves a speed of up to 54.4MHz. The latency for a time slice is 64 clock cycles, what means that the duration of a time slice can go down to 1.17 μs. The neuron was implemented with a latency of 64 clock cycles (the equivalent to one time slice) to allow it to interact with larger neurons with up to 62 inputs, guaranteeing uniformity in the spike duration.

However, given that for this specific network, only 30-input neurons are used, the latency can be reduced to 32 clock cycles. This latency reduction can also slightly increase the operating frequency of the system, since it implies some reduction of the logic resources.

6.2 Coevolutionary setup for adapting fuzzy systems

Nature has long inspired scientists from many disciplines, but it is only very recently that technology is allowing the physical implementation of bio-inspired systems. Currently a non-negligible part of computer science is devoted to building and developing new bio-inspired systems and most of them yield quite good performance, but often even their creators do not know why and how such systems work since they perform opaque heuristics. Fuzzy systems are an exception among these approaches since they might provide both good results and interpretability of them. Nevertheless, the construction of fuzzy systems is a hard task involving a lot of correlated parameters, which are often subject to several constraints to satisfy linguistic criteria. EAs fit well to such a task [178]. *Fuzzy CoCo* is an evolutionary technique, based on cooperative coevolution, conceived to produce accurate and interpretable fuzzy systems [150].

Three platforms are typically used when implementing fuzzy systems: microprocessors (or software), dedicated ASICs, and FPGAs. Maximum flexibility can be reached with a software specification of the full system; however, fuzzy systems are highly parallel and microprocessor-based solutions perform poorly when compared to their hardware counterparts. A dedicated ASIC is the best solution for achieving performance, but such an approach reduces dramatically the adaptability of the system [27]. Finally, FPGA-based systems provide both: higher performance for parallel computation than software solutions, and enhanced flexibility compared to ASICs thanks to their dynamic partial reconfiguration (DPR) feature [245]. Thus, they constitute the best candidate for supporting high-performance evolvable fuzzy systems. Moreover, their run-time reconfiguration features can be used to reduce execution time by hardwiring computationally intensive parts of the algorithm [89].

In this section, a hardware platform for coevolving fuzzy systems by using Fuzzy CoCo is proposed in order to speed up both evolution and execution while offering equivalent performance. The methodology of node evolution (subsection 4.3.2.2) is used for setting up the reconfigurability features. This system has already been reported in our paper [127]. Fuzzy CoCo has already been introduced in subsection 3.2.5. Subsection 6.2.1 describes the EHW platform used for the Fuzzy System. Subsection 6.2.2 describes the genome used to encode the system. Finally, subsection 6.2.3 presents the experimental setup and results of the simulated platform.

6.2.1 The evolvable FPGA platform

The proposed platform consists of three parts: a hardware substrate, a computation engine, and an adaptation mechanism, as described in chapter 5.

The *hardware substrate* supports the computation engine. It must provide good performance for real-time applications and enough flexibility to allow fuzzy system evolution through the adaptation mechanism. The substrate must permit one to test different possible modular layers dynamically. As mentioned before, programmable logic devices such as FP-

GAs appear as the best solution, providing high performance thanks to their hardware specificity, and a high degree of flexibility given their dynamic partial reconfigurability.

The *computation engine* constitutes the problem solver of the platform. Fuzzy systems have been chosen given their ability to provide not only accurate predictions, but interpretability of the results. Other computational techniques are not excluded, such as filters, oscillators, or neural networks.

The *adaptation mechanism* provides the possibility to modify the function described by the computational part. Two types of adaptation are allowed: major structural modification and parameter tuning. The architecture is kept modular in order to allow structural adaptation as described in detail in section 7.1. Herein, the main focus is done on parameter tuning.

6.2.1.1 The fuzzy computation engine

The fuzzy architecture consists of three layers: (1) *Fuzzification* that transforms crisp input values into membership values. (2) The *rule-based inference*, which computes the firing of each fuzzy rule, providing an activation level for one of the four output MFs. As several rules can propose the same action—i.e., the same output MF—the output fuzzy values are aggregated by using an aggregation operator, e.g., *maximum*. Finally, (3) *Defuzzification* produces a crisp output from the resulting aggregated fuzzy set. Inference and defuzzification are merged into a single physical module since the latter is static. Figure 6.10 shows a top level view of the platform.

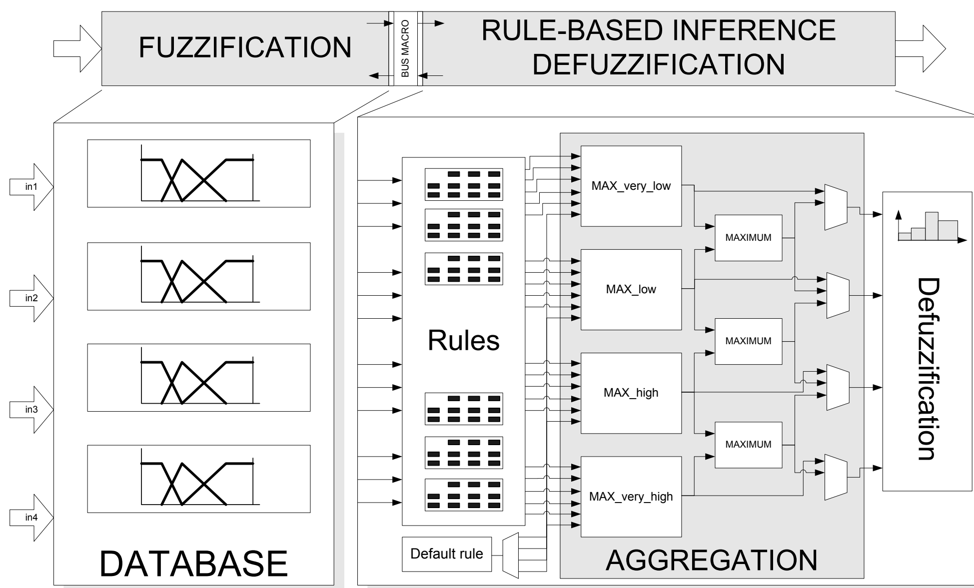


Figure 6.10 *Schematic of the evolvable fuzzy platform.*

6.2.1.2 Reconfigurability

In this architecture, parameter tuning implies modifying lookup table (LUT) functions. The difference-based reconfiguration flow is used, because only small modifications are performed.

This method involves two advantages: (1) minimization of the reconfiguration bitstream size and hence the reconfiguration time, and (2) it allows the posibility of automatically generating the bitstream. To achieve that, three hard macros using LUTs were created for each evolvable part of the platform: the input MF parameters, the inference rules, the aggregation configuration and the output MF parameters. By using hard macro's location constraints, one can locate each LUT and hence modify it by using difference-based reconfiguration as described in [245].

6.2.1.3 Fuzzy model setup

The proposed implementation has 4 input variables with 3 triangular MFs each. The inference layer contains 20 rules that take up to 4 input fuzzy values from different input variables, though, the system is easily scalable for increasing the number of inputs or rules. For the sake of interpretability, a *default rule* was added, whose effect is important when the other rules are not very active. In this implementation, the default rule has a fixed activation level encoded by the genome. One of the most commonly used defuzzification methods is the *center of areas*, which is very costly since it includes division. An iterative method and the use of rectangular output MFs was proposed for implementing this stage. Below there are provided more details on these issues.

- **Fuzzification:** Taking into account semantic criteria, consecutive MFs of a given input variable are orthogonal [150]. The whole variable is, thus, defined by means of three parameters, say p_1, p_2 and p_3, defining the function edges as shown in figure 6.11.(a). Each parameter represents a key point and is taken from the LUTs. To compute the fuzzy membership value, an iterative approach is proposed. The graphic and the pseudocode shown in figure 6.11, describe an example of fuzzification, for the second MF of a variable, of an input value between p_2 and p_3.

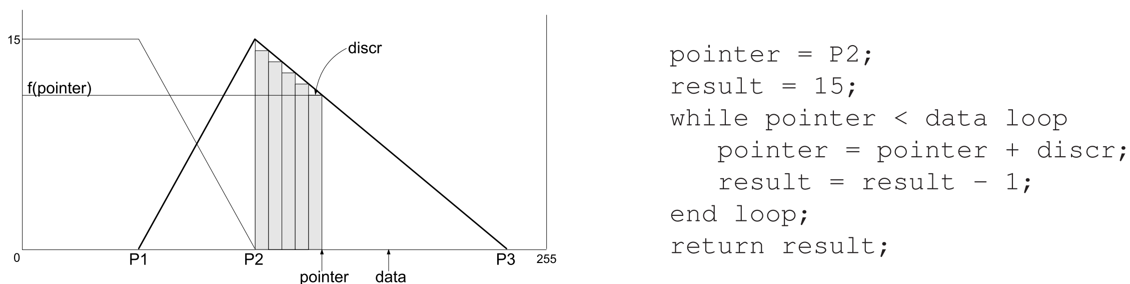


Figure 6.11 *Fuzzification algorithm from subsection 6.2.1.3*

- **Rules:** For maximum flexibility, there are required rules able to include fuzzy *AND* and *OR* operators (i.e., respectively *minimum* and *maximum*). As explained in subsection 6.2.1.2, a hard macro that uses only LUTs has been created to compute any combination of *AND* and *OR* operators on 4 fuzzy values chosen from among 16 input values. Figure 6.12 shows an implementation of the fuzzy *AND* between two 4-bit values a and b.

- **Aggregation:** As mentioned before, the activation level of each MF output corresponds to the aggregation of all the rules proposing such MF as output. As shown in figure 6.10,

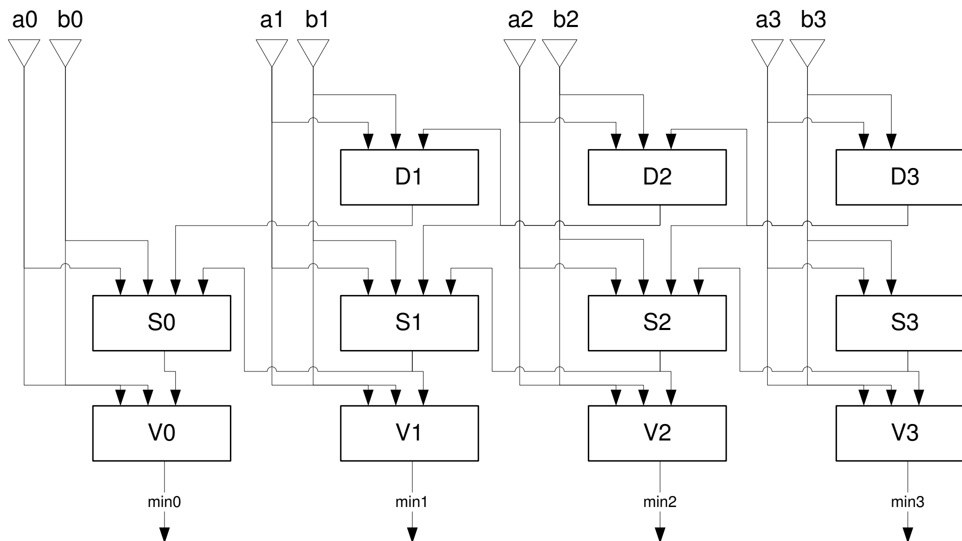
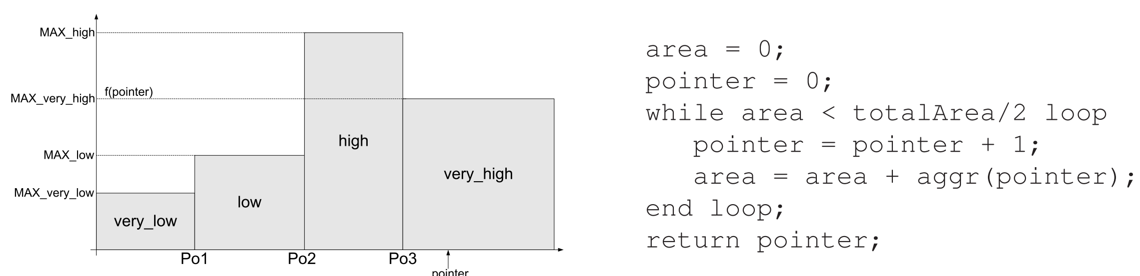


Figure 6.12 *Implementation of a 4-bit minimum operator. Each rectangle represents a LUT taking 4 bits as input. The macro is made up of three layers (D, S and V) and four stages (one per bit). The layer D indicates to the next stage whether a decision can be made or not. Once a decision is made, further D units transmit this fact. The layer S indicates which value, a or b, is chosen by the multiplexer V.*

the maximum number of rules for each output MF is five. However, the merging of two consecutive output MFs is allowed, which has the double effects of increasing the limit on the number of rules per MF and decreasing the number of available output MFs.

- **Defuzzification:** In the proposed architecture, there are considered 4 rectangular output MFs, such as those shown in figure 6.13. This form, intermediate between singletons and triangular MFs, allows the use of an iterative algorithm to approximately compute the center of areas. Although this method increases latency, it reduces logic and can be efficiently pipelined. The defuzzification process is made up of two steps, the first step computes the total area. The second one, illustrated by the pseudocode in figure 6.13, iterates until it reaches half of the total area.




```
area = 0;
pointer = 0;
while area < totalArea/2 loop
   pointer = pointer + 1;
   area = area + aggr(pointer);
end loop;
return pointer;
```

Figure 6.13 *Rectangular defuzzification MFs and defuzzification pseudocode. Note that in the pseudocode, `aggr()` is a function that returns the activation level of the current output linguistic value. Moreover, `totalArea` was previously computed by the first step of the algorithm which differs only in its end criterion.*

6.2.2 Genome encoding

Figure 6.14 illustrates the genome encoding. The i-th input variable is defined by three 8-bit parameters: P_{i1}, P_{i2} and P_{i3} (subsection 6.2.1.3). For simplicity purposes, five rules have been pre-assigned to each output linguistic value. The genome must describe the connections between the input MFs and the rules. Encoding the *k-th* rule requires five parameters: four 2-bit antecedent values, A_{kj}, to choose the applicable MF and one bit, t_k, for the type of operator. The *default rule* is encoded by two parameters: one 4-bit value, dr, for its activation level and one 2-bit value, dr_a for its consequent. The *aggregation* needs four 2-bit parameters, M_l with $l = 1, 2, 3, 4$, that indicate the value to be chosen for each output linguistic value among the original and the merged results (Subsection 6.2.1.3). The *output MFs* are completely encoded by three 8-bit values Po_1, Po_2 and Po_3 that represent their boundaries (See figure 6.13). The genome of the first individual, encoding 4 input variables, is 96 bits long. The genome of the second individual, encoding 20 active rules, the default rule, 4 aggregated MFs and an output variable, is 218 bits long.

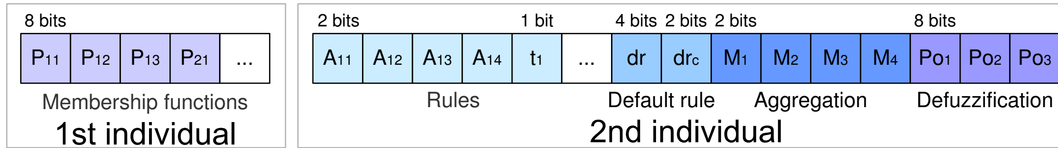


Figure 6.14 *A schematic view of the genome describing the controller.*

6.2.3 Platform setup and results

6.2.3.1 Setup

The experimental setup consists of two parts : (1) a Matlab simulation of the migration of a fuzzy system individual (FSI) from software to hardware implementation, (2) a Matlab simulation of the evolved hardware implementation.

Migration of an FSI from software to hardware implementation: by using Fuzzy CoCo evolution, 50 evolutions were executed using the software fuzzy system in order to generate 50 FSIs using at most 10 rules. Then, we compared performances of both, the hardware and the software fuzzy systems for all these individuals.

Performance of evolved hardware implementation: in this case, the performances are compared on the basis of two different individuals, each of them being specially evolved for a given implementation. 48 evolutions were run with different parameters combinations.

The Iris problem was chosen as benchmark since it has already been used for evaluating Fuzzy CoCo [150]. Fisher's Iris data is a well-known classification problem consisting of feature measurements for the speciation of iris flowers. In order to test the system performance, the Matlab Fuzzy CoCo simulations were reused, but another fuzzy system was implemented. This system takes into account all the constraints imposed by the hardware implementation, as described in subsection 6.2.1.3, except that the default rule activation level is fixed to 2 (i.e. 13%) in the simulation. Therefore, one could consider the real system more flexible than the

Table 6.5 *Comparison between software and hardware implementation performances for the same individual. The best and worst cases are given according to the hardware performance in comparison with the software using the same individual.*

	Software	Hardware	Loss
Mean	97.6	89.5	8.31
Best	96.7	96.7	0.0
Worst	97.3	75.3	22.6
Std dev	0.9	6.3	

Table 6.6 *Comparison between evolved software and hardware implementation performances after 100 generations. The best and worst cases are the overall best and worst performances of both implementations.*

	Software	Hardware	Loss
Mean	97.6	97.4	0.14
Best	99.3	98.7	0.66
Worst	95.4	94	1.4
Std dev	1	0.9	

simulated one. The overall percentage of correctly classified cases over the entire database was considered as the performance metric.

6.2.3.2 Results

Migration of an FSI from software to hardware implementation The overall performance loss in this case is about 8.4%, but it should be observed that the standard deviation of these results is high (6.85%). Some individuals perform the same in both implementations while others make the hardware system lose 20% accuracy, as shown in table 6.5.

Performance of evolved hardware implementation The experiment shows that the hardware can reach almost the same accuracy as the software implementation. Table 6.6 shows the experimental results.

The mean values are almost the same, although the software implementation performs slightly better with its best and worst individuals. One may notice that the evolution allows a great reduction of the hardware standard deviation. Figure 6.15 provides a synthetical view of both implementation's performances.

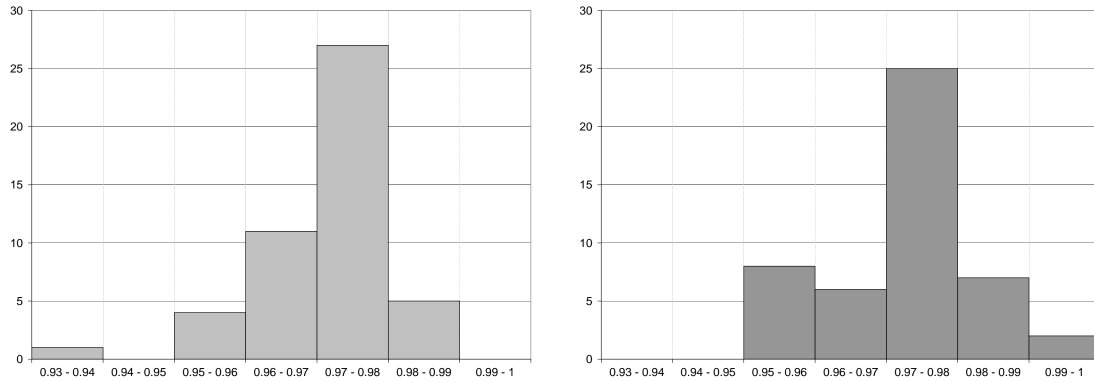


Figure 6.15 *Summary of results of 48 evolutionary runs on both evolved implementations (hardware on the left, software on the right). The histogram depicts the number of systems exhibiting a given mean accuracy value on the complete database.*

6.3 A self-reconfigurable platform for non-uniform cellular automata

Non-uniform cellular automata have already been introduced in section 3.2.4. They are discrete time dynamical systems, consisting of an array of computing cells implementing different boolean functions. Several features make them interesting for hardware implementation: they are massively parallel cellular systems, they are easily scalable, and their basic computing unit is closely related to the basic components of digital hardware -logic gates-.

For FPGAs, non-uniform CA have an even more direct analogy. A typical FPGA architecture uses LUTs for implementing combinatorial functions, a LUT being the ideal substrate for implementing a changing boolean function. Additionally, the ICAP present in Xilix FPGAs along with the bitstream format description reported in subsection 2.3.2 allows one to perform an on-line and on-chip modification of CA rules by dynamically reconfiguring the LUTs' configuration.

The system presented in this section exploits these reconfigurability features offered by Xilinx FPGAs, for evolving non-uniform CA rules. The methodology of bitstream evolution (subsection 4.3.2.2) is used for setting up the reconfigurability features. The work presented in this section has been published in [223].

6.3.1 The evolvable platform

This section presents a platform able to self-reconfiguring non-uniform CA rules through the ICAP. The platform consists in a MicroBlaze soft-processor running on a Virtex-II FPGA from Xilinx. The main advantage of using the vendor-provided soft-processor is the high number of IP peripherals available, and the user-friendly programming environment provided.

6.3.1.1 General system description

The complete system schematic is depicted in figure 6.16. A MicroBlaze soft-processor from Xilinx runs an EA. The program is stored in an internal BRAM, and an external SRAM is used for data storage -i.e. individuals' genomes in this case. The system interfaces with the external world through an UART peripheral , providing a console for monitoring and debugging from a PC. The CA to be evolved can be accessed for reading or for writing the states through general purpose I/O interfaces. However, rule modifications are exclusively performed by the HWICAP peripheral. The HWICAP module allows the MicroBlaze to read and write the FPGA configuration memory through the Internal Configuration Access Port (ICAP) at run time, enabling an evolutive algorithm to modify the circuit structure and functionality during the circuit's operation, specifically, in this case, CA state transitions.

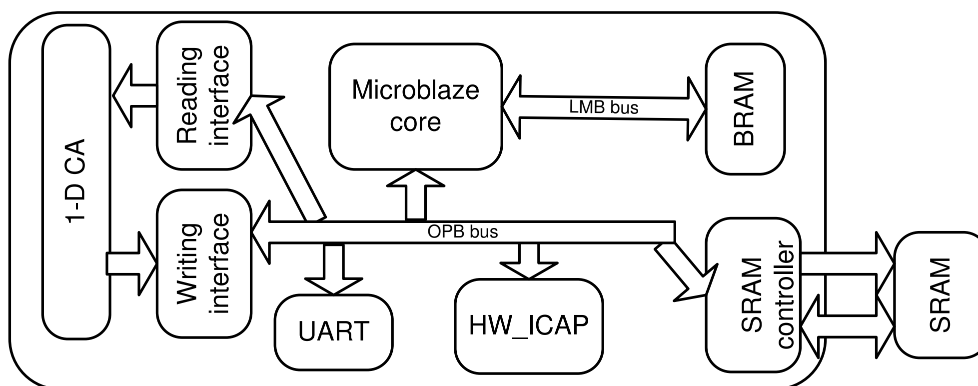


Figure 6.16 *System schematic for self-reconfiguring cellular automata*

6.3.1.2 Cellular automata implementation

This work concentrates on 1-d grids, with a number of states per cell $k = 2$, denoted 0 and 1. In such CA, each cell is connected to T local neighbours (cells) on either side, as well as to itself, where T is a parameter referred to as the radius (each cell has $2T + 1$ neighbours). In this case, the radius $T = 1$, thus the neighborhood equals 3.

A 1-d CA composed of 50 automata is included; it can be configured for running on free-run mode -i.e. a state update at each clock cycle- or on controlled iterative mode -i.e. a state update at each "update" command from the soft-processor. An initial state for the CA can be configured through the writing interface, while the full state can be read by the reading interface.

A special interest has been focused on 1-d CA, with $k = 2$ and $r = 1$, given their analogy with FPGA's basic elements (LUTs and flip-flops). Such an automaton implemented in hardware would require a flip-flop, for storing the current state, and a $3-$input LUT. The most basic logic cell of Virtex-II FPGAs is a slice, which contains 2 flip-flops and 2 4-input LUT, a good fit for implementing two of the above described automaton. Larger automata can be implemented by representing rules with several LUTs.

As explained in subsection 4.3.2, hard macros allow specifying the exact placement of a desired component on a design. In the previous section, hard macros are used for instantiating

fuzzy rules, which are then evolved from a PC. In this case, the hard macro depicted in figure 6.17 has been designed, consisting in a slice containing 2 automata. Then, a 1-d automata array with size 50 was instantiated (i.e. by using 25 hard macros), where the input A4 of LUTs is set with a constant value of ’0’, and A3, A2, and A1 receive the signals from the self, lower, and upper states respectively.

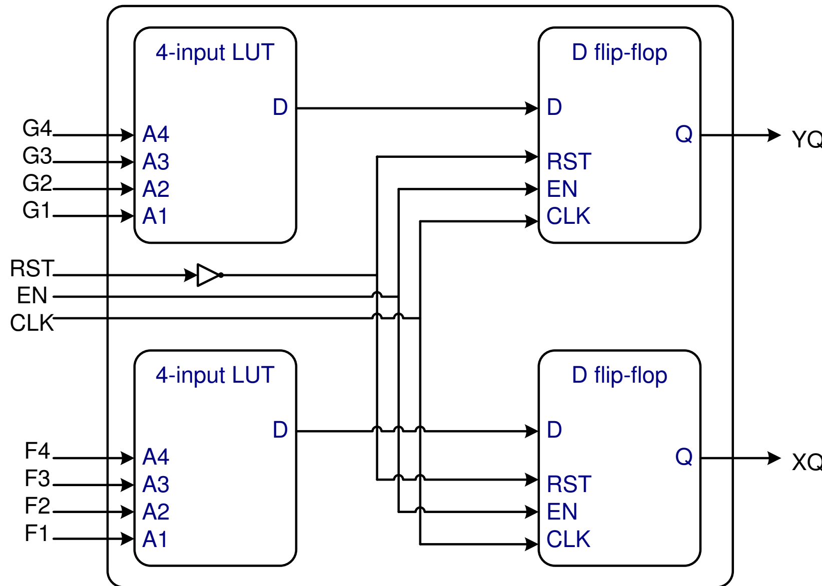


Figure 6.17 *2-automata hard macro*

As described in subsection 2.3.2, one can access the LUT configuration of a whole column of slices in a single configuration frame. That’s the reason why one must place the set of 25 hard macros in a single column: for minimizing the number of frames to access. Then, just by reading and writing a single frame, one can evolve the configuration bitstream containing the LUTs’ functions. By using this implementation in a Virtex-II 1000 FPGA, CA rules for CA of up to 160 automata can be updated just by modifying a single frame. It must be noted that for reconfiguring the full Virtex-II 1000, one may configure it with a full bitstream containing 1104 frames.

6.3.2 Experimental setup and results

Two problems were chosen for validating the proposed platform: firefly synchronization [187] and a random number generator [190]. In both cases the cellular programming algorithm described in subsection 3.2.4 was used, modifying only the fitness evaluation function.

6.3.2.1 Firefly synchronization

In some areas of south-east Asia, one can find certain species of bioluminescent insects called fireflies. These insects emit flashes of light as mating communication signals. When forming a group fireflies flash rhythmically in a synchronized way. This emergent behavior exhibited

by fireflies constitutes a biological spectacle, amazing from the esthetic and scientific point of view [17].

When talking about artificial cellular systems, the firefly synchronization problem consists in synchronizing the firing of a set of 2-state automata. CA are initialized with a random configuration, and after a number of iterations (typically larger than the CA size) each automaton must switch from one state to the other, synchronizing with its neighbors.

In this implementation 50 automata were implemented on 25 slices as described in subsection 6.3.1.2, and they were executing the same cellular programming algorithm described in [187]. For the cellular programming approach, the genome is initialized for every cell in a random way, and through the HWICAP peripheral the genome contents are mapped to the frame containing the LUT contents. It must be considered that, as described in subsection 2.3.2, each slice has 2 LUTs (G-LUT and F-LUT) and the LUT contents are stored in a different order in the configuration bitstream (most significant bit at left for G-LUT and at right for F-LUT), so a simple mapping genome-bitstream must be applied for re-ordering them. Once the frame is re-configured, one can test the CA through the reading and writing interfaces. A random initial state is loaded to the CA, and then it runs for 54 iterations. The fitness is computed by the MicroBlaze soft-processor, by reading the CA state. For computing the fitness, one must let the CA execute four more iterations: if the sequence is 0-1-0-1 the fitness is 1, otherwise the fitness is 0. In that way, the fitness values for 300 initial states are accumulated in order to obtain the total fitness. Then a new genome for each cell is generated as described in subsection 3.2.4. The described system successfully finds genomes able to synchronize the switching of the states, as well as described in [187].

6.3.2.2 Pseudo-random number generator

Good random number generators are not easy to implement; they are mainly consequence of natural physical processes. On the other hand, pseudo-random number generators are commonly used by information systems: starting from a seed value, a non-linear transformation is applied for simulating the previously mentioned real random number generators. Performing measures for determining the quality of a given transformation function is not an easy task; however, a simple and effective way of doing it has been proposed by Koza for evolving random number generators with genetic programming [91]. Koza used the entropy of the generated sequence for computing the fitness.

For the pseudo-random number generator, the same 50-cell CA described in the previous subsection is used, as well as the same cellular programming algorithm (except the fitness function). In [190], Sipper and Tomassini evolved a random number generator on a 1-d 50-cell CA by using cellular programming. Their same algorithm is implemented, with the difference that in this case, a value is not read at each CA update, but the CA runs in free-run mode.

The fitness computation consists in:

- Partially configuring the FPGA with a given CA.
- Random initialization of states and sampling of 4096 consecutive values.
- Compute entropy of the system as the mean entropy for each bit subsequence, with the expression:

$$E_h = \frac{\sum_{i=1}^{n} E_h^i}{n} \tag{6.9}$$

with n being the number of cells, h the subsequence length, and E_h^i is the entropy for the cell i considering a subsequence length h, defined by:

$$E_h^i = -\sum_{j=1}^{k^h} ph_j^i \log_2(ph_j^i) \tag{6.10}$$

where ph_j^i is the probability of obtaining a given subsequence j on the cell i when the subsequence length is h.

- Repeat the steps from the state initialization until 300 experiences have been performed; the total fitness is computed as the average value of the fitness obtained from the 300 experiences.

In the experiments, a sequence length value $h = 4$ is considered, allowing a maximal theoretical value of entropy $E_h = 4$. In [190], they reported a maximum fitness of 3.997; However, they do not specify how many evolutions were performed before finding such solution. The maximum fitness obtained by the platform presented here after running 20 evolutions is 3.963.

6.4 Conclusions

This chapter presents three different approaches for performing parametric adaptation in bio-inspired reconfigurable hardware systems. Each one of the approaches differ in the bio-inspired computation engine used, in the adaptation mechanism, and in the manner that the hardware substrate is modified for allowing the system to adapt.

The first section presented a hardware-oriented spiking neuron model, where two adaptation mechanisms were presented: an off-line GA, and an on-line, on-chip hebbian learning. The computation engine modification is done in the form of update of memories containing the network synaptic weights. This implementation does not modify the configuration bitstream of the underlying FPGA; however, memory updates are done in a parallel, distributed, and completely intrinsic way.

The second section presented a fuzzy system, where the adaptation mechanism was done in the form of a coevolutionary algorithm. In this case, computation engine modifications are done by generating partial bitstreams able to partially reconfigure the FPGA. The main drawback is that each partial bitstream must be generated by the FPGA vendor tools, requiring a PC to do it. In this case, system modifications are done in an extrinsic way.

Finally, the third section presented a non-uniform cellular automata implementation, where the rule adaptation was done by a cellular programming algorithm. In this platform, modifications to the system are done by directly manipulating the configuration bitstream. In this case, unlike the solution of the second section, the bitstream generation does not depend

on the vendor tools, so it can be performed by an on-chip processor, which in addition, can self-reconfigure the FPGA.

These three implementations show a multiplicity of techniques that can be used for including parametric adaptability on available commercial reconfigurable devices. The methodologies proposed in this chapter deal with useful issues when adapting hardware in a general way. Performing on-chip adaptation on reconfigurable platforms has always been an important challenge, and this chapter describes several ways to implement it in an efficient way.

The adaptation presented in the third section constitutes a novel system approach for evolving hardware, and in a larger scope a new dynamic partial reconfiguration design flow. The platform has shown itself to be suitable for evolving non-uniform CA, and the same approach can be easily extended to other cellular structures (like the artificial neurons or the fuzzy rules of the first two sections) just by defining their respective hard macros. The system on chip supporting these reconfigurability capabilities provides the hardware platform required to support the so-called on-chip and on-line self-reconfigurable adaptable systems. It provides the flexibility needed by a real phenotypical modification on the evolved hardware individual.

This chapter discussed parametric adaptation, allowing function tuning, parameter tuning, or rule fitting. However, the search space can be hugely expanded by mixing these techniques with the structural adaptation that will be presented in the next chapter, which include topological adaptation for modifying the neighborhood, the CA size, or even changing the cell's architecture.

Chapter 7

Structural Adaptation

> Judgement comes from experience, and
> experience comes from bad judgement.
>
> Simon Bolivar

Evolution, development, and learning in living beings implies several types of morphological modifications where the organism structure is involved. Learning and development, closely related during human-beings first years of life, are characterized by a high brain plasticity with a very high rate of neuron births and deaths, as well as synaptic development and pruning. Evolution, acting in a larger time scale, also exhibits morphological changes in species from one generation to the next. Even if these changes only become evident when observing a species evolve over thousands of generations (or more), one can also observe that, even in asexual reproduction, an offspring is not an exact copy of his parent and that some gene mutations allow him to differ from his parent. One can thus identify several types of structural adaptation in living beings, which can happen at individual or species level.

In bio-inspired reconfigurable hardware, structural adaptation is related to major system modifications. It basically consists in topology modifications and tuning of the computation engine by modifying, for instance, the number or the type of neurons in a neural network, the size of a cellular automata, or the connectionism in a neural network. In the POE model, structural adaptation is mainly associated with the phylogenetic and ontogenetic axes. The phylogenetic axis of bio-inspired hardware, better known as evolvable hardware, mainly targets the building of logical circuits, implying an architectural construction of the circuit, as presented in chapter 4. The ontogenetic axis requires a circuit to be constructed from a genetic description, given a developmental rule. A hardware platform must consequently provide the required flexibility for supporting individuals with the diversity exhibited by genetic descriptions and developmental rules.

Bringing structural flexibility to hardware is very expensive in terms of logic resources, since one must provide a certain configurability for providing such flexibility. In the EHW field, this configurability has been provided by building a custom evolvable chip or by us-

ing a virtual reconfigurable architecture as presented in section 4.3, with the high costs and inefficiency that those solutions can imply.

In this chapter, two approaches for modifying the architecture of the computation engine by means of partial reconfiguration are presented: a coarse-grained and a fine-grained approach. The main advantage over previous work is the cost efficiency achieved by the partial reconfigurability of commercial FPGAs. Section 7.1 presents a technique for exploiting the FPGAs' modular reconfigurability for evolving a layered spiking neural network. Section 7.2 presents a fine-grained approach, where modifications are not performed at a layer level, but at the level of individual connections.

7.1 Topology evolution of ANNs: a coarse-grained approach

The coarse-grained approach presented in this section considers modules as the minimum reconfigurable part of a system. A module can contain a neural network layer, a fuzzy logic inference rule, or a stage in a multistage filter. It is based on the modular reconfiguration design flow presented in section 2.3, where the module size determines the level of granularity of the reconfiguration.

As an example, this section presents the case of evolution of layered spiking neural networks, which along with the spiking neuron model of section 6.1, has been published in [217, 218]. Different partial bitstreams implementing layer topologies are available for each one of the modules. Then, from a repository of layers, an EA determines the set of layers most adequate for solving the problem. In this way, each layer performs a part of the whole computation.

This type of network fits well into the concept of *modular artificial neural networks* [164]. Modular ANN insert a level of hierarchy by considering a network as a set of interconnected modules. At the same time, these modules contain the neurons which are the basic computing nodes for both modular and traditional ANN. In [164], Ronco and Gawthrop summarize several reasons why modular ANN perform better than traditional networks. The main reason is that complex behaviors require different kinds of knowledge, which is only possible by including a structured network architecture. Dividing the network in modules also allows a more understandable network solution; in a complex task each module is in charge of computing a certain part of the whole computation, and by extracting each module's functionality one can identify the data flow and the decision-making process of the network. From a bio-inspired perspective, modular ANN are more biologically plausible than traditional ANN. The brain is not just a bunch of interconnected neurons, but there are specific areas in the brain in charge of specific tasks.

Another possible approach is the evolution of ANN ensembles [56, 107]. In this case, each module would contain an independent ANN, and an EA would manage a population of them. At the end of the evolution, the provided solution would not a single ANN, but a set of them composed by the best individuals. The mapping genotype-phenotype can involve the parametric adaptations already described, as well as the fine-grained topological adaptations that will be described in section 7.2.

This section proposes a reconfigurable hardware platform using DPR, which tackles the ANN topology-search problem, where the methodology of modular evolution (subsec-

tion 4.3.2.2) is used for setting up the reconfigurability features. Subsection 7.1.1 presents a description of the evolvable platform supporting the ANN evolution. Subsection 7.1.2 describes the hardware substrate necessary to support the proposed platform. Subsection 7.1.3 discuss the implementation of a GA on the hardware platform. Finally, subsection 7.1.4 discuss implementation issues about future work on ANN ensembles.

7.1.1 Description of the platform

As introduced in chapter 5, the proposed platform consists of three parts: a hardware substrate, a computation engine, and an adaptation mechanism. Each of them can be addressed in a separate way; however, they are tightly correlated.

The hardware substrate supports the computation engine. It also provides, the flexibility for allowing the adaptation mechanism to modify the engine. Maximum flexibility could be reached with a software specification of the full system; however, computation with neural networks is a task that is inherently parallel, and microprocessor-based solutions perform poorly as compared to their hardware counterparts. FPGAs provide high performance for parallel computation and enhanced flexibility compared to application specific integrated circuits (ASIC), constituting the best candidate for the required hardware substrate.

The computation engine constitutes the problem solver of the platform. The special interest in spiking neurons is because of their low implementation cost in FPGA architectures [163, 165, 211, 218]; However, any other neuron model can also be considered for the coarse-grained adaptation presented in this section. Other computational techniques are not excluded, such as the fuzzy system described in section 6.2, multistage filters, or simple polynomial functions.

The adaptation mechanism provides the possibility to modify the function described by the computational part. The structural adaptation can be very intuitive given the hardware substrate presented, and consists in a modular structural exploration, where different module combinations are tested by using the modular reconfiguration presented in section 2.3. This principle also applies for any kind of computational technique and can be implemented using different search algorithms such as swarm optimization, a simple GA, or a coevolutionary algorithm. This type of adaptation can be very powerful when combined with a parametric adaptation mechanism. For the presented system implemented with neural networks, this parametric adaptation refers to synaptic-weight learning which implies modifying only the contents of a memory. For neural network implementations it can also refer to ontogenetic methods, such as module-restricted growing and pruning techniques, where neurons might be added to or discarded from the network. In the same way, for other computation methods, the parametric adaptation mechanism must refer to adaptation techniques specific for the given method.

7.1.2 Hardware substrate

A hardware substrate is required to support the platform. It must provide good performance for real-time applications and enough flexibility to allow topology exploration. The substrate must provide a mechanism to test different possible topologies dynamically, to change connectionism, and to allow a sufficiently wide search space. Application specific integrated circuits

(ASICs) provide very high performance, but their flexibility for topology exploration can be reduced to a connection matrix, given the high complexity of ASIC design. Software based solutions display a high degree of flexibility, with very poor performance, making them unsuitable for real-time applications. Programmable logic devices are again the best solution, providing high performance thanks to their hardware specificity, and high flexibility given their dynamic partial reconfigurability.

Under the constraints presented in section 2.3 for DPR in Xilinx devices, I propose a hardware substrate that contains two fixed and one or more reconfigurable modules. Fixed modules constitute the codification and de-codification modules. The codification module, placed at the left side of the FPGA (referring to the schema in figure 2.11), receives signals from the real-world and codifies them as inputs for the neural network. This coding may be a frequency or phase coding for spiking neurons, or a discrete or continuous coding for perceptron neurons. In the same way, the decoding module is positioned at the right side of the FPGA and interprets the outputs from the network to provide real output signals.

Reconfigurable modules contain the neural network; each one of them can contain any component or set of components of the network, such as neurons, layers, connection matrices, and modules in the case of modular ANN. Different possible configurations must be available for each module, allowing different possible combinations of configurations for the network. A search algorithm should be responsible for searching for the best combination of these configurations, specifically, a GA for this case, as presented in the next section.

7.1.3 The proposed on-line evolving ANN

DPR flexibility fits well with topology evolution. The main consequence of the aforementioned features of DPR is a modular structure, where each module communicates solely with his neighbor modules through a bus macro (figure 2.12). This structure matches well with a layered neural-network topology, where each reconfigurable module contains a network layer. Inputs and outputs of the full network are fixed at design time, as well as the number of layers and their interconnectivity (number and direction of connections). While each layer can have any kind of internal connectivity, connections among layers are fixed and restricted to neighboring layers.

For each module, there exists a pool of different possible configurations. Each configuration contains a layer topology (i.e. a certain number of neurons with a given connectivity). As illustrated in figure 7.1, each module can be configured with different layer topologies, provided that they offer the same external view (i.e. the same inputs and outputs). Several generic layer configurations are generated to obtain a library of layers, which may be used for different applications.

A GA [44, 227] is responsible for determining which configuration bitstream is downloaded to the FPGA. The GA considers a full network as an individual (Figure 7.2). For each application the GA may find the combination of layers that best solves the problem. Input and output fixed modules contain the required logic to code and decode external signals and to evaluate the fitness of the individual depending on the application (the fitness can also be evaluated off-chip).

As in any GA the phenotype is mapped from the genome, in this case the combination of layers for a network. Each module has a set of possible configurations and an index is assigned

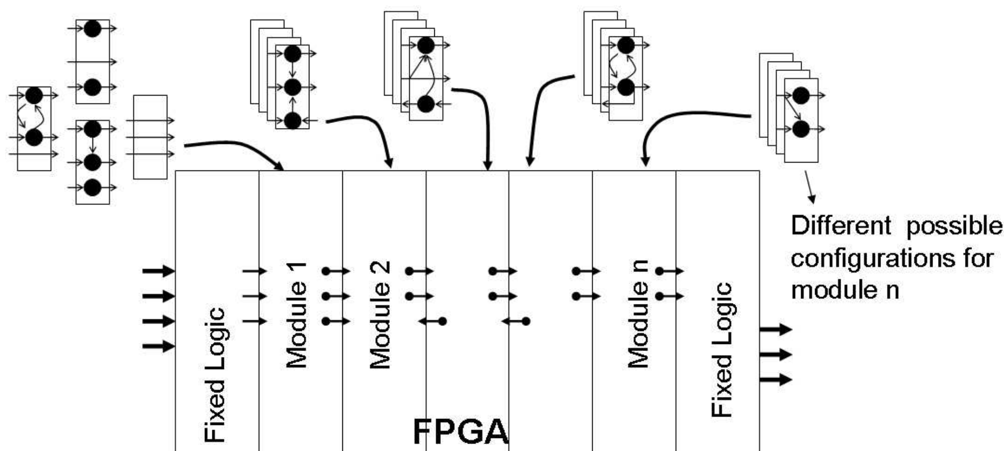


Figure 7.1 *Layout of the reconfigurable network topology.*

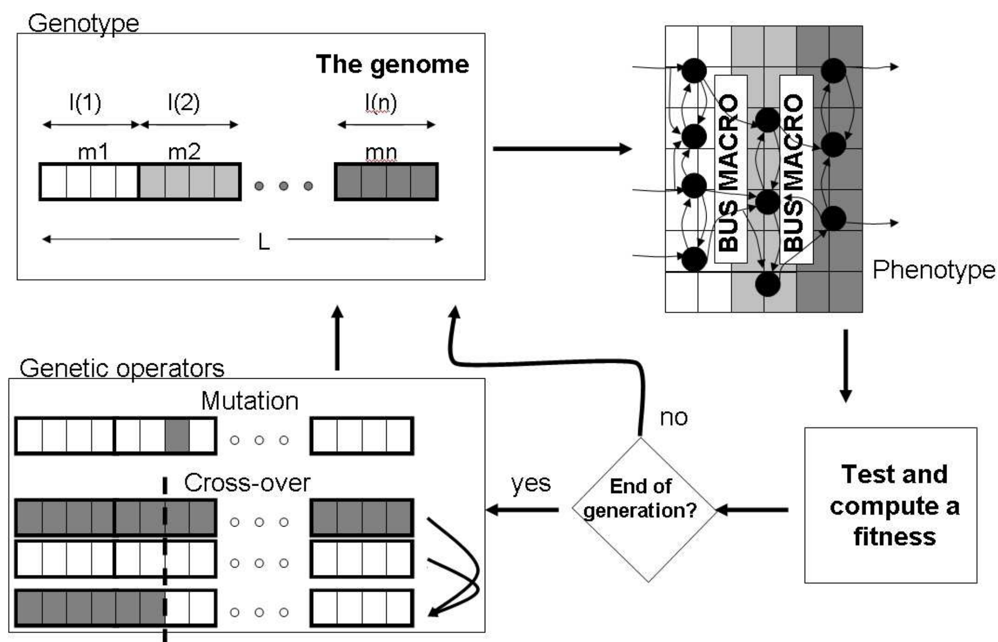


Figure 7.2 *Evolution of a layered neural network. The genome uses a binary codification. The genome maps an individual, a neural network in this case. When a measure of the fitness is obtained, a new individual can be tested, and so on. When the full population is tested the genetic operands can be applied and restart the calculation of the fitness.*

to each configuration. The genome is composed of a vector of these indexes. The genome length for a network with n modules, and $c(i)$ possible configurations for the $i-th$ module (with $i = 1, 2, n$), is given by $L = \sum l(i)$. For a binary genome encoding $l(i) = \log_2 c(i)$, while for a positive integer encoding $l(i) = 1$.

Thanks to the indirect encoding used for mapping the network architecture, one does not have to care about the permutation problem [246], commonly encountered when evolving ANN. By initially generating a set of initial layers, sufficiently different from one another,

one can safely use a GA with typical crossover operations without destroying good network structures.

In a more general framework, one can consider this reconfigurable network as an example of the modular ANN described in the introduction of this section. Related work about evolution of modular ANN has been reported by Happel and Murre [57], where they use modules which are able to categorize and learn in an autonomous way, with an architecture which is evolved at the same time. They thus present, a system involving architectural modular evolution along with individual learning. Such a system fits well in the proposed coarse-grained structural adaptation, where each ANN module corresponds to a reconfigurable module. Learning is performed as a parametric adaptation inside each one of the modules by updating some registers' values.

Another relevant work is presented by Cho and Shimohara in [24], where they use genetic programming [91] for evolving the structure of intermodule connections and the number of nodes in each module. As in the presented example, the network evolution is complemented by synaptic weight learning, in this case by using hebbian learning for adapting intermodule weights. This modular ANN also fits well with the proposed reconfigurable platform, since it requires reconfigurability of intermodule connections and a variable module size.

7.1.4 Modular ANN ensembles

Typically, when training an ANN by using any learning algorithm or any evolutionary technique one obtains as result a single ANN: the one exhibiting the best performance among the set of networks already tested and discarded by the algorithm. However, the resulting network could have been unable to classify some patterns that a discarded ANN was able to classify, or it may perform very well for the training patterns failing to generalize when new patterns are presented.

ANN ensembles aim to exploit the knowledge acquired by less performant ANN in order to provide a more robust solution [56]. The resulting solution is thus a set of the best networks rather than the single very best network. Such a solution may provide better performance for several tasks: a robot navigation control system may use a certain network for avoiding obstacles when required, and another network for following a light source. Another advantage of ANN ensembles are their capability to generalize. Considering, for instance, the decision making of a classification problem. The average classification done by an ANNs ensemble, provides a smoother decision boundary than using a single ANN, featuring consequently better generalization capabilities.

EAs have been used for finding these ANN ensembles. EAs are closely related to ANN ensembles since in both cases a population is involved. In this way, at the end of the evolution, the solution can be defined as the set of the most performant networks. Other ANN selection criteria can consider the diversity of the ANN composing the ensemble. A set of solutions close to each other in the search space may provide very similar decision boundaries, making the ensemble useless because of redundant networks. Maximizing the distance (by using any metrics) between ANN, while keeping only good networks, would increase the generalization ability and the overall performance of the whole system [147]. An example of evolution of ANN ensembles is that presented by Liu and Yao [107], where they use evolutionary programming (EP) [37] for evolving the architecture of feed-forward network: the EP algorithm

defines the number of neurons and connections in each network of the ensemble.

The modular evolvable platform presented in this section fits well with these ANN ensembles. One can envision a final solution where each one of the reconfigurable modules contains an ANN, and the final decision process is performed in a fixed module. With such a reconfigurable system, one can also imagine the usage of these ANN ensembles in dynamic environments, where some of the networks can be dynamically updated according to the environmental requirements.

The EA testing the different ANN can also benefit from this reconfigurability, since it can exploit the enhanced parallelism offered by the platform. Different individuals can be evaluated in parallel and independently by loading each one of them into a reconfigurable module. For algorithms using small population sizes, one can even think about evolving the whole population in parallel.

In the case of ANN ensembles the genotype-phenotype mapping will be very different from the mapping presented in the previous subsection (figure 7.2). In this case, a genome describes the configuration of a single module, instead of the combination of the whole set of modules. The genome can thus contain a given set of parameters, as discussed in chapter 6, or a network topological description by using the reconfiguration technique described in the next section.

7.2 Topology generation for random boolean networks: a fine-grained approach

Randomly connected networks have proven to be universal computing machines. By interconnecting a set of nodes in a random way one can model very complicated non-linear dynamic systems. Although random Boolean networks (RBN) use Boolean functions as their basic component, there are not hardware implementations of such systems. The absence of implementations is mainly due to the arbitrary connectionism exhibited by the network, and connection flexibility is very expensive in terms of hardware resources. This section presents an on-chip self-reconfigurable approach for providing a flexible connectionism at very low resource cost by partially reconfiguring Virtex-II FPGAs, which has been reported in our paper [222].

As presented in section 2.3, Xilinx FPGAs can be partially reconfigured by using several design flows. This section presented a system exhibiting the maximum autonomy and the maximum flexibility. An FPGA system can self-reconfigure for evolving a part of the circuit in a similar way to the system presented in section 6.3 for evolving CA. However, unlike CA, this section presents a system with flexible topology. The methodology of bitstream evolution (subsection 4.3.2.2) is used for setting up the reconfigurability features. Topological modifications are based on the bitstream description of subsection 2.3.2 for dynamically modifying LUTs' and multiplexers' configuration.

7.2.1 Random boolean networks

ANNs are information processing systems able to compute a function in an efficient and parallel way. ANNs are composed of a number of simple components called neurons, nodes, or

cells. These nodes are typically uniform or semi-uniform and are well suited for being adapted to fit a desired function.

Neuron models can have different levels of complexity, ranging from simple Boolean function (McCulloch & Pitts) [125] to the most biologically plausible models (Hodgkin & Huxley) [71]. In all cases, the way in which these nodes are connected is a very important issue.

7.2.1.1 Randomly connected systems

Simplified connectionism schemas have been shown to perform well for several problems. Layered ANNs are widely used for classification and control tasks, while cellular automata dynamics have been widely studied, as they exhibit interesting emergent behaviors. However, biological systems don't use this simplistic connectionism, it being a critical point when building systems targeting self-adaptation and emergence.

Several approaches using arbitrary connectionism have been shown to perform well for several applications. Jaeger and Haas [82] have shown the suitability of echo state networks (randomly connected recurrent ANNs) to predict chaotic time series, improving accuracy by a factor of 2400 over previous techniques. Maass et al. [116] present their liquid state machines (randomly connected integrate and fire neurons), which are able to classify noise-corrupted spoken words. A more simplistic node is used in random boolean networks [39], which consist of a set of N nodes implementing a boolean function, each one with K inputs, randomly interconnected. Several classifications are considered, whether the nodes' state update is performed in a synchronous or asynchronous way, and in a deterministic or random order.

Even if RBN use Boolean functions as their basic node, there are not hardware implementations of such systems. The absence of implementations is mainly due to the random connectionism exhibited by the network and the high cost of connection flexibility in terms of hardware resources.

7.2.1.2 Main differences between RBN and CA

RBN differ in several fundamental aspects from non-uniform CA, making difficult to apply the same rule-adaptation algorithms to both. The main differences are:

1. In RBN, the node's neighbourhood is asymmetric: if A's state is an input to B, it does not imply that B's state is an input to A; in CA, it does.

2. In RBN, nodes' neighbourhood is non-uniform: if A_k is connected to A_{k+1},it doesn't imply that A_{k+1} is connected with A_{k+2}; for CA, it does (for $k+2=N$).

3. CA rules typically consider the topological order in the rule-inputs order: In a 3-inputs rule the input in the middle constitutes the cell state; in RBN, inputs can have any order.

These fundamental differences do not allow one to directly apply algorithms designed for CA to RBN. That's the reason why new algorithms or variations to old ones must be proposed.

7.2.1.3 Cellular programming in RBN

The cellular programming algorithm, presented in subsection 3.2.4, is a distributed EA targeting cellular systems. Unlike most EAs, evolutionary operations are performed locally in each cell of the system, by sharing chromosomes with its neighbors. The cellular specificity of this adaptation mechanism fits well with random boolean networks, even though some considerations must be taken before implementing it.

When implementing an algorithm like cellular programming in a given cellular structure for a given task, one must first analyze whether a genome that was good for a certain cell can potentially be good for its neighbors. Let's suppose, as an extreme case, two completely different nodes: an integrate and fire neuron and a fuzzy rule. The same genome, having two possible mappings to two different structures, would certainly not perform well in both cases.

Considering primarily the third item of the previous subsection, one can deduce that a node's genome that was useful for solving the firefly problem described in subsection 6.3.2.1, having its current state as input, cannot be useful for a node that doesn't have it.

Because of that, this study has been focused on a particular type of neighborhood and cell. The cell consists in a Boolean function with 3 inputs: a random input, its own cell state, and a random input, in that order. This distribution of cell inputs allows the use of the same structure and rule notation that is used in CA [233], while keeping a flexible neighborhood.

Given, also, the neighborhood asymmetry described in the previous section one cannot directly use the standard Cellular Programming algorithm for adapting RBN rules. The concept of neighborhood does not have a placement or index connotation any more, but a connectivity one. This neighborhood paradigm generates new issues. The state of a given cell can be the input of many other cells or it can be completely source-less. On the other hand, one can be sure that there are two cells driving the inputs. Because of this, the neighborhood is considered to be the inputs to the cell instead of the outputs.

Taking into account these considerations, one can apply the Cellular Programming algorithm, described in subsection 3.2.4, to RBN.

7.2.2 The RBN cell array

A hardware architecture of a cellular system allowing a completely arbitrary connectionism constitutes a very hard routing problem. The main problem to face is the scalability. Allowing full connectionism in a 2x2 cellular system is an easy task. However, increasing size implies not only increasing the number of cells but also the size of the multiplexers selecting the nodes inputs. This fact makes the resource requirements increase exponentially when increasing the amount of nodes.

This subsection presents an RBN cell array that allows full implementation scalability. It allows connecting any node with any other; however, some constraints must be introduced to the connectionism: the first connections to route don't have congestion problems, but the further connections are constrained by the routing of the previously connected nodes. Two main advantages of the proposed architecture might : implementation resource efficiency and direct genome mapping.

Figure 7.3 illustrates the RBN cell array: it consists of an array of identical elements, each one containing a rule implemented in a look-up-table (LUT), a flip-flop storing the cells

state, and flexible routing resources implemented in the form of multiplexers.

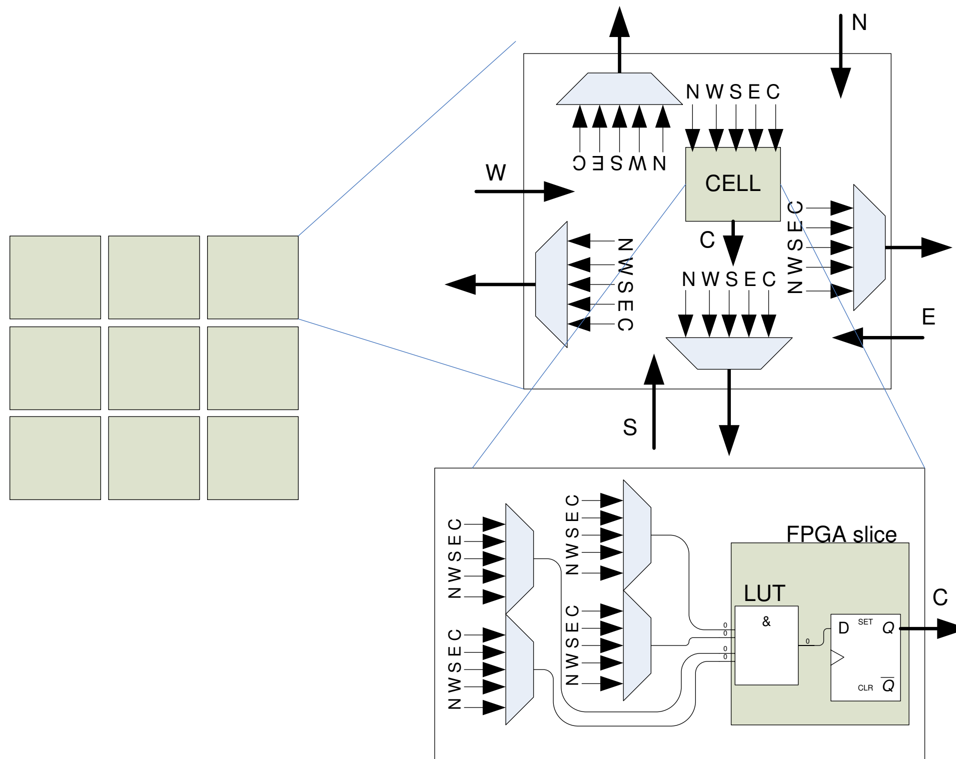


Figure 7.3 *RBN cell: rule and connectionism implementation.*

In the 2-d case, each cell has 4 inputs and 4 outputs corresponding to its four cardinal points: north, west, south, and east, which fits well with the current 2-d IC fabrication technology. Additional dimensions would require two more inputs per dimension.

An output from the cell can be driven by the cell's state or by any other input, allowing the outputs to act as a bypass from distant cell states. In a typical 2-d CA, outputs would always be driven by the cell's state.

A cell's state is updated by a rule -a Boolean function-. As cell outputs, rule inputs can be driven by any input or by the cell's state. If two multiplexers select the same driver the 4-inputs rule becomes a 3-inputs rule, including also the possibility of becoming a 1-input rule if all multiplexers select the same input.

Figure 7.4 shows an example of an implemented network. One can observe that while cell $(3,1)$ has 4 inputs (N, S, E, and C), cell $(3,3)$ has just 2 (N and E), and cell $(1,3)$ has only 1 input (C) and is completely isolated from the other nodes. It must be also noted the existence of floating nets (drive-less nets) in the array. The net created from cell $(1,2)$, to $(2,2)$, to $(2,3)$, and to $(1,3)$, has no driver, and has cell $(2,3)$ as a source. This floating net can be considered as a source of noise, which may be desirable for exploring fault-tolerant or noise-tolerant systems. However, in general it would be an undesirable connection in the network.

One can envision 3 approaches of generating a random connectionism in this array:

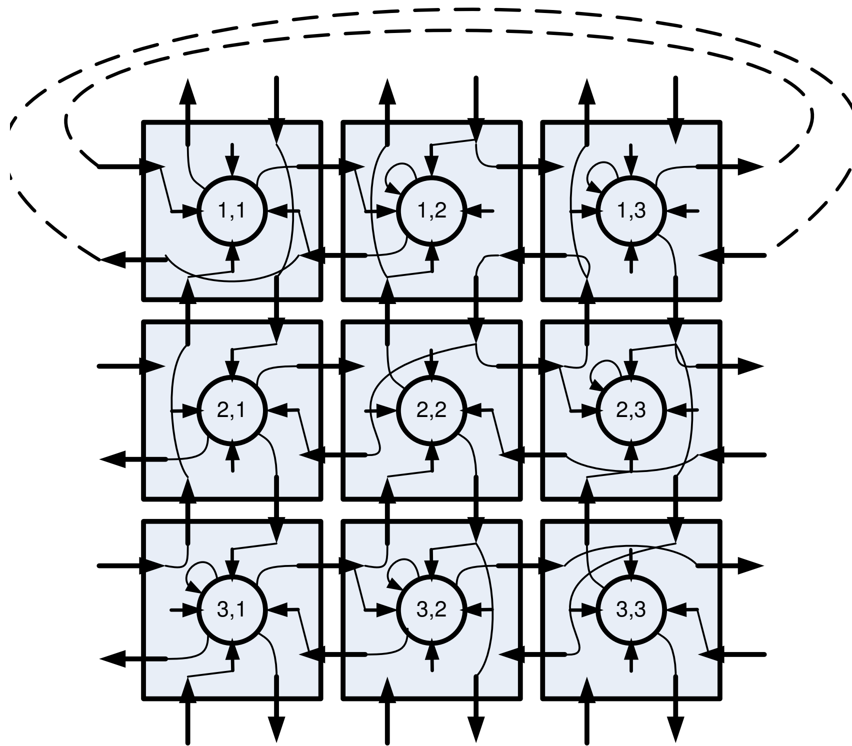


Figure 7.4 *Example of a RBN cell array configuration.*

The *first approach* consists in randomly generating the sources for each node by randomly assigning a cell for driving each cell's input. This approach corresponds to what is traditionally done in software implementations of RBN. This technique requires an additional routing procedure in order to select the multiplexer's states allowing the connections. For slightly larger networks there is a very high probability for the design of being unroutable, given congestion in the network. A possible solution can be to provide more RBN cells than required. In this way nodes can be distributed along the array and congestion problems may be reduced. For a deep analysis of congestion probability in the routing of 4-neighbors cells see [202].

The *second approach* consists in randomly assigning values to multiplexers' selections. This solution avoids congestion problems without requiring the addition of useless cells, by restricting the number of possible network configurations to the ones allowed by the network (in the previous approach this was not the case). Its main advantage is that no routing phase is needed, while the remaining problem is that it can easily generate floating nets as explained before.

The *third approach*, and the one implemented in this work, consists in randomly generating values of multiplexers' selections, while forcing random drivers for floating nets. The pseudo-code depicted in algorithm 2 presents a sequence allowing one to randomly create networks free of floating nets.

This algorithm guarantees that every connection will be part of a net, and every net will be driven by a cell. However, the algorithm doesn't prevent the formation of isolated nodes or sub-nets. This algorithm considers a *connection* as each one of the unidirectional links interconnecting two RBN cells, a *net* is a set of interconnected connections, and a *drive-less*

Algorithm 2 Random routing pseudocode

```
Initially, every connection is drive-less
while drive-less connections exist do
    Begin a net by randomly selecting a non-driven connection as current connection
    while current net is drive-less do
        Assign a random value to the current connection multiplexers' selection
        if selection is C or a connection already used for another net then
            the current net has found a driver
        else if selection is a connection of the current net (it will form a floating net) then
            force selection to C
        else update current connection with the connection driving the current net
        end if
    end while
end while
```

net is a is a net that has not been connected to a cells output.

7.2.3 Setup of the self-reconfigurable system

This section presents the FPGA platform that self-reconfigures the RBN connectionism and Boolean rules through the ICAP. The platform consists in a MicroBlaze soft-processor running on a Virtex-II FPGA from Xilinx. The main advantage of using the vendor-provided soft-processor is the high number of IP peripherals available, and the user-friendly programming environment provided.

7.2.3.1 General system description

The complete system schematic is depicted in figure 7.5. A MicroBlaze soft-processor from Xilinx runs an adaptive algorithm. The program is stored in an internal BRAM, and an external SRAM is used for data storage -i.e. genome storage in the case of evolutionary algorithms. The system interfaces with the external world through an UART peripheral , providing a console for monitoring and debugging from a PC. The RBN cell array to be adapted can be accessed for reading or for writing states through general purpose I/O interfaces; however, connections and rule modifications are exclusively performed through the HWICAP peripheral. The HWICAP module allows the MicroBlaze to read and write the FPGA configuration memory through the Internal Configuration Access Port (ICAP) at run time, enabling the adaptation algorithm to modify the circuit's structure and functionality during the circuit's operation, specifically RBN connections and rules in this case.

7.2.3.2 Reconfigurable RBN implementation

The RBN cell is implemented as a hard macro. Figure 7.6 depicts how it is implemented by using the four slices in a CLB. The RBN cell has 4 inputs from its neighbors: N_in, W_in, S_in, and E_in (summarized as NWSE_in). It has, in the same way, 4 outputs to its neighbors: N_out, W_out, S_out and E_out. Three global input signals are included for system control:

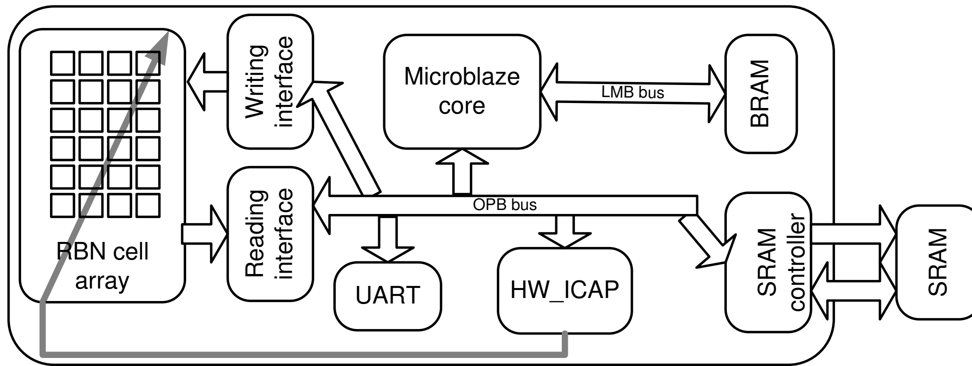


Figure 7.5 *Self-reconfigurable platform setup.*

CLK, EN, and RST. An output signal for observing the cell's state from the processor is also included.

A common alternative to using the FPGA's low level resources would be to define the RBN cell as a virtual reconfigurable circuit. In this case, the reconfigurable circuit is described by a HDL and further synthesized, placed, and routed, by automatic tools. Implementing the RBN cell in this way, would require 18 Virtex-II slices (5 CLBs), while implementing it by defining a hard-macro for further reconfiguring the logic supporting it just requires 4 slices (1 CLB) as depicted in Figure 7.6. In this way, using a virtual reconfigurable substrate would imply an overhead of $\times 4.5$ respective to the implementation in the actual FPGA LUTs and multiplexers.

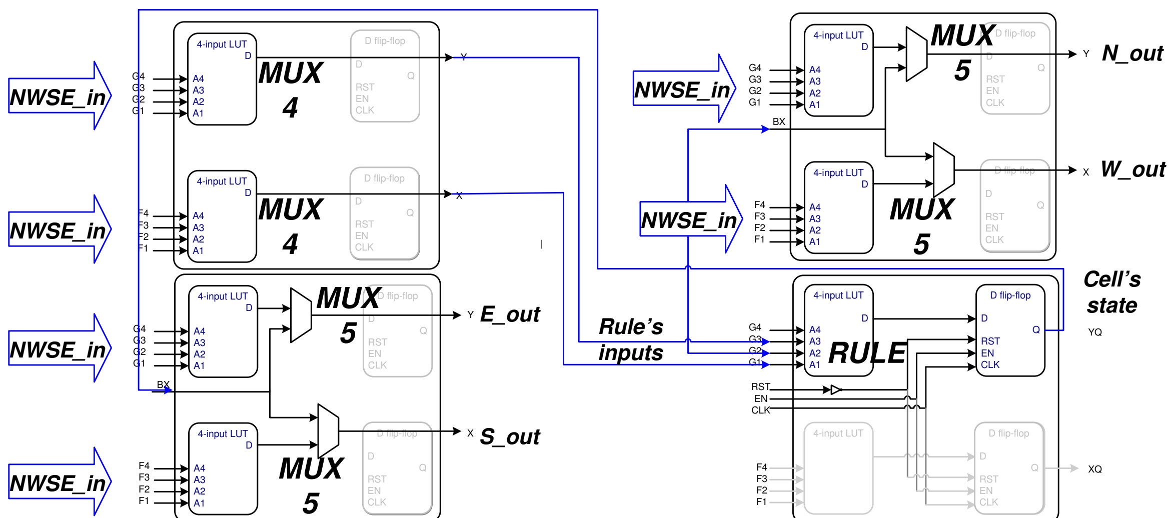


Figure 7.6 *Hard-macro for the RBN cell.*

7.2.4 Example task: firefly synchronization

In artificial cellular systems, the firefly synchronization problem consists in synchronizing the firing of a set of 2-state nodes. Nodes are initialized at a random state, and after a number of iterations each node must switch from one state to the other, synchronizing with its neighbors. This problem has been used for validating the RBN cell.

In this implementation, a 5x6 RBN cell array is implemented on an array of 5x6 CLBs. The MicroBlaze processor initializes the Cell Array by randomly configuring the RBN connectionism. Then the processor executes the Cellular Programming algorithm for RBN described in the sub-section 7.2.1.3. Through the HWICAP peripheral, the genome contents are mapped to the frames containing the LUT and multiplexer configuration information. In this way, the system rewrites 4 frames per array column when reconfiguring connectionism (24 frames in this example) and 1 frame per column when updating rules (6 frames in this case).

Once frames are reconfigured, one can test the RBN through the reading and writing interfaces. The fitness is computed by the MicroBlaze soft-processor, by reading the nodes' states. For computing the fitness, the states are read when the number of iterations is completed, the phase of the majority of the nodes is computed, and then the RBN executes four more iterations. If the sequence is $0-1-0-1$ (or $1-0-1-0$) when the majority phase is 1 (or 0) the fitness is 1, otherwise the fitness is 0. In that way, the fitness for 20 initial states is accumulated in order to obtain the total fitness. Afterwards, a new genome for each cell is generated as described in subsection 3.2.4.

For measuring the performance of the algorithm, 1000 simulations were run. Each simulation consists in:

- Random initialization of connections and rules.
- For 100 generations do:
  - For 20 different initial states do:
    - Random initialization of cell states
    - Let the RBN run for 34 iterations.
    - Compute partial fitness for each cell
  - For each cell, compute total fitness as the sum of partial fitness.
  - Update cell rule according to the cell fitness.
- Deliver the best result -the one with the highest average fitness.

The proposed system successfully finds an RBN able to synchronize the switching of the states. Among the 1000 simulations, 3.4% managed to fully synchronize. it must be noted that the result is highly dependant upon the initial conditions: connectivity and initial rules. A random network with isolated nodes will never fully synchronize; a network not containing initial "good" rules will also have difficulties in converging to a good solution.

7.3 Conclusions

In this section, I presented two approaches for efficiently implementing flexible connecting systems on commercial FPGAs. The low level utilization of FPGA basic components guarantees the efficiency of the approaches. ANN and RBN have been used as case study given their needs of connectionism flexibility, their inherent massive parallelism, and their node structure analogy to reconfigurable hardware logic cells.

In both cases a platform defined by 3 parts has been presented: a hardware substrate, a computation engine, and an adaptation mechanism. Each of these 3 parts is presented along with a description about how can they be merged. The platform design, simulation, and validation is also described, and several options are proposed to apply different computation and adaptation techniques to the platform.

Another important aspect of this chapter is the proposal of an on-chip and on-line self-adaptive flexible system on a reconfigurable platform, which has been always an important issue in self-adapting systems. This chapter describes how to implement it in an efficient way, on current commercial devices.

The proposed system constitutes a novel system approach for evolving hardware. The platform has shown to be suitable for coevolving cellular systems, and the same approach can be easily extended to other connectionism systems -like evolving ANNs or liquid state machines- just by defining the reconfigurable modules as in section 7.1, or by plugging the cells to a hard macro allowing the flexible connectionism as in section 7.2. The system on chip supporting these reconfiguration capabilities provides the hardware platform to support the so called on-chip and on-line self-reconfigurable adaptable systems, by providing the flexibility needed by a real phenotype modification on the evolved hard individual.

The presented structural adaptation mechanisms, when combined with the parametric adaptation mechanisms of the previous chapter, constitute a very powerful problem solver, given its high performance and flexibility. Now the question that arises is: What kind of application can require such degree of flexibility and performance? What application can really exploit the huge capabilities offered by such systems? The next two chapters will suggest some answers by giving some tips about how to exploit these capabilities, and how to implement them in a real system.

Chapter 8

A Reconfigurable Framework for Modular Robotics

The great thing in this world is not
so much where you stand, as in what
direction you are moving.

Oliver Wendell Holmes

A modular robot can be defined as a robotic system consisting of a set of discrete components that can be assembled in different ways to obtain diverse shapes such as snake, quadruped, biped, or hand-like robots. These discrete components are typically referred to as *modules* in literature; however, in this chapter they are called *units* for avoiding confusions with the *reconfigurable modules* introduced in section 2.3. Each one of these units should be autonomous, and should have the capability to "do something" independently from other units (i.e. to move, to sense, to compute ...).

Modular robots offer a robust and flexible framework for exploring adaptive locomotion control. They allow assembling robots of different types e.g. snakelike robots, robots with limbs, and many other different shapes. Locomotion with modular robots holds a great potential and at the same time constitutes a very difficult challenge [22, 97, 248, 249]. In comparison to conventional monolithic robots, modular robots present the advantage of supporting fast reconfiguration of their structure. To build a robot of the desired shape, a completely new robot does not need to be constructed, but can be reassembled by simply disconnecting and reconnecting units. Furthermore, modular robots constitute a challenging framework for exploring distributed control when each unit contains its own controller and sensors.

Self-reconfigurable modular robots arise as a great engineering challenge. These systems are composed of homogeneous or heterogeneous components and have the ability of self-assembling or disassembling multi-unit structures, configuring their shapes without human intervention. This feature also allows the robot to self-repair in case of a unit failure by replacing the failed unit [144, 249].

This special field of robotics holds many interesting challenges in fields as diverse as mechatronics, MEMS (Micro Electro-Mechanical Systems), smart actuators, distributed control, autonomous strategies, learning algorithms, ad-hoc networks, nanotechnologies, bio-inspired systems, etc.

One can foresee that a system with such a degree of mechanical flexibility can benefit greatly from the logic flexibility offered by self-reconfigurable hardware systems. One can foresee, in this framework, the concept of a self-reconfigurable machine, capable of self-reconfiguring its shape (mechanically) and its circuitry (logically). Different types of controllers and functionalities can be tested for different robot shapes, and it can be very interesting for exploring locomotion controllers [79], bio-inspired architectures, learning algorithms, evolutive controllers, etc.

This chapter introduces the YaMoR robot (*Yet another Modular Robot*) in the first section. YaMoR is a modular robot supporting enhanced hardware reconfigurability features along with a wireless framework, which we have reported in [134,135]. YaMoR has been developed at the BIRG (Biologically Inspired Robotics Group), with my co-supervision mainly for the issues concerning hardware reconfigurability. The robot presented in this thesis corresponds to the first prototype; however, we are currently refining the second prototype of the robot.

The second section presents a dynamically reconfigurable framework for the reconfigurable devices present in the robot, for providing a user-friendly framework when implementing reconfigurable controllers. It has been published in [216]. The goal of this section is to show the suitability of using bio-inspired reconfigurable hardware in such a field of application, by providing the modular robot along with the reconfigurable framework, and by linking this to the techniques presented in the previous three sections. However, at the time of this writing, we still have not implemented a bio-inspired controller on the actual robot -the project is still in the controller simulation phase at the BIRG [13, 119].

8.1 Yet another Modular Robot - YaMoR

YaMoR is designed to act as a cheap platform for (1) testing different control algorithm for locomotion and their implementation in both software and hardware, (2) exploring the capabilities for locomotion of a large variety of different robot configurations and shapes, and for (3) finding new applications for wireless networks.

The main characteristics of our modular robots are: (1) each unit contains a Bluetooth interface [50] for inter-unit communication as well as for communication between the units and a base station like a PC (most modular robots use direct electrical connections, which are less flexible), and (2) each unit comprises an FPGA for reconfigurable computation -most modular robots use traditional microcontrollers, but see [46] for an example of a robot using a single FPGA for controlling all units.

A control software called Bluemove [134, 135] has been designed and implemented for allowing a user to control the YaMoR units from a PC via Bluetooth. Bluemove offers an easy way to explore the capabilities for locomotion of different configurations of units.

In the case of the YaMoR project, we are particularly interested in the adaptive control of movement and locomotion in the multi-unit structures. The units described in this section will be used to implement adaptive control of locomotion based on the biological concept of cen-

tral pattern generators (CPGs) [79]. CPGs are neural networks capable of producing coordinated oscillatory signals without any oscillatory inputs. For example, CPGs for swimming and walking motions can be simulated using neural network models or coupled oscillator models [78]. CPGs are an interesting concept for modular robotics because of their distributed nature (they are made of multiple coupled oscillatory networks) and because of their robustness against perturbations and lesions. In our implementation, movement of each robotic unit will be controlled by one or several nonlinear oscillators which will synchronize with their neighbors through coupling connections implemented with the Bluetooth communication protocol. However, implementation of CPGs is not within the framework of this thesis.

Subsection 8.1.1 gives an overview of the mechanics and electronics of our modular robot. Subsection 8.1.2 presents the reconfigurable substrate supporting the robot controllers in the form of an FPGA board. Subsection 8.1.3 describes the Bluetooth interface contained in each robot unit. Subsection 8.1.4 gives an introduction to Bluemove, the control software that is used for exploring locomotion. In subsection 8.1.5, the first examples of locomotion are described illustrating the capabilities of YaMoR.

8.1.1 YaMoR - mechanics and electronics

YaMoR consists of mechanically homogeneous units. One of the key features of YaMoR is its low cost: in contrast to the majority of modular robots, YaMoR is constructed with off-the-shelf components. Each unit contains a powerful one degree of freedom servo motor (with a 73Ncm maximal torque). Its casing consists of cheap printed circuit boards (PCB) that can also serve as support for printed circuits (see Fig. 8.1).

Figure 8.1 *YaMoR unit. Unit closed and with open casing*

The casing of each unit is covered with strong velcro. Velcro offers the advantage of connecting robot units together with no restriction on angles between the surfaces of the units. Unfortunately, it does not support self-reconfiguration and the units can only be connected together by hand.

The units are autonomous. They are powered by on-board Li-Ion batteries and include the necessary electronics for power management, motor control, communication, and running algorithms. To achieve more flexibility and modularity in terms of control each YaMoR unit contains three separated control boards: (1) a board including a Bluetooth-ARM microcontroller combination, (2) a board carrying a Spartan-3 FPGA, and (3) a service board containing power supply and battery management. Currently, we are working on the construction of the second prototype which includes an additional sensor board and a microcontroller board.

YaMoR was constructed as a framework for a variety of different projects. For instance, a user may choose between using a microcontroller, an FPGA or a combination of both for implementing the desired control algorithm. Configuring the FPGA to contain a MicroBlaze soft-processor [234], allows exploitation of the hardware-software codesign capabilities offered by the platform, and also may take advantage of the flexibility provided by the FPGAs partial reconfiguration feature [245].

The YaMoR architecture with distributed electronic components gives a flexible solution for connecting the electronic boards: the FPGA board can be left out, to save energy if it is not needed; or it can be replaced by a board with specific sensors if useful. The new sensor board can still take advantage of the electronics mounted on the remaining boards. So a designer for an additional sensor board does not have to worry about power supply or battery management.

8.1.2 Reconfigurable substrate

The main electronic component of the YaMoR's FPGA board is a Spartan-3 XC3S400 FPGA with 400.000 gates meeting most requirements for reconfigurable hardware. The FPGA board supports two different reconfiguration modes: Slave Serial and Boundary Scan (JTAG). It supports partial reconfiguration and MicroBlaze implementations. The FPGA board also contains a 4 MBit high speed SRAM directly connected to the Spartan-3. The PCB board placement of components takes into account the modular design constraints presented in section 2.3, for supporting a unit in one side of the FPGA which has access to the memory resources without crossing other units. In this way, a soft-processor, for instance a MicroBlaze, can take advantage of the SRAM while using the partial reconfiguration feature (See board in figure 8.2).

The FPGA is directly driven by a 50 MHz oscillator. However, the Digital Clock Manager included in the Spartan-3 FPGA allows modification of the internal clock frequency. General purpose Input-Output pins distributed around the FPGA are accessible through micromatch connectors on the PCB for debugging purposes. A push button allows the implementation of a reset or test input.

8.1.3 Bluetooth - the wireless interface to YaMoR

Bluetooth [50] has been chosen for wireless communication given its flexibility and energy efficiency. Wireless communication between units allows creating a new robot configuration by

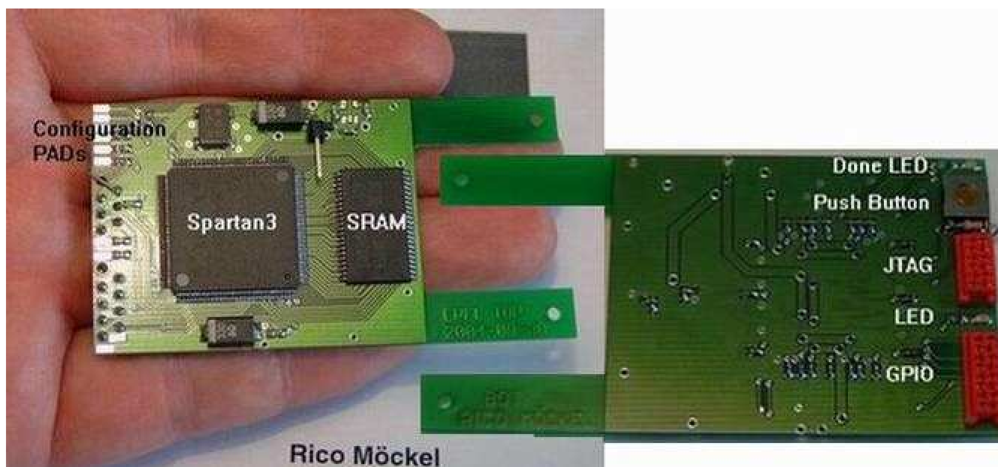


Figure 8.2 *FPGA board.*

simply disconnecting and reconnecting the mechanical units without the need for reconnecting cables or changing the control infrastructure. In comparison to a communication based on wires, Bluetooth has the constraint that a unit normally does not know its physically connected neighbors just by communicating with them. However, this constraint can be overcome with the addition of touch or distance sensors.

The ARM on the Bluetooth board is running both a real time operating system and the embedded Bluetooth stack. It can also be used for customized software e.g. a control algorithm or for reconfiguring the FPGA via Bluetooth. The ARM program code and FPGA configuration bitstream are stored inside a Flash memory on the microcontroller board.

The Bluetooth-ARM board was designed to provide a wireless interface that can be easily controlled. The embedded Bluetooth stack provides two types of services: it supports a remote configuration of the FPGA by using the XSVF format [238] for complete and partial bitstreams, and it allows taking advantage of wireless communication for sending simple commands via UART. For instance, a researcher concentrating on FPGA based algorithms may implement a simple UART module on the FPGA and is able to communicate wirelessly with a PC or other units.

8.1.4 Bluemove - controlling YaMoR via Bluetooth

For easily exploring new configurations of units and their capabilities for locomotion, an interactive Java based control software called Bluemove has been developed. Using a graphical user interface (GUI) on a PC, a user can quickly start a new project, register all units used for the current robot configuration and implement a controller. To control the units Bluemove allows: (1) writing trajectories that can be continuously sent to the units via Bluetooth and interactively modified without any resetting, (2) the use of plugins for controlling the units from a PC, and (3) programming the FPGAs as well as the ARMs for autonomous control in the units without needing a PC. Plugins can act as inputs (hand-drawn trajectories, generators, oscillators, etc.), filters (signal processors, multiplexers, etc.) and outputs (data sent to the units, files, streams, etc.). Plugins support the generation of controllers with feedback from sensors. The whole project including the trajectories and plugins can be saved in XML. The

main parts of the graphical user interface are:

1. The Module Manager (or Unit Manager) serves to manage all units belonging to the current robot configuration including unit names and Bluetooth addresses of the units.

2. The Timelines Manager (see figure 8.3) allows generating trajectories for each actuator of the units registered in the Module Manager by setting key points on the GUI with a mouse. Linear and spline interpolation can be chosen to draw the trajectories and connect the key points set before. Trajectories even can be changed "online" while transmitted to the units e.g. by changing the position of the key points.

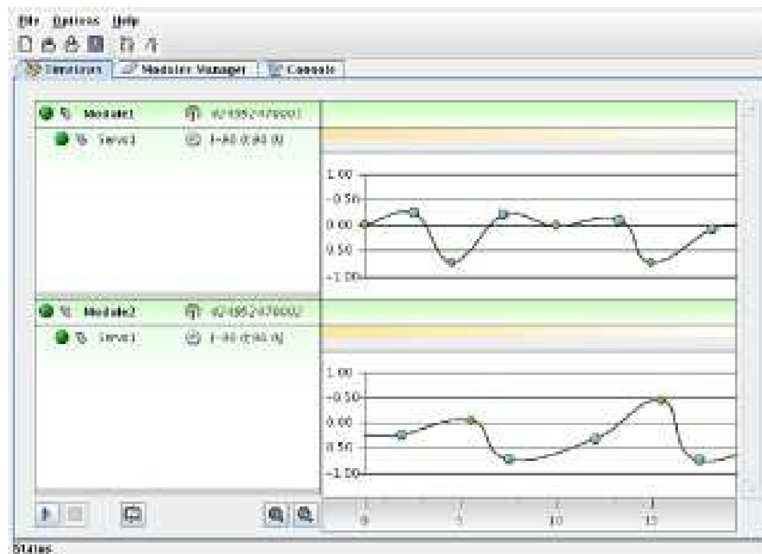


Figure 8.3 *Timelines manager.*

3. The Real-Time Module (see figure 8.4) supports an easy use of Bluemove with the help of plugins. New plugins can be created with a script editor. The relations between different plugins are visualised with the help of a graph.

Bluemove is implemented in Java, taking advantage of its standard and consistent interface for Bluetooth. Given the popularity of Bluetooth and Java, it would be possible to create Bluetooth applications for the modular robot on mobile phones, PDAs or other small systems that support Java and Bluetooth.

8.1.5 Exploring locomotion

The locomotion capabilities of different YaMoR configurations with up to six units have been explored using Bluemove to generate the joint angle trajectories for the servo motors. By trial and error, interesting gaits can be generated for a variety of robot structures, such as travelling waves for worm and "wheel" structures (see figure 8.5 for four snapshots of a moving wheel), crawling gaits for limbed structures, and other peculiar modes of locomotion (see figure 8.6 for different examples of configurations of YaMoR units). For videos, the reader is kindly requested to visit the project website [13].

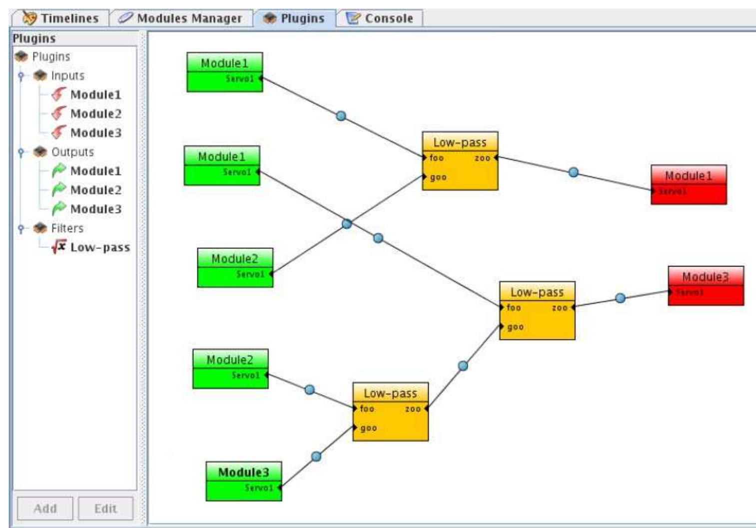


Figure 8.4 *Real-time modules.*

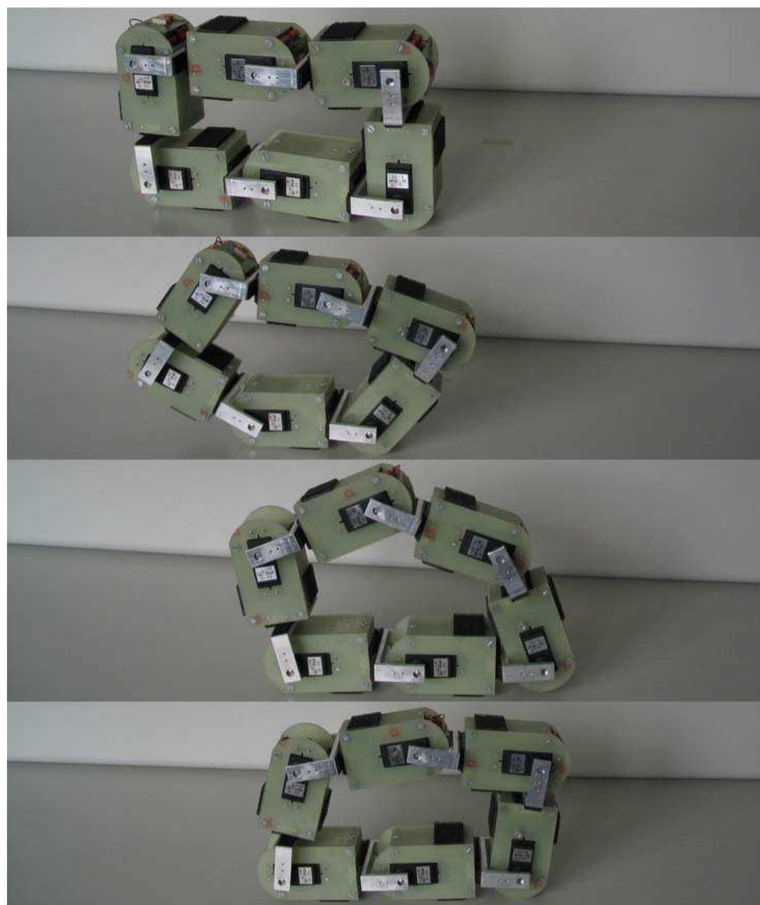


Figure 8.5 *Rolling wheel.*

These gaits are only a first step towards adaptive locomotion, but already represent an interesting example of control with a "human in the loop". The user can indeed interactively

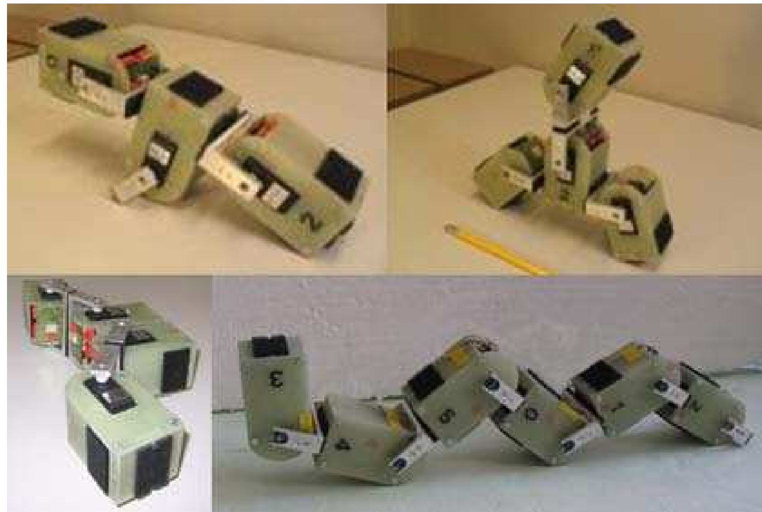


Figure 8.6 *Different configurations of YaMoR units.*

adjust the gaits in real time, in order to optimize the speed of locomotion for instance, as well as modulate the gaits in terms of speed and direction, by modifying frequency and amplitude parameters.

For exploring more complex shapes of a robot, tests with more than six units are needed. For example, most of the configurations tested so far do not allow changing the direction of the movement, and would therefore not be capable of avoiding or overcoming obstacles.

8.2 An FPGA dynamically reconfigurable framework for YaMoR

Dynamic reconfiguration has always constituted a challenge for embedded systems designers. Currently, technological developments make possible to do it on Xilinx FPGAs, but setting up a dynamically reconfigurable system remains a painful and complicated task. This section proposes a framework for performing it in an easy way, for using it in the YaMoR system. An generic architecture is proposed containing a MicroBlaze processor and a reconfigurable module. The module is defined in VHDL and synthesized by the user; then I provide the scripts for easily generating the corresponding configuration bitstreams for a dynamic partial reconfigurable controller for the modular robot. The proposed framework is easily extendable to other applications.

8.2.1 Self-reconfigurable machines

A Self-Reconfigurable Machine is a machine that has the possibility to modify its own hardware configuration. This feature provides an enhanced flexibility that intends to reduce product and computational cost. Computational cost C can be defined in terms of power consumption P and execution time T by the equation $C = \alpha P + (1 - \alpha)T$ where α is a trade-off term for giving more importance to P or T, given the application. These reductions would be mainly

achieved by two facts:

- **Reusability**: The same hardware substrate allows any number of functionalities without increasing chip area, just by reconfiguring the hardware.
- **Power Consumption**: For low power applications, specialized low power circuits can be loaded when required. Additionally, the reusability should avoid consumption in unused circuits.

The proposed platform intends to provide the possibility of self-configuring a system implemented on an FPGA, taking advantage of the Dynamic Partial Reconfiguration (DPR) property from Xilinx FPGAs. The proposed reconfiguration is module-based, providing the possibility of self-reconfiguring a full peripheral (or several of them), a full processor, the full system, or just some components of a peripheral. For our purposes a peripheral can be a neural network, a fuzzy controller, or a coupled oscillators set. Several algorithms implementations on FPGAs have been documented as being faster than on processors, while for power consumption just some case studies have demonstrated that FPGAs can be less energy demanding [154].

The reconfiguration is driven by an embedded soft-processor as shown in figure 8.7. This processor loads partial bitstreams from a bitstream repository via wireless communication, and stores them on an on-board memory; previous works on wireless sensor networks platforms [11, 70] provide an appropriate low power framework. This processor partially reconfigures the system looking for minimizing the computational cost, given a set of operations to be performed for a given task. Operations should be executable by hardware or software as described in [228]; the choice should be done according to the expected computation cost. Future developments on FPGAs technologies should allow a pipelined reconfiguration as described in [45], thanks to reductions in reconfiguration latencies, or storage of backplane bitstreams.

The applicability of self-reconfigurable machines extends to diverse fields such as wearable computing [154], wireless sensor networks [47], and modular robotics as described in this chapter. The first board prototype has been described in subsection 8.1.2. This board provides the requirements needed to implement the above described self-reconfigurable machines. It supports DPR as well as the implementation of a soft-processor, taking into account a set of constraints specified for these types of designs. It provides also a Bluetooth interface for allowing access to the bitstream repository.

8.2.2 Reconfigurable controllers

The process for implementing partially reconfigurable designs in Xilinx devices is a task that remains painful and complicated for FPGA designers (even for experts!!); the goal of this section is to provide the framework necessary to profit from the advantages that DPR offers for self-reconfigurable machines. In the next lines, a basic architecture is presented, along with the scripts required for allowing a non-expert designer, with just some knowledge on VHDL, to implement his own reconfigurable controller for YaMoR.

The approach consists in proposing an initial structure, which is application-dependant. In this case the system will not offer the maximum flexibility, but just the flexibility that should be useful for a given application and for a given board. In this way the controller designer can

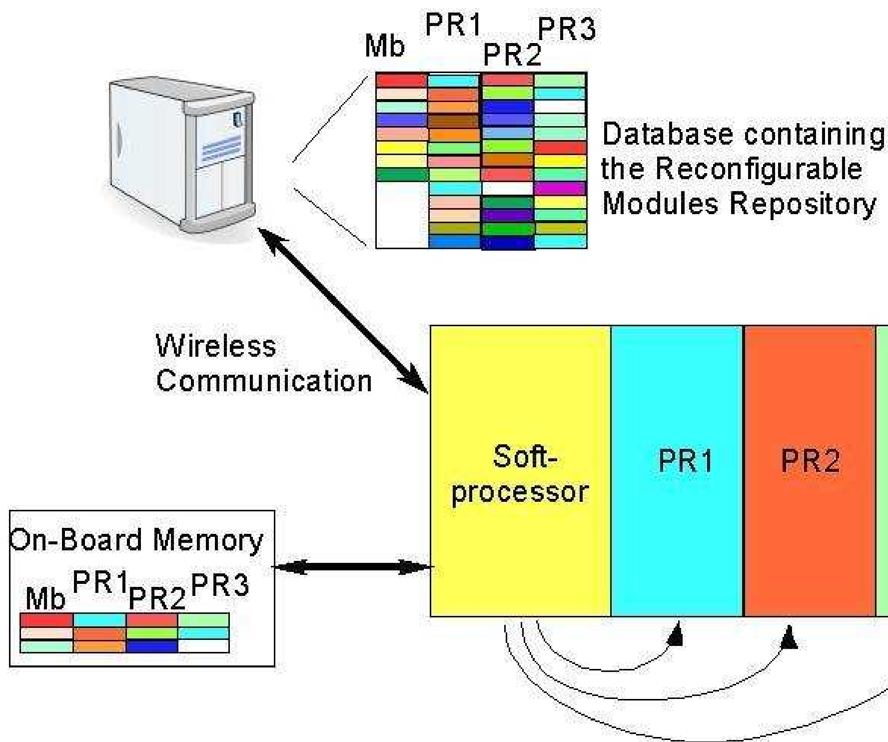


Figure 8.7 *Self-reconfigurable machine schema.*

benefit of user-friendliness under the cost of losing unneeded flexibility. It is clear that the concept of "unneeded flexibility" remains very subjective and nobody can determine what kind of structure is the most appropriated for a given application.

The modular robot control unit disposes of a hardware-software platform. The user can describe the whole controller in software, to be run on a soft-processor, getting rid of all the hardware stuff. Or he can also describe his own hardware peripherals, having the possibility to replace them in a dynamical way, for reducing power consumption or execution time as discussed in subsection 8.2.1.

8.2.2.1 System architecture

The basic architecture contains two modules as depicted in figure 8.8. The first one is a fixed module that mainly contains a MicroBlaze processor from Xilinx [234], featuring a RISC architecture with Harvard-style, separate 32-bit instruction and data busses running at full speed to execute programs and access data from both on-chip and external memory. The second one is a reconfigurable module, allowing the implementation of the user defined logic: the robot controller.

- **Fixed module:** The fixed module contains a MicroBlaze processor including some peripherals: 2 UART ports -one for communicating with the Bluetooth chip, and the second one for monitoring from a PC -, a PWM generator for controlling the servomotors, two 32 bits general purpose input-output (GPIO) for interfacing with the reconfigurable

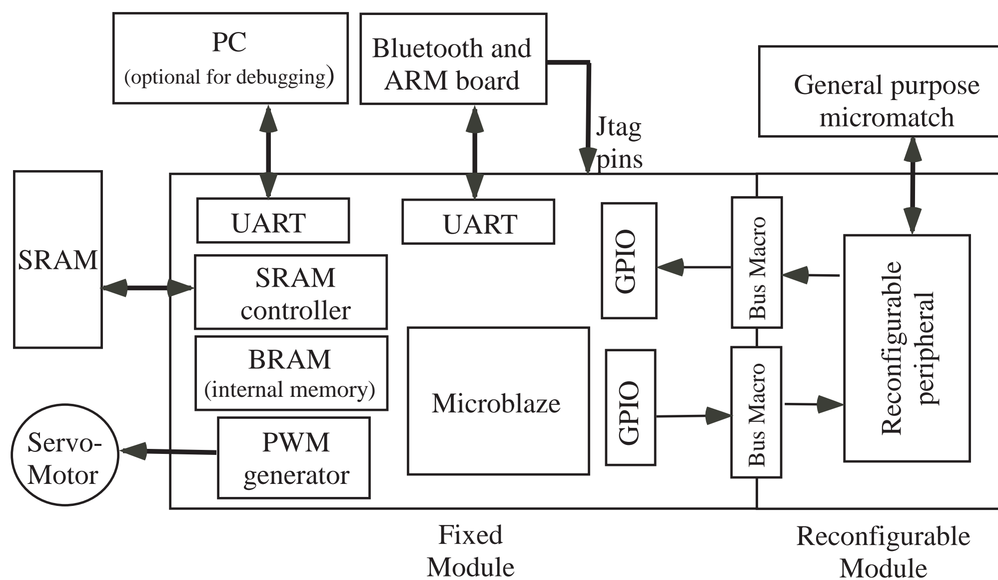


Figure 8.8 *Reconfigurable controller.*

module, and the necessary peripherals for memory managing: external SRAM controller and internal block RAM (BRAM) data and instructions controllers.

- **Reconfigurable module:** The reconfigurable module is connected to the MicroBlaze GPIOs through a Bus Macro, making the module content a peripheral. This module can also access four external pads, connected to an external micromatch, allowing a direct interface of the reconfigurable module from outside the FPGA for debugging purposes. This module allows to reconfigure a peripheral contained in it, while keeping the processor core. Different kinds of hardware controllers are interesting in the framework of the YaMoR project, namely: non-linear oscillators, neural networks, and fuzzy logic systems.

8.2.2.2 DPR on Spartan-3 devices

DPR is supported for Virtex families (E, II, II-pro, IV); however, even though Spartan families (E, II, II-E, 3) can be partially reconfigured, they do not have an ICAP and the dynamic reconfiguration feature is not supported -i.e. the FPGA can be partially reconfigured, but the unaffected logic is disabled. This limitation on Spartan-3 FPGAs does not allow reconfiguring modules directly by the soft-processor contained in the FPGA. Instead of this, partial reconfigurations, as well as the initial configuration, are done by the external ARM microcontroller contained in the Bluetooth chip.

As explained in section 2.3, modular partial reconfiguration requires the bus macros based on 3-state buffers depicted in figure 2.12. However, Spartan-3 FPGAs do not have 3-state buffers, and Xilinx does not provide bus macros for them. That is the reason why I have designed a special type of Bus Macros. Bus Macros are usually implemented with internal 3-state buffers with the only goal of guaranteeing a fixed connectivity among modules for every reconfigurable module. Instead of 3-state buffers (not available in Spartan-3), there were used

slices' LUTs. Figure 8.9 depicts the implementation of a LUT-based bus macro: the inputs in the left are bypassed to the outputs in the right, while the routing among LUTs is fixed for every reconfigurable module. Two bus macros are provided: one for signals going from left to right, and another one for the inverse. Specifically, for the Spartan-3 XC3S400 it allows a maximum bus macro width of 132 bits between two adjacent modules.

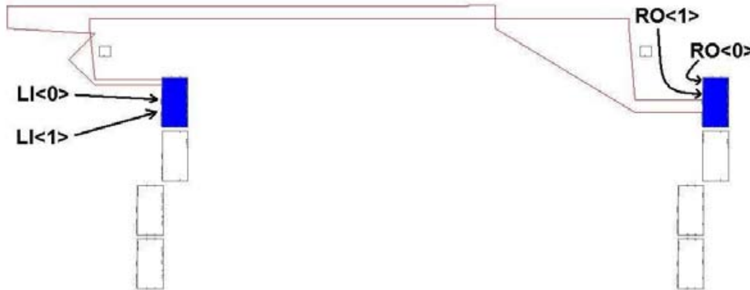


Figure 8.9 *LUT-based bus macro for Spartan-3 FPGAs.*

Additionally, the module-based flow for the Spartan-3 family is currently not documented and supported by Xilinx. The bitgen tool (bitstream generator) does not allow generating partial bitstreams for modular designs, but just for difference-based designs. For dealing with this problem I use the difference-based bitstream generation for emulating the module-based one by executing the following steps:

1. Assembling of a complete design for each possible configuration of the system -i.e. a full system including fixed and reconfigurable module.

2. Generation of partial bitstreams containing the difference between the initial system and each one of the remaining configurations and vice versa.

3. Configuration of the FPGA with the initial bitstream.

4. For loading a module, a partial bitstream is downloaded, which only contains the difference between the initial and the second system.

5. Before downloading a new partial bitstream containing a third system, unlike in regular module-based flow, one must come back to the initial system, since directly downloading it may result in internal contentions.

If the number of possible systems is not very large, it would be possible to generate the partial bitstreams for switching from any system to any other one, for avoiding to come back to the initial configuration.

Given that the proposed reconfiguration is difference-based, it would be possible to get rid of all the module-based flow, and it would still be correct. However, doing that would dramatically increase the size of the partial bitstream, since the modular flow ensures that the difference bitstream will just contain the reconfigurable module, keeping unchanged the microprocessor module.

8.2.2.3 Bitstream generation

As explained before, the bitstream generation uses to be complicated. Given the huge complexity of FPGAs configuration bitstreams, Xilinx tools are still not very well debugged and for each reconfigurable design you have to deal with lots of incomprehensible error messages. A given system working properly on a given FPGA can generate errors when changing the FPGA or when modifying the system. Usually, these errors can be solved by "finding alternative paths": dealing with placements constraints, with tools options, or manually placing and routing components.

Given that the proposed architecture targets a specific application and hardware platform and that several problems have already been solved, it will save a lot of time to the user if he does not have to solve them again. A set of scripts is provided for dealing with these problems. However, it is clear that for complex reconfigurable modules new problems can appear.

Some of the proposed scripts are:

- *create_project n*: creates the required directory structure, for a number n of reconfigurable modules, and copies the required initial files -user constraint file, bus macros, and netlists files for top level and the fixed module.
- *run_rec_module i*: runs the ngdbuild, map, par, and pimcreate for the module i. i can be "system" (standing for the fixed module) or "1","2"..." n" (index number of the reconfigurable module).
- *assemble_complete i*: runs the final assembly phase for the module i, generating a complete bitstream containing the processor and the module i.
- *assemble_partial i k*: runs the final assembly phase for the module i, generating a partial bitstream containing the module i considering the module k as the initial system.
- *run_all*: after creating a project and copying the modules' netlists, this script calls the scripts run_initial, run_rec_module, assemble_complete, and assemble_partial, for generating a complete bitstream with the processor module and the reconfigurable module 0, as well as the partial bitstreams for remaining modules.

It must be noted that these scripts can be reused by other designs and applications, and can be easily modified when new problems are found and solved.

8.3 Conclusions

In this chapter I presented the motivations for including bio-inspired reconfigurable systems on modular robots, along with an initial platform and some methodology proposals. I am convinced that an application such as modular robotics, where mechanical flexibility is the main goal, can benefit greatly from the flexibility and performance offered by bio-inspired reconfigurable hardware systems. Dynamically modifying controllers for different types of dynamically modified robot shapes can greatly enhance the capabilities of these robots.

As a complement to the partial reconfiguration techniques presented in section 2.3, a technique for performing partial reconfiguration on Spartan-3 FPGAs (which is not supported

by the FPGA vendor)is also presented. A module-based partial reconfiguration is emulated by using the difference-based one. The section discuss also the advantages, limitations and possible improvements for this technique.

The presented approach may increase system flexibility thanks to DPR, while keeping low memory requirements given the Bluetooth access to a bitstream repository. This wireless channel may also simplify the process of loading an initial bitstream (a complete bitstream), as well as a partial bitstream, which can be very painful when reconfiguring a set of robots.

The proposed framework remains simple and user-friendly; additionally it provides enough flexibility for the specific application. This approach can be extended to more demanding applications by adding more reconfigurable modules, and faster peripheral interfaces for connecting to modules such as IPIF instead of GPIO.

To achieve adaptive locomotion, the "human in the loop" control that we were using for the first experiments is clearly not sufficient. In a complex environment a robot has to react and adapt in real time. That is why Bluemove has been designed, to support feedback signals from sensors with the help of plugins. Distributed algorithms can be implemented in the ARM processor and FPGA of each YaMoR unit. Both the parameters of these algorithms and the shape of the modular robot can be optimized under different constraints, like energy efficiency or speed.

In the framework of the YaMoR project, we are currently extending the presented work along two main axes: (1) the design of the next generation of YaMoR units with more computational power and sensing capabilities (IR sensors, inertial sensors, and contact sensors), and (2) the implementation of distributed locomotion controllers based on central pattern generators. Central pattern generators (CPG) are biological neural networks capable of producing coordinated patterns of rhythmic activity while being initiated and modulated by simple input signals. CPG simulation experiments have demonstrated to be ideally suited for implementing distributed control of locomotion in simulated YaMoR units [119].

Chapter 9

Reconfigurable Pervasive Systems

Security is mostly a superstition. It
does not exist in nature...
life is either a daring adventure or
nothing.
Helen Keller

These days, pervasive systems are increasingly becoming established in our lives. Pervasive systems are defined as embedded systems that automatically adapt to changes in their environment and operate based on users' needs [166]. They can communicate among themselves; they are mobile, context-aware, invisible, and omnipresent.

In this context, customer electronics devices provide more functions day by day: video and audio processing, communications, entertainment, etc. At the same time, these functions demand more complex support: operating systems, secure communications, etc. Guaranteeing high performance for these applications is not possible when the processing is fully performed by software. A common approach for improving performance is to include specialized hardwired coprocessor units. However, given their static architecture, these systems lack flexibility, and having specialized coprocessors for each task is not feasible given the amount of logic required and the possible incompatibility of upgraded versions of the algorithms.

When implementing pervasive systems one faces several new paradigms not present in the design flow of traditional embedded systems. Among the key challenges raised by pervasive systems, we find several issues:

- **Security:** Given their omnipresence, pervasive systems are highly vulnerable. Authentication and confidentiality must be provided in order to trust other devices and to protect data communication. Cryptography allows the addressing of both of those issues.

- **Seamless communication:** Given the mobility inherent to pervasive systems, wireless communications are essential for satisfying their interaction requirements. This communication requires specialized protocols, able to handle high mobility.

- **Flexibility:** Presently, pervasive systems provide a wide range of functions. At the same time these functions are constantly being upgraded. How to deal with this dynamic scenario?

Pervasive systems are intended to interact with a nondeterministic environment; consequently, it is not possible to plan at design time the evolution of the application functions used by a pervasive system (e.g. signal processing). In addition, these applications require a set of services in order to guarantee a robust and reliable framework. Among these services one find operating systems, routing protocols, and cryptographic algorithms, which imply an additional computational cost to the application itself. This complex computational framework imposes very hard design constraints. The system must be thus performant, flexible, and must exhibit low power consumption.

Performance, flexibility, and low-power are mutually exclusive features in traditional static systems. Traditional applications must target one of them when designing a given system, as it has been difficult to find a reasonable trade-off between them. However, bio-inspired reconfigurable hardware systems, or reconfigurable computing in a more general scope, offer the framework for providing the best trade-off between them. High performance can be provided by the hardware nature of the coprocessed solutions proposed. Flexibility is guaranteed by two facts: the upgradeability offered by the partial reconfigurability, and the adaptability offered by bio-inspired techniques. Power consumption is reduced given the fact that only a single coprocessor (or a few of them) is present at a given moment, and no power is consumed by unused coprocessing elements. However, it must be said also that low-power is not among the strengths of current FPGA devices when compared to ASIC solutions.

Reconfigurability is still not among the features of current pervasive systems; however, performance and flexibility requirements are constantly increasing and eventually reconfigurable computing will arise as the solution for tackling pervasive systems design. As a first platform for exploring issues in this direction, this chapter presents ROPES (Reconfigurable Object for Pervasive Systems), along with two reconfigurable systems targeting it.

The first section introduces the ROPES platform, which has been developed in collaboration with the Reconfigurable & Embedded Digital Systems group (REDS) at the *Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud* (HEIG-VD).

The second section presents an example of a complete reconfigurable system, which is reported in [98]. Even though the presented system is a cryptographic coprocessor not involving any of the aforementioned bio-inspired hardware architectures and techniques, it provides a full reconfigurable pervasive system, which is very important for supporting stand-alone bio-inspired hardware systems. The presented system provides a set of system components required for supporting pervasive systems based on bio-inspired reconfigurable hardware. The system is composed of: a soft-processor, an operating system (uClinux), communication capabilities (Ethernet and Bluetooth), cryptography, self-reconfigurable coprocessors, and the required support for loading (and sharing) remote partial bitstreams.

Finally, the third section presents a self-reconfigurable system targeting the problem of channel equalization. The system is composed of a particle swarm optimizer as the adaptation mechanism, and a ANN as the computation engine. This work has been reported in [149]. The work presented in this section has not been implemented on the ROPES platform; however, it deals with a problem typical of mobile wireless systems (channel equalization). This section

shows that an approach based on the decomposition of a computation engine modified by a adaptation mechanism as developed in this thesis, performs better than previous works reported for this specific task, and for optimizing testbench functions.

9.1 Reconfigurable Object for Pervasive Systems - ROPES

ROPES (figure 9.1) is a prototyping platform for reconfigurable pervasive systems. It provides enhanced reconfigurability capabilities, along with several communication channels. The system is modular for allowing the designer to customize the platform in which the reconfigurable pervasive system will be prototyped.

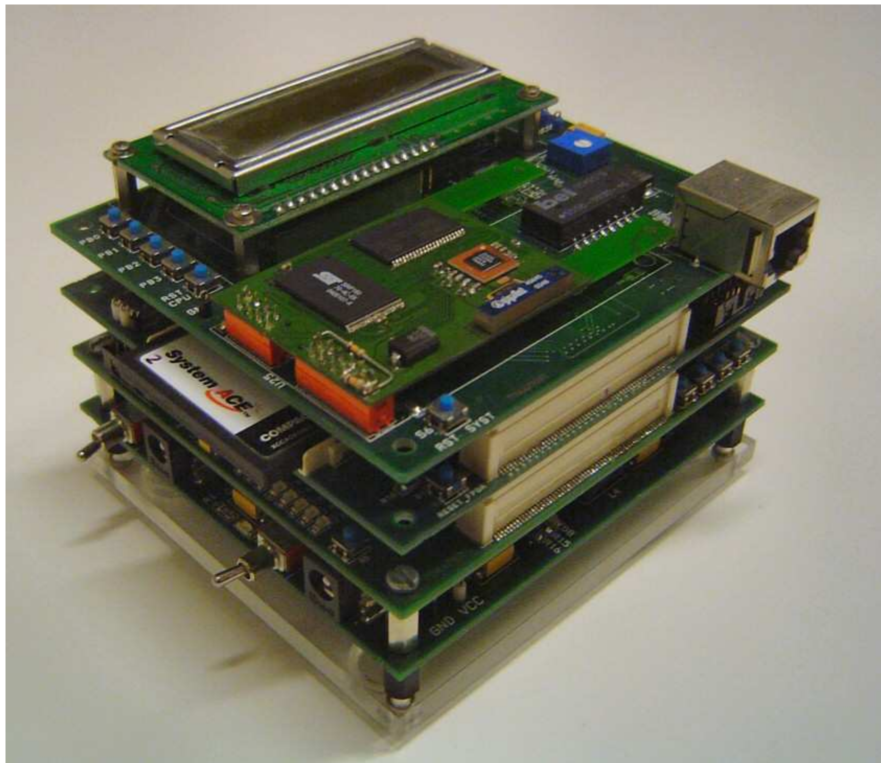


Figure 9.1 *Reconfigurable Object for Pervasive Systems - ROPES.*

ROPES provides pervasive reconfigurable capabilities, well suited for DPR (with an appropriate pin locating strategy). It includes: a Virtex-II 1000 Xilinx FPGA, strong communication features (Bluetooth, Ethernet), and different types of memory (32 MB SDRAM, 1 MB SRAM, and a CompactFlash drive).

9.1.1 ROPES layout

One of the key features of ROPES is the flexibility provided by the modularity: it is constituted of several modules (layers), arranged one on top of the other as shown in figure 9.1, which communicate among themselves through two buses of 120 bits. In this way the physical size of the system is tailored to the application's needs.

The first (bottom) layer is the power supply. The second layer contains the FPGA with two SRAM chips and some LEDs and push buttons. The third layer features a CompactFlash card reader which can be used to configure the FPGA through SystemACE [237]. Finally the fourth layer consists of a display, a 32 MB SDRAM, a Ethernet PHY chip and a Bluetooth mini-board. The Bluetooth miniboard is a replacement for the RS-232 cable connection, and the FPGA reconfiguration can also be performed through this interface as it is connected to the JTAG port.

9.1.2 FPGA board

The FPGA board contains a Virtex-II XC2V1000FG456 from Xilinx, two SRAM memories of 32KB each, a serial interface, and some leds and buttons. The main interest in developing our own platform is to allow the board's layout to support the modular design flow presented in section 2.3. That section, which described the partial dynamic reconfiguration of Xilinx devices, introduced a set of constraints that must be satisfied for implementing a module-based partially reconfigurable system. When designing a PCB the main constraint to consider is IOB placement; an IOB can only be accessed by the adjacent module. Consequently, if one wants the FPGA to contain a microcontroller (a MicroBlaze, for instance) which can access the external SRAM and which has self-reconfigurable capabilities, one must take care to place the memory pins together at the right side of the FPGA. In this way the microcontroller can be placed in a module at the right side of the FPGA for accessing the memory, and for accessing the ICAP port, which is placed at the bottom right in Virtex-II devices.

Figure 9.2 shows an example of the implementation layout of a reconfigurable system in ROPES. The microcontroller is contained in a module at the right side of the device; in this way it can access the external memory and the ICAP port. The left side module contains a reconfigurable coprocessor which can be reconfigured by the microcontroller through the ICAP port, and which is connected to the microcontroller through a standard bus (OPB or FSL) by using bus macros in the physical implementation. Section 9.2 presents, in a more detailed way, the implementation of a reconfigurable cryptographic coprocessor.

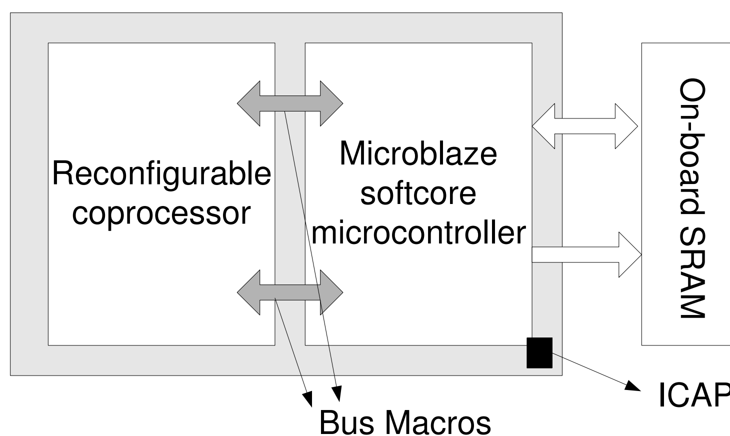


Figure 9.2 *Layout of a reconfigurable system in ROPES.*

9.1.3 Communications and reconfigurability

The nature of pervasive systems implies that they must be distributed, context aware, and mobile in most cases. In order for these systems to support this, they must have the possibility to communicate among themselves. Additionally, when the pervasive platform is supposed to be reconfigurable, it must support the possibility of being remotely reconfigured, supporting in this way remote upgrades.

The fourth layer board contains an Ethernet physical layer transceiver (PHY), providing access to a computer network. The transceiver is a LAN83C185 from SMSC, compliant with IEEE 802.3/ 802.3u standards, supporting 10 and 100 Mbps operation. In this way, the ethernet PHY can be accessed from the FPGA by including, for instance, the *OPB Ethernet Media Access Controller* (EMAC) which is included in the Embedded Development Kit for developing systems containing the MicroBlaze and PowerPC microcontrollers.

This fourth layer provides also the possibility of plugging in the same Bluetooth board designed for the YaMoR modular robot, presented in subsection 8.1.3. In this way, the operating system and the Bluetooth stack developed for the YaMoR project can be reused in ROPES. For the moment, the Bluetooth communication supports two options: it can be used as a remote console for the platform by replacing the UART port, or it may be used to support the remote configuration of the FPGA, for both complete and partial bitstreams.

The third layer board contains a *system advanced configuration environment solution*, best known as SystemACE [237], along with a compact flash card reader that allows storing several configuration bitstreams (complete and partial) for the FPGA. The SystemACE also provides a microprocessor interface (called MPU) that allows one to control and monitor the operation of the SystemACE. Through the MPU, one can read and write data to the compact flash, start a configuration, determine the source of configuration, and control the bitstream version, among other functions. In ROPES the MPU is connected to the FPGA, with the purpose of allowing a MicroBlaze microcontroller to determine when to reconfigure the FPGA, and to select the desired bitstream from among a set of them.

9.2 Self-reconfigurable pervasive platform for cryptographic application

The complexity exhibited by pervasive systems is constantly increasing. Customer electronics devices provide a larger amount of functionalities day by day. A common approach for guaranteeing high performance is to include specialized coprocessor units. However, these systems lack flexibility, since one must define in advance the coprocessor functionality. A solution to this problem is to use run-time reconfigurable coprocessors, exploiting the advantages of hardware while keeping a flexible platform.

This section describes a self-reconfigurable pervasive platform containing a dynamically reconfigurable cryptographic coprocessor. There are considered three ciphering algorithms and there are compared the performance of the coprocessor-based solution against a full-software implementation.

As case-study, a reconfigurable cryptographic coprocessor is presented, improving performance while providing hardware flexibility. Subsection 9.2.1 introduces the concept of

reconfigurable pervasive systems. Subsection 9.2.2 focuses on the cryptographic application, explaining the different ciphering algorithms tested in the platform. In subsection 9.2.3, the system is detailed by describing the hardware platform, the processor, the coprocessor and the operating system. Finally, subsection 9.2.4 gives the results and explains how they are measured.

9.2.1 Reconfigurable computing in pervasive systems

Reconfigurable computing refers to the ability to modify the physical circuit functionality of a logic chip. Reconfigurable devices' functionality can be dynamically modified, thus enabling a hybrid computer structure combining the flexibility of software with the speed of hardware. Reconfigurable computing appears in a wide range of applications because one can tailor the logic to the needs of the application. Among these, one can find cryptography and data compression.

Pervasive systems bring embedded computation into the environment, in order to allow a more natural interaction with them. Instead of having a traditional computing system, the use of many devices allows a seamless integration and an automatic adaptation to the user. Pervasive systems' computation is highly dynamic, nodes can be constantly changing their position, their environment, their neighborhood (i.e. other accessible pervasive nodes), and their functionality, depending on the previous items. Reconfigurable systems are easily integrated in pervasive systems, since they offer the flexibility to adapt their function by reconfiguring the device.

9.2.2 Cryptography

Pervasive systems are particularly vulnerable to data falsification and being misused due to their mobile nature. The need for data protection and authentication is addressed by cryptography. Since ubiquitous data can be very large, high-speed cryptography is essential. Ciphering algorithms are well known for benefiting significantly from hardware acceleration [83]; they can be highly computing-resource intensive when processing large amounts of data. That is the reason why they represent an excellent testbench for the proposed self-reconfigurable pervasive platform.

In order to cover different widely used algorithms, the presented system handles three ciphering algorithms: RC4 [162], DES [192] and Triple DES [95].

The *Rivest Cypher 4* (RC4) is a stream cypher with byte wise processing [162]. The principle of the cypher is to generate series of pseudo-random bytes, encrypted by XORing the random bytes with the input data bytes. For RC4, the encryption and decryption is performed in the same way, making it possible to use the same implementation for both.

Figure 9.3 depicts the schematic of the hardware implementation of the RC4 algorithm. A Permutation of 256 bytes is initialized by the key and stored into a BlockRAM to optimize the performance. The key stream generated by the key setup is then used in the pseudo-random number generator to create a stream of bits which is XORed with the input stream.

The *Data Encryption Standard* (DES) is a block cypher that operates on data blocks of 64 bits by using a key of 56 bits. It operates on left and right halves of a block of bits in multiple

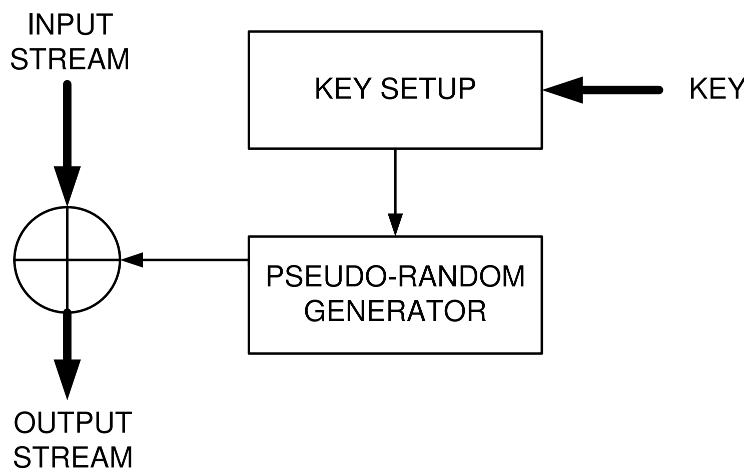


Figure 9.3 *RC4 hardware implementation.*

rounds. In each round, halves are exchanged from their previous round order, until completing 16 rounds.

In the coprocessor module, there are not enough logic resources for implementing a full pipelined 16-round implementation, so a single round was implemented as illustrated in figure 9.4, and a finite state machine to control the execution of the 16 required rounds.

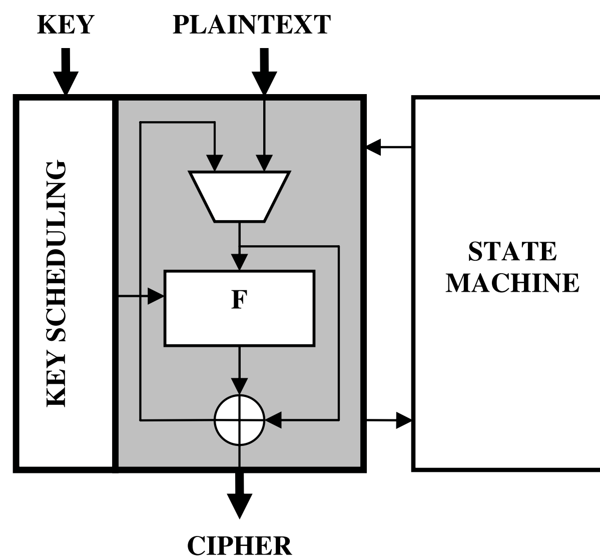


Figure 9.4 *DES hardware implementation.*

The lack of security offered by the DES algorithm, given its relatively short key, has been strengthened with the *Triple DES* [95]. Triple DES uses a 168-bit (56x3) key, and basically performs the simple DES three times, in a three-phase execution of encryption-decryption-encryption.

For the hardware implementation of the triple DES, the same datapath implementation as for the simple DES was used, which is depicted in figure 9.4. The only difference is that

the state machine, instead of a single encryption, must perform the three phases: encryption-decryption-encryption.

9.2.3 System description

The whole system is composed of a MicroBlaze processor and a reconfigurable coprocessor that performs the cryptographic function. The uClinux operating system [229] running on the processor allows managing the system and the peripherals. The processor can reconfigure the coprocessor in a dynamic way by sending partial configuration bitstreams through the ICAP port.

9.2.3.1 Processor

Currently, commercial reconfigurable devices offer two types of processors: hardcore and softcore. A hardcore processor is implemented directly in IC transistors, achieving maximal performance, while a softcore processor is an IP core which is implemented on the FPGA's logic cells. The main benefits of using a softcore processor include a configurable trade-off between cost and performance, easy integration in the FPGA, and upgradeability. The price to pay is a lower performance compared to hardcore processors.

Several softcore processors are available for Xilinx FPGAs: MicroBlaze [234], LEON [38], and OpenRISC [146], among others. For this implementation, a MicroBlaze processor was selected, since it offers several advantages: it is optimized for Xilinx FPGAs, it has a low area usage, it is well documented, and it has good development tool support [123].

9.2.3.2 Coprocessor

A coprocessor adds computing power, leaving available the main processor for other tasks. The encryption is done by the coprocessor, so the communication latency between it and the processor must be minimal for achieving a maximal throughput. Among the several ways to interface a coprocessor to a MicroBlaze, the Fast Simplex Link (FSL) provides maximal performance thanks to its FIFO-like interface [242].

As illustrated in figure 9.5, there are 3 different types of data to be communicated between the coprocessor and MicroBlaze. First, there is a 64 bit data bus for the input of the coprocessor (*Data_in1* and *Data_in2*), a 64 bit data bus for the output of the coprocessor (*Data_out1* and *Data_out2*), and a 32 bit data bus for the key (which is sent in several blocks). So in total there are 2 FSL links for data (2 slaves and 2 masters), and one FSL link for the key (only one master required).

9.2.3.3 Operating system

An operating system is essential for the platform since it allows the management of memory, scheduling of tasks, interaction with hardware devices, handling of data with a file system, and support for a user-interface.

Among the existing embedded operating systems for MicroBlaze, uClinux constituted the best option for the platform. uClinux is a Linux fork for embedded systems lacking an

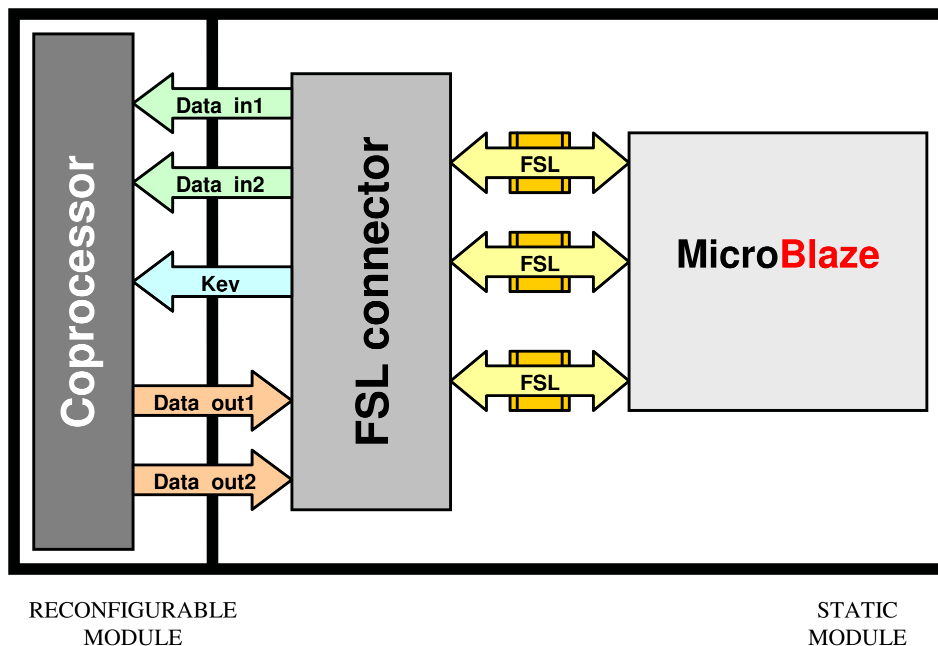


Figure 9.5 *FSL connections between MicroBlaze and the coprocessor.*

MMU (Memory Management Unit). The MicroBlaze open-source port is supported by Petalogix (founded by John Williams) [229]. uClinux provides robustness and reliability (over ten years of development and use in many devices), a configurable and scalable architecture, great support for networking, a large pool of skilled developers, and more.

9.2.3.4 Reconfiguration

Within the framework of reconfigurable pervasive systems, three possible reconfiguration scenarios are considered: *exo-configuration*, *endo-configuration* (also known as self-reconfiguration), and a *hybrid* approach that profits from the advantages of the first two techniques.

The *exo-configuration* constitutes the traditional way to configuring an FPGA. A configuration bitstream is generated by a host computer, and then it reprograms the FPGA (the bitstream can complete or partial). In this way, new coprocessors or upgraded versions, can be created and used at any moment. This approach exhibits upgradeability, but the platform is totally dependent of the host computer for modifying its function.

The *endo-reconfiguration* considers a different scenario. An FPGA reconfigures itself using its own local resources. The platform is thus totally independent, as it does not require an external source to provide a bitstream and to decide whether to self-reconfigure. The main draw-back is that partial bitstreams need to be previously generated by a host computer. This approach thus benefits from an autonomous reconfiguration with limited upgradeability.

With the aim of combining the advantages of both reconfiguration techniques, the *hybrid* platform presented here can perform self-reconfiguration using local bitstreams, but can also request other bitstreams from a remote server as shown in figure 5. The remote server provides

a full repository of bitstreams as it does not have the memory limitations of the pervasive platform. The server can also generate new bitstreams so the latest version is available to the platform.

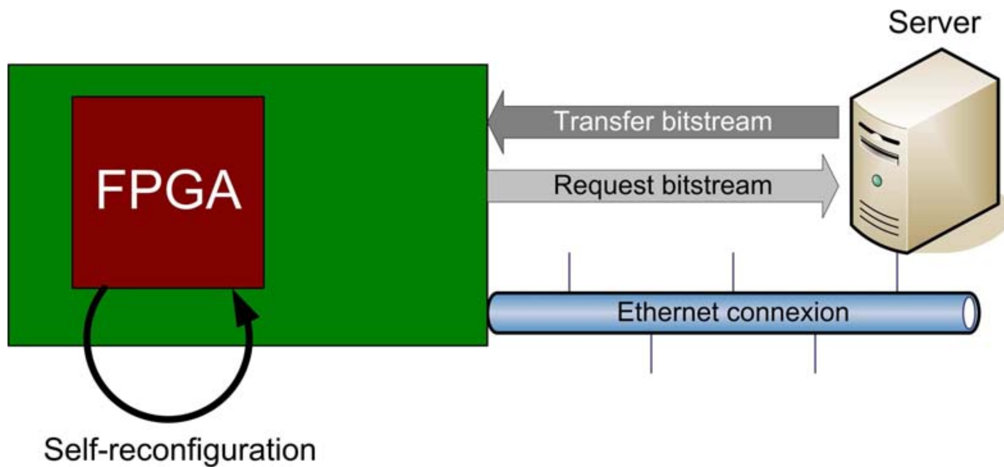


Figure 9.6 *Hybrid configuration platform.*

9.2.4 Experimental setup and results

The validity and efficiency of the platform was demonstrated by developing a benchmark to compare the performances of two platforms providing similar flexibility: a full-software solution running on MicroBlaze against a reconfigurable coprocessor implementation. The throughputs are measured by a benchmark running on uClinux and Xilkernel [239] (a small, robust and modular kernel which provides minimal essential services). Both the full-software and the coprocessor-based implementations process randomly generated data. For the Xilkernel environment there were used 512 KB of sample data and for uClinux there were used 256 KB.

The reconfiguration time was also measured: the time to request the bitstream from a distant server and the time to self-reconfigure the FPGA through the ICAP.

9.2.4.1 Hardware and software setup

The MicroBlaze processor was enabled with a barrel shifter, a divider, and a multiplier, running at a frequency of 27 Mhz. There were also considered the speed-ups obtained with and without activating the cache memory. The cache was not activated for uClinux because the system was not stable enough. The coprocessor is connected with 3 FSL links as explained in subsection 9.2.3.2. The processor setup features also an OPB bus (a standard peripheral bus) with several peripherals attached to it; the most important are the ICAP and the Ethernet controllers.

uClinux was parameterized with networking support, as well as networking tools such as FTP, and basic tools to manage an OS.

9.2.4.2 Logic resources

The available resources of the reconfigurable module (hosting the coprocessor) are 896 slices, while the static module (hosting the MicroBlaze) has an area of 9856 slices, leaving enough space for the different peripherals. The resource usage for the different designs is presented in table 9.1.

Table 9.1 *Logic resources.*

	Slices	FF/Latches Slices	Module utilization
MicroBlaze	3670	3372	35%
RC4	614	511	48%
DES	499	511	35%
Triple DES	815	654	70%

9.2.4.3 Hardware speedups under the Xilkernel

Table 9.2 shows the comparison of the benchmark throughputs for the different algorithms' implementations (full-software or coprocessor-based) and their corresponding speedups. RC4 clearly has the lowest speedup because this algorithm is better suited for software implementations. However, the coprocessor still has a valuable improvement. Finally, it must be also added that logically, the cache benefits the full-software implementation more.

Table 9.2 *Results under the Xilkernel [Mbps].*

	Algorithm	Throughput without cache	Throughput with cache
	RC4	0.95	3.94
Full Software	DES	0.24	1.19
	Triple DES	0.08	0.42
	RC4	3.22	8.44
Coprocessor based	DES	3.52	10.23
	Triple DES	2.09	5.64
	RC4	3.4	2.1
Speedup	DES	14.4	8.6
	Triple DES	24.8	13.4

9.2.4.4 Hardware speedups under uClinux

As shown in table 9.3, one can obtain lower speed-ups with uClinux than with the Xilkernel. Focusing on the throughputs, the full-software throughputs are almost the same as with the Xilkernel, while for the coprocessor-based they decrease. The gap between the two environments could be partially explained by the complexity of interrupt handling and memory management in uClinux. Nevertheless, the speedups are still promising.

Table 9.3 *Results under uClinux [Mbps].*

	Algorithm	Throughput without cache
Full Software	RC4	1.08
	DES	0.24
	Triple DES	0.08
Coprocessor based	RC4	2.10
	DES	1.12
	Triple DES	1.09
Speedup	RC4	1.9
	DES	4.6
	Triple DES	13.3

9.2.4.5 Reconfiguration time

To handle the bitstream, three scenarios are considered. The first one is exactly the endo-configuration described in 9.2.3.4: the bitstream is stored in an on-board memory so the downloading time is not necessary and the self-configuration can take place through the ICAP. The second and third scenarios are two examples of the hybrid reconfiguration described in 9.2.3.4. The second scenario is when the system rarely needs to update its coprocessor, so it will initiate a new connection to retrieve the file. A simple way to do so is to retrieve the file through the HTTP protocol. The third scenario is when the system has several coprocessors to download in a row. To remove the overhead of initiating one connection for each download, only one connection is set up. The more convenient way to do this is to open a persistent FTP connection and then retrieve the bitstreams successively. Table 9.4 shows, for each partial bitstream, the reconfiguration time for each scenario.

Table 9.4 *Reconfiguration time.*

	File size [bytes]	ICAP processing [ms]	HTTP download [ms]	FTP persistent download [ms]
RC4	70832	117	2385	1281
DES	62896	104	2379	1137
Triple DES	65116	107	2382	1178

These data allow one to determine if the usage of a reconfigurable coprocessor is justified in a real scenario. To do so, the minimum data size to be encrypted is computed, so that the self-reconfigurable platform performs better than a full-software solution, including the overhead due to the manipulation of the configuration bitstream. In a full-software implementation, switching from an algorithm A to an algorithm B does not imply any overhead. With the self-reconfigurable platform, the self-reconfiguration processing time must be added, and if required, the downloading time too. So, for having a better performance for a coprocessor-based than for a full-software computation, one must guarantee that:

$$t_A^{soft} + t_B^{soft} > t_A^{hard} + t_{download} + t_{reconfig} + t_B^{hard} \tag{9.1}$$

As it is not significant to compute the time required to switch between any two coprocessors, there are only considered the worst and best cases. The worst case for the coprocessor-based implementation is starting from RC4 and switching to DES in uClinux (these are the lowest speedups) and the best case is starting from DES and switching to Triple-DES in the Xilkernel without cache.

In this way, one can extract the minimum data size by replacing the variables in equation 9.1 (knowing that the elapsed time for encryption is the file size divided by the throughput):

$$Min_{size} = \frac{t_{download} + t_{reconfig}}{\left(\frac{1}{Tp_A^{soft}} + \frac{1}{Tp_B^{soft}} - \frac{1}{Tp_A^{hard}} - \frac{1}{Tp_B^{hard}}\right)} \tag{9.2}$$

The results for the best and worst cases in the three scenarios are exhibited in table 9.5. In cryptography, one generally finds sizes larger than 100 kilobytes. So, the platform using self-reconfiguration has been shown to always perform better than a classical software implementation, even if storage in a remote server is needed.

Table 9.5 *Minimum data size [kb].*

	Local reconfiguration	**HTTP download and reconfiguration**	**FTP persistent download and reconfiguration**
Best case	0.84	19.56	10.10
Worst case	3.49	83.35	41.66

9.3 On-line self-reconfigurable system for adaptive channel equalization

Pervasive systems are often omnipresent mobile systems with seamless communication requirements. Wireless communications are vulnerable to errors, given perturbations and low signal integrity because of the communication channel. Two complementary techniques are often used for minimizing communication errors: software solutions using error-correction codes and hardware solutions implementing channel equalizers. The mobile nature of pervasive systems implies a changing communication channel model, for which on-line adaptive channel equalizers constitute a promising solution.

This section presents an hybrid bio-inspired optimization technique that introduces the concept of discrete recombination in a particle swarm optimizer, obtaining a simple and powerful algorithm, well suited for embedded applications. The algorithm is validated using standard benchmark functions, and is used for training a neural network-based adaptive equalizer for communications systems.

On-line and on-chip adaptation in self-reconfigurable hardware systems provide architectural flexibility, allowing the chip to adapt dynamically and autonomously to changes in its environment [223, 247]. A popular approach for building adaptive circuits is by means of bio-inspired techniques. EHW tackles this problem by using EAs: inspired by the process of

natural evolution, a population of circuits is incrementally improved through the application of genetic operators (selection, crossover, and mutation).

From an algorithmic point of view, EAs are *stochastic population-based* optimization techniques. They are population-based because, contrary to other global optimization techniques like branch-and-bound [153], they keep a population or a set of solutions in memory instead of sweeping the whole search space. The algorithm produces populations of solutions sequentially from an initial population P_0 to a final population P_G, deriving the new population from the current population through the use of a *manipulation* function [64]: $P_t = m(P_{t-1}, f(P_{t-1}))$, where P_t is the population at time step $t = 1, 2, ...G$, $f(\cdot)$ is the fitness function that returns a vector of fitness values, and P_0 is randomly initialized. Finally, EAs are said to be stochastic because selection, recombination and mutation operators always include randomness, giving as a result a non deterministic manipulation function.

Particle swarm optimizers constitute another group of optimization algorithms, beyond the EAs family, which are said to be *stochastic population-based*. Particle Swarm Optimization (PSO), already introduced in subsection 3.2.3, is a bio-inspired technique founded on the social behavior of bird flocking and the idea of culture as an emergent process [87]. In PSO, a swarm or population of solutions "fly" through the search space according to certain velocity update rules, producing new sets of solutions in subsequents time steps. The population of solutions is thus "evolved" through the application of a certain manipulation function, as EAs do.

Given the similarities between the two optimization approaches, it is natural to think about proposing an alternative solution to EAs that carries out adaptation in EHW by a particle swarm optimizer. Some preliminary steps have already been taken in this direction. PSO has been used, for instance, in the context of evolutionary circuit design [25,64] and in the problem of placement and routing in Xilinx FPGAs [49, 137, 225]. However, PSO has not already been used for *on-line on-chip* hardware adaptation or evolution.

When intending to implement *on-line* and *on-chip* hardware adaptation, one must consider the computational complexity of the involved search algorithms. This section presents a simple, hybrid algorithm that takes the concept of recombination of EAs to incorporate it in the original scheme of PSO. The proposed algorithm is *hardware friendly*, making it suitable for efficient implementation in either an embedded processor or parallel hardware: it does not use multiplications and requires a minimal random number generator (RNG).

The proposed algorithm has been conceived targeting the adaptation of a society of agents embedded in a self-reconfigurable adaptive platform. Each of these agents can be, for instance, a channel equalizer in a communication system. The hardware setup considers a population of neural networks with material existence in an FPGA, being evaluated and adapted on-line by the proposed algorithm running in an embedded microprocessor. The general approach used in existing solutions to channel equalization is to have one single equalizer working at the time, being adapted through supervised learning [151] or a GA [141, 143].

The proposed approach is bio-inspired at two levels: at the *computation engine* level and at the *adaptation mechanism* level. The computation engine constitutes the problem solver of the system: the problem at hand, being in this case the channel equalizer, is implemented as the ANN of the type *Binary Radial Basis Functions*, that will be described in the sub-section 9.3.2. The adaptation mechanism provides the possibility to modify the function described by the computation engine. This adaptation is performed by the Particle Swarm Optimizer with

Discrete Recombination described in the sub-section 9.3.1.

In this section, the proposed algorithm is compared against the standard approach in a rather theoretical experimental setting involving the minimization of four mathematical functions. Then, there are presented some preliminary results in the use of the method for evolving simple neural networks with binary activation functions for channel equalization.

9.3.1 Particle swarm optimization with discrete recombination

The standard PSO intruduced in subsection 3.2.3, has a number of features that make it suitable for embedded applications. It is simple enough to be implemented in software to be run in a microprocessor, or to be implemented directly in hardware. Still, it requires the performing of 3 multiplications and the generation of $2B$ random bits per particle and per dimension, where B is the bit resolution of φ_1 and φ_2. Thus, a total amount of $M \times n \times 3$ multiplications and $M \times n \times 2 \times B$ randomly generated bits are required per iteration of the algorithm. For some critical applications, this can be prohibitive in terms of area, power consumption and/or performance.

PSODR was designed bearing in mind a complete avoidance of multiplications and the reduction of the number of randomly generated bits to a minimum. The proposed model incorporates the notion of discrete recombination as used in the field of Evolution Strategies [12], to the personal best vectors, proposing novel velocity update rules and a blending of *lbest* and *gbest* topological models.

The idea is to consider *lbest* neighborhoods without the self, so that the neighborhood of a given particle j is comprised only of the left and right neighbors in the circular array:

$$N(j) = \{left(j), right(j)\} \tag{9.3}$$

Within these neighborhoods, a recombinant $\mathbf{r}_j^t = (r_{j1}, r_{j2}, \ldots, r_{jn})$ is generated by coordinate-wise random selection from the corresponding coordinate values of the neighbors:

$$r_{id} = \begin{cases} p_{left(j)i} & \text{if } RAND() = 0 \\ p_{right(j)i} & \text{otherwise} \end{cases} \tag{9.4}$$

where $RAND()$ is a 1-bit (0 or 1) random number.

One can think of at least two modifications to the *inertia weight* update rule of Eq. 3.2 using this *recombinant target*. The first one replaces the neighborhood best by the recombinant and keeps the personal best:

$$\mathbf{v}_j^{t+1} = w \cdot \mathbf{v}_j^t + \varphi_1 \cdot (\mathbf{p}_j^t - \mathbf{x}_j^t) + \varphi_2 \cdot (\mathbf{r}_j^t - \mathbf{x}_j^t). \tag{9.5}$$

The second one replaces the personal best by the recombinant while keeping the neighborhood best:

$$\mathbf{v}_j^{t+1} = w \cdot \mathbf{v}_j^t + \varphi_1 \cdot (\mathbf{r}_j^t - \mathbf{x}_j^t) + \varphi_2 \cdot (\mathbf{p}_{b(j)}^t - \mathbf{x}_j^t). \tag{9.6}$$

Notice the replacement of the random variables $\mathbf{U}[0, \varphi_1]$ and $\mathbf{U}[0, \varphi_2]$ of the original algorithm in equation 3.2 by the fixed constants φ_1 and φ_2. This fact allows an important

simplification of the necessary RNG, given that only $M \times n$ bits need to be randomly generated per iteration (to produce the recombinants).

Taking into account the typical choices of $\varphi_1 = \varphi_2 = 2$ in the standard PSO and the fact that the expected values of $\mathbf{U}[0,2]$ and $\mathbf{U}[0,2]$ are 1, $\varphi_1 = \varphi_2 = 1$ reveals as a natural choice for the update rules of Eq. 9.5 and Eq. 9.6. This choice eliminates two of the multiplications required in the original algorithm. If, in addition to that, a constant inertia weight of 0.5 is taken, PSODR can be implemented without the need of any multiplier: a multiplication by 0.5 is just a right shift operation.

Two PSODR models are considered: the *lbest* model and the *gbest* model. The first one always uses a *lbest* topology and a velocity update rule given by Eq. 9.5. The second one uses a *lbest* topology to calculate the recombinant $\mathbf{r}_j^t$, but a *gbest* topology to calculate the neighborhood best . The pseudocode of the algorithm is shown in the Alg. 3.

Algorithm 3 PSODR

procedure PSODR(METHOD)
  Initialize positions, velocities and personal bests
  repeat
    for each particle j in the population **do**
      if $f(\mathbf{x}_j) < f(\mathbf{p}_j)$ **then**
        $\mathbf{p}_j = \mathbf{x}_j$
        if method is *gbest* **then**
          if $f(\mathbf{p}_j) < f(\mathbf{p}_g)$ **then**
            $g = j$
          end if
        end if
      end if
      for each dimension i **do**
        $r = RAND()$
        if $r = 0$ **then**
          $k = left(j)$
        else
          $k = right(j)$
        end if
        if method is *gbest* **then**
          $v_{ji} = w \cdot v_{ji} + (p_{ki} - x_{ji}) + (p_{gi} - x_{ji})$
        else(method is *lbest*)
          $v_{id} = w \cdot v_{ji} + (p_{ki} - x_{ji}) + (p_{ji} - x_{ji})$
        end if
        $v_{ji} \in (-V_{max}, V_{max})$
        $x_{ji} = x_{ji} + v_{ji}$
      end for
    end for
  until termination condition is reached
end procedure

9.3.2 Binary radial basis functions

ANNs are structures of densely interconnected neurons, each of which receives an input vector and process it by passing its inner product with a *weight vector* through an *activation* or *transfer function* [60]. In practice, ANNs allow to efficiently design any function by setting the correct parameters (weights). This efficiency is provided thanks to their cellular architecture and the possibility of applying optimization algorithms (learning or evolution) for finding the correct set of parameters. By selecting the correct weights, with PSODR for instance, an ANN can be used as channel equalizer.

Given their cellular nature, ANNs lend themselves to hardware implementation. However, one face several practical problems when implementing them in hardware. Most neuron models, such as perceptron or radial basis functions, use logistics, gaussians or other continuous functions as transfer functions. Additionally, each network connection (synapse) requires a multiplier for weighting the inputs to each neuron. Hardware implementation of these functions result very expensive in terms of logic resources. An example showing the complexity of such hardware systems is, for instance, the GRD chip, in which each neuron of the net is implemented in a DSP [143]. That's the reason why one find in the literature several approaches to simplified hardware-oriented neuron and network models [136].

In this framework, there is a special interest in simplistic low-cost ANNs, so that an entire population of them could be easily embedded in a commercial FPGA. Given that no supervised learning is intended, the constraint of having differential activation functions can be eliminated, and binary activation functions can be considered. One such neural net with binary activation functions is, for instance, a feed-forward neural network with Heaviside function. Training this kind of nets can be time consuming, so for this implementation it will be considered instead the *binary radial basis function* (BRBF) that is described in the following lines.

In a radial basis function (RBF) net the inputs are fed into a single layer of neurons, whose outputs feed an output adder. The j-th neuron calculates a response $\phi_j(\mathbf{x})$ based on the distance between the center $\mathbf{c}_j$ of its receptive field and the input vector $\mathbf{x}$. The closer the input vector to the center of the receptive field, the higher the activation of the neuron. Standard RBFs use gaussian activation functions, so that the output of the j-th neuron is given by

$$\phi(\mathbf{x}) = \exp\left(-\frac{D(\mathbf{x}, \mathbf{c}_j)}{2\sigma_j^2}\right) \tag{9.7}$$

where $D(\mathbf{x}, \mathbf{c}_j)$ is the euclidean distance between $\mathbf{x}$ and $\mathbf{c}_j$, and σ_j is the width of the receptive field.

The output y of the net is typically given by

$$y = \sum_{j=1}^{J} \phi_j(\mathbf{x}) \cdot w_j, \tag{9.8}$$

where w_j is the weight of the output connection corresponding to the j-th neuron, and J is the total number of neurons in the net. A RBF net as the one previously described is bad suited for resource-optimal hardware implementation, because of the required multiplications and the necessity of calculating an exponential.

To have a truly hardware friendly neural net, a BRBF net is proposed, in which the activation function is described by:

$$\phi_j(\mathbf{x}) = \begin{cases} 1 & \text{if } D(\mathbf{x}, \mathbf{c}_j) \leq \sigma_j \\ 0 & \text{otherwise} \end{cases} \tag{9.9}$$

and the distance function $D(\mathbf{x}, \mathbf{c}_j)$ is the Manhattan distance function:

$$D(\mathbf{x}, \mathbf{c}_j) = \sum_i |x_i - c_{ji}| \tag{9.10}$$

With these definitions, the BRBF net is, basically, a *linear approximator with binary features* that does not require any multiplication nor in the distance calculation nor in the output part (the multiplication by 1 or 0 is trivial) and can thus be easily implemented in hardware. A schematic view of the proposed ANN is shown in the figure 9.7.

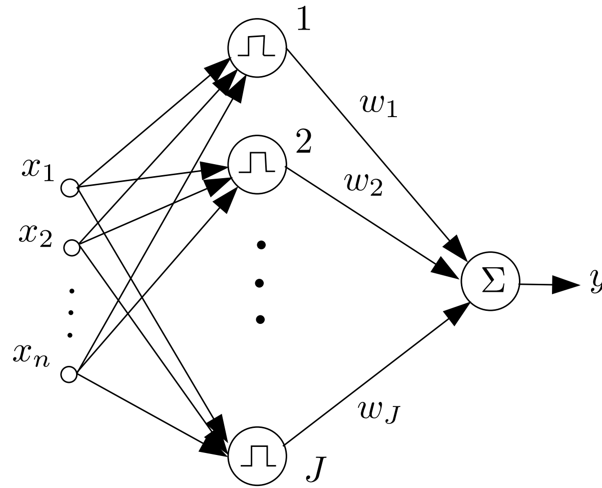


Figure 9.7 *Binary radial basis function*

9.3.3 Experimental settings and results

In order to verify the computational capabilities of the proposed PSODR and BRBF, two sets of experiments were devised. First, the two variants (*lbest* and *gbest*) of the proposed PSODR model were tested against their standard PSO counterparts in four benchmark minimization problems. Then, a population of BRBF neural nets was evolved using one variant of PSODR to solve the problem of static channel equalization in a simple communication system model.

9.3.3.1 Benchmark function minimization

Experimental Settings

Four functions were used in this experiment. The first function is the (generalized) Sphere function, described by

$$f_1(\mathbf{x}) = \sum_{i=1}^{n} x_i^2. \tag{9.11}$$

The second function is the (generalized) Rosenbrock function, given by the equation:

$$f_2(\mathbf{x}) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2. \tag{9.12}$$

The third test function is the (generalized) Rastrigin function:

$$f_3(\mathbf{x}) = \sum_{i=1}^{n} x_i^2 + 10[1 - \cos(2\pi x_i)]. \tag{9.13}$$

The fourth test function is the (generalized) Griewank function:

$$f_4(\mathbf{x}) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1. \tag{9.14}$$

For all the four functions, the minimum value is 0.

These functions have been widely used in the literature to test different EA and PSO-based algorithms (see, for instance, [5]). The Sphere function is an easy, unimodal function that any optimization technique should be able to solve with a good degree of resolution and that helps to identify good local optimizers. The Rosenbrock function is also unimodal, but generally difficult to optimize even for gradient-based algorithms. The last two functions are multimodal, having many local minima and help to test the global optimization capabilities of the tested algorithms.

The proposed *gbest* and *lbest* versions of the PSODR algorithm were tested against their standard counterparts, using swarms of 20, 40 and 80 particles and function of 10, 20 and 30 dimensions. For the four algorithms an inertia weight of 0.5 was used. For the two standard models the acceleration constants were set to $\varphi_1 = \varphi_2 = 2$, while for the two PSODR models values of $\varphi_1 = \varphi_2 = 1$ were used. The maximum number of iterations was set to 1000, 1500 and 2000, which correspond to dimension sizes of 10, 20 and 30 respectively.

The positions of the particles were initialized according to the asymmetric initialization method proposed in [5]. Table 9.6 shows the the initialization ranges and the values of v_{max} for each function.

Table 9.6 *Initialization ranges and* v_{max} *for each function*

Function	**Initialization Range**	v_{max}
Sphere	$(-100, 100)^n$	100
Rosenbrock	$(-100, 100)^n$	100
Griewank	$(-600, 600)^n$	10
Rastrigin	$(-5.12, 5.12)^n$	600

Results and Discussion

Tables 9.7, 9.8, 9.9 and 9.10 list the values of the best solution found at the final iteration by each method. The following nomenclature was used: SPSOG (standard *gbest* PSO), SPSOL (standard *lbest* PSO without the self), PSODRG (*gbest* PSODR) and PSODRL (*lbest* PSODR). The results are averaged over 50 independent runs.

Table 9.7 *Mean fitness values for the Sphere function*

M	D	SPSOG	SPSOL	PSODRG	PSODRL
20	10	9.03E-39	6.04E-17	**1.18E-51**	2.77E-38
	20	6.07E-24	1.69E-9	**1.95E-44**	3.06E-30
	30	2.24E-18	1.69E-6	**1.72E-39**	4.65E-27
40	10	2.61E-46	5.75E-18	**1.16E-54**	1.44E-38
	20	3.69E-30	6.95E-10	**1.54E-49**	1.62E-30
	30	1.37E-23	8.96E-7	**5.82E-48**	3.40E-27
80	10	2.98E-52	2.93E-18	**6.38E-57**	6.52E-39
	20	1.22E-36	3.97E-10	**1.83E-52**	9.56E-31
	30	1.94E-28	4.59E-7	**1.35E-52**	2.13E-27

Table 9.8 *Mean fitness values for the Rosenbrock function*

M	D	SPSOG	SPSOL	PSODRG	PSODRL
20	10	31.14	12.70	17.75	**9.11**
	20	80.62	73.84	**28.78**	30.42
	30	157.90	163.18	**59.05**	88.24
40	10	18.78	6.42	**3.79**	4.44
	20	61.32	27.18	**16.68**	23.68
	30	80.04	74.68	**40.84**	60.30
80	10	10.40	2.02	**1.65**	1.93
	20	80.82	14.37	**3.26**	14.53
	30	76.98	63.82	**19.42**	39.86

As it can be seen, one of the two proposed methods always performed better than the standard methods for the two unimodal functions. The best method was PSODRG, obtaining the minimum values in all the cases, except for Rosenbrock with $M = 20$ and $D = 10$, where it was beaten by PSODRL. The proposed *lbest* method also performed better than the standard *lbest* model.

For the multimodal functions, the best proposed method performed better than the best standard method in 11 of the 18 cases. In the Rastrigin function, PSODRG was the best when dealing with low dimensionalities of the problem, but the SPSOG performed better in higher dimensionalities. PSODRL was always outperformed by SPSOL. For the Griewank function, the *lbest* models performed better than the *gbest* models. Here, the standard version was better

Table 9.9 *Mean fitness values for the Rastrigin function*

M	D	SPSOG	SPSOL	PSODRG	PSODRL
20	10	8.26	8.68	**7.34**	15.22
	20	**35.26**	37.24	39.96	75.22
	30	**77.55**	76.43	98.90	148.95
40	10	4.22	5.63	**2.99**	13.63
	20	22.76	30.16	**22.60**	68.52
	30	**48.20**	64.84	54.66	144.14
80	10	3.02	4.70	**1.93**	11.54
	20	15.90	26.94	**11.57**	60.80
	30	**34.50**	59.42	35.26	132.58

Table 9.10 *Mean fitness values for the Griewank function*

M	D	SPSOG	SPSOL	PSODRG	PSODRL
20	10	7.54E-2	**4.51E-2**	7.16E-2	1.00E-1
	20	2.28E-2	6.50E-3	1.26E-2	**4.96E-3**
	30	1.32E-2	5.05E-3	1.29E-2	**1.84E-3**
40	10	7.60E-2	**3.61E-2**	5.69E-2	7.35E-2
	20	2.25E-2	2.42E-3	1.55E-2	**7.94E-4**
	30	1.69E-2	1.82E-3	5.71E-3	**2.84E-4**
80	10	6.95E-2	**2.12E-2**	4.29E-2	4.02E-2
	20	2.66E-2	4.12E-4	1.35E-2	**6.69E-5**
	30	1.49E-2	5.52E-4	5.02E-3	**7.75E-7**

when dealing with dimension sizes of 10, while the proposed version was better when dealing with dimension sizes of 20 and 30. PSODRG performed better than SPSOG in all the cases for this function.

The proposed *gbest* PSODR performed better than its standard counterpart always for the Sphere, Rosenbrock and Griewank functions and in 5 out of 9 cases in the Rastrigin function (low dimensionality of the search space). On the other hand, the proposed *lbest* PSODR performed better than its standard counterpart always for the Sphere function, in the majority of the cases (10 out of 11) for the Rosenbrock function and the Griewank function (6 out of 9), but worst in all the cases for the Rastrigin function.

Figures 9.8, 9.9, 9.10 and 9.11 show the learning performance of the different algorithms for the four tested functions with $M = 20$ and $D = 10$. The curves represent the fitness of the best solution ($f(\mathbf{p}_g^t)$) averaged over 50 runs. Observe how the PSODR models are characterized by a faster convergence than the SPSO models (except for Rastrigin function and the *lbest* case, in which SPSOL converges faster than PSODRL). In particular, the PSODRG model converges faster than any of the other methods. It can also be easily seen that the *lbest* models converge slower than the *gbest* methods. This slower convergence does not pay off in the unimodal functions, but it allows the discovery of better solutions in one of the multimodal functions (Griewank).

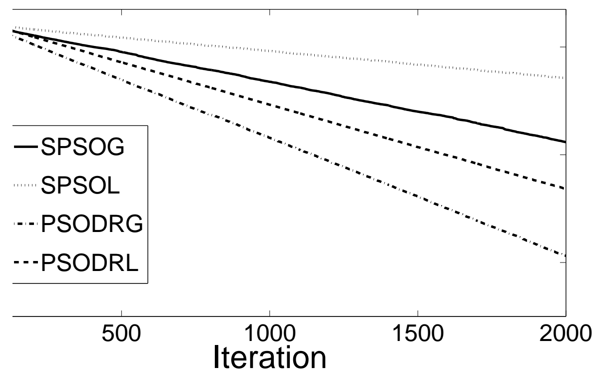


Figure 9.8 *Learning performance for the Sphere function with $M = 20$ and $D = 30$*

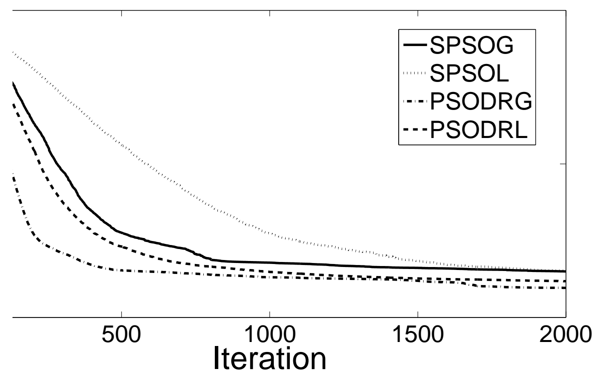


Figure 9.9 *Learning performance for the Rosenbrock function with $M = 20$ and $D = 30$*

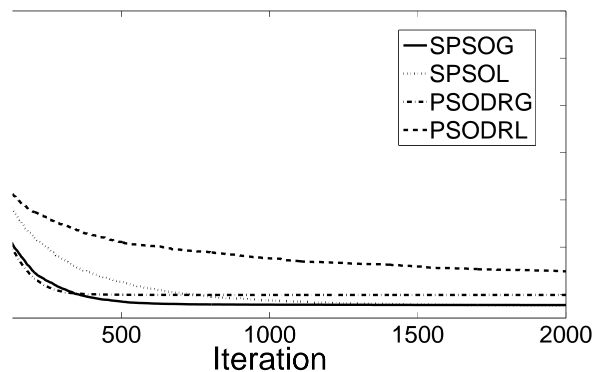


Figure 9.10 *Learning performance for the Rastrigin function with $M = 20$ and $D = 30$*

9.3.3.2 Channel equalization of a digital communications system

In a digital communication system, a series of symbols $s(t)$ is generated in a source and transmitted over a channel to a receiver. In practice, the channel is not ideal and data is corrupted with nonlinear distortion, intersymbolic interference (ISI) and noise.

One way to alleviate these problems and obtain reliable data transmission is to use a

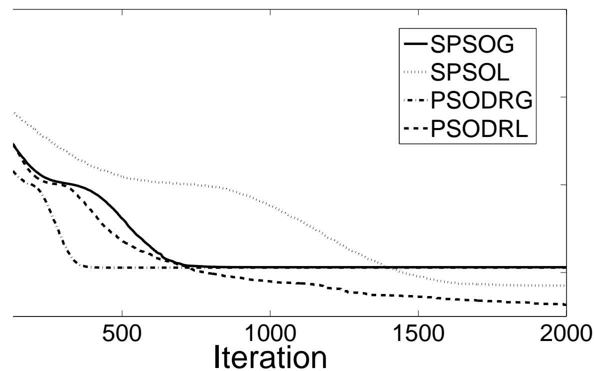


Figure 9.11 *Learning performance for the Griewank function with $M = 20$ and $D = 30$*

channel equalizer in the receiver [159]. The task of the equalizer is to reconstruct the original signal $s(t)$ from the received signal $r(t)$ or, in other words, to generate a reconstructed version $\hat{s}(t)$ of $s(t)$ as close as possible to it (Fig. 9.12). The addition of an equalizer usually reduces the bit error rate (BER): the ratio of the received bits in error to the total of bits transmitted.

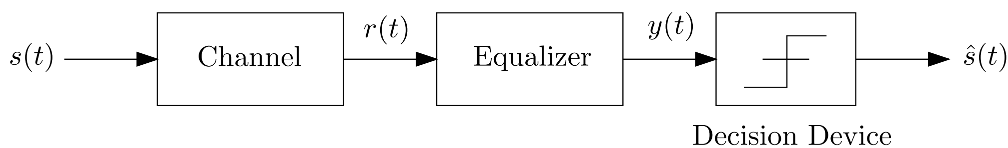


Figure 9.12 *Communication system model*

Traditional adaptive equalization relies on the use of a linear transversal filter. This filter is generally adjusted using a known training sequence at the beginning of the transmission and LSE or gradient descent to determine the optimal set of coefficients for the filter. However, when nonlinear distorsion and intersymbolic interference are severe, nonlinear equalizers such as neural nets can give a better performance [151]. Nonetheless, the training of these structures generally involves the use of backpropagation or other related, supervised techniques which are generally computational expensive.

In [143], Murakawa et al presented the GRD chip and used it for adaptive channel equalization. The GRD chip is a group of 15 DSPs connected in a binary-tree network that implement a feed forward neural network. The net is reconfigured and trained by a GA and steepest gradient descent running in an embedded RISC processor. The population of solutions do not have material existence: only one physical net is implemented by the tree network of DSPs, with each individual being downloaded for evaluation. The proposed solution presents two main drawbacks. First, even though a GA is used, the type of learning is essentially supervised, needing a training sequence to be transmitted from the source. Second, the solution is rather expensive, since a single neuron is implemented in a dedicated DSP.

At present, commercially available FPGAs benefit of large amounts of configurable resources, allowing the implementation of very complex circuits. In this approach, it will be considered the implementation of a whole population of simplistic neural networks (e.g, the of the type of the the proposed BRBF net) in an FPGA along with an embedded soft-processor which

would be responsible of running the self-adaptation mechanism (e.g, the proposed PSODR) and the reconfiguration of the population. The setup of the complete system can consist in a self-reconfigurable platform as the one described in sections 6.3 and 7.2.

Based in hardware synthesis reports, a single 15-neuron BRBF-network with data resolution of 8 bits, implemented in the Virtex-II FPGA 2v4000 from Xilinx, requires 420 slices (2% of the FPGA's logic resources). Therefore, it is reasonable to imagine a self-reconfigurable platform with a MicroBlaze soft-processor reconfiguring a population of until 30 BRBF networks embedded in a single 2v4000 FPGA.

This platform will be used for the solution of adaptive channel equalization. In order to do this without the need of a training sequence, the BER of each neural network-based equalizer will be estimated by means of an error detection code. Using these measures, the PSODR adapts the different parameters of the nets in the population, finding incrementally a good solution in the search space, and decreasing the actual BER of the whole system. The best solution found so far will always be physically present, giving the actual output of the equalizer (Fig. 9.13).

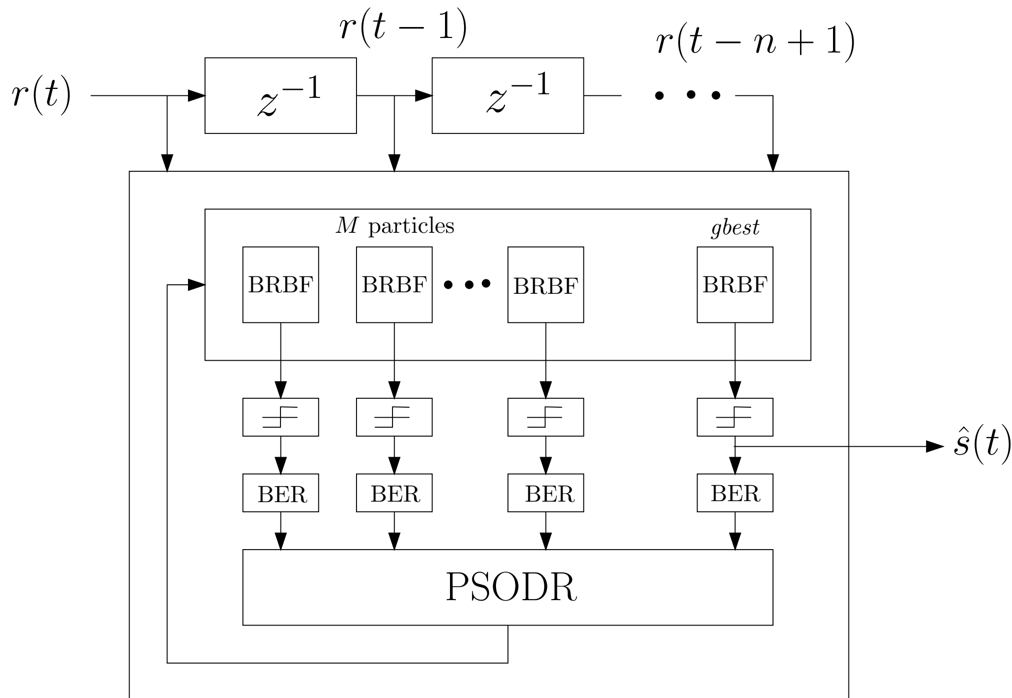


Figure 9.13 *The proposed equalizer*

For the sake of comparison, the communication system proposed in [143] is used. The source transmits a randomly generated sequence of bipolar symbols (-1 and $+1$) through a linear channel with additive, zero-mean Gaussian noise. The transfer function of the channel is $H(z) = 1 + 1.5z^{-1}$. The order of the equalizer (number of delay elements at the input of the equalizer) was thus set to $n = 2$ (see Fig. 9.13).

Using this setup, populations of 10, 20 and 30 BRBF nets with $J = 15$ neurons each were evolved using the *gbest* version of PSODR with $w = 0.5$ and $\varphi_1 = \varphi_2 = 1$. The PSODR algorithm was responsible for adapting not only the output weights w_j of the net, but also the

centers $\mathbf{c}_j$ and the widths σ_j of the neurons: a $2J + NJ = 60$-dimensional search space.

A generation comprises the reception of 10^4 symbols by every BRBF net. For the sake of simplicity, an ideal BER estimator was assumed, estimating the BER of each particle as the ratio of misclassified to total number of symbols in the output of the decision device. The decision device uses the threshold function:

$$\hat{s}(t) = \begin{cases} -1 & \text{if } y(t) < 0 \\ 1 & \text{otherwise} \end{cases} \tag{9.15}$$

The learning performance of this simulation for a signal-to-noise-ratio (SNR) of 15dB is shown in the Fig. 9.14. For each population size, each curve shows the measured BER of the *gbest* solution for each generation, averaged over 100 independent runs. As it can be seen, the BER is improved over the generations showing a satisfactory learning process. When compared with the results given in [143] for this experiment, it can be seen that the final average BER obtained by the proposed solution is much lower than the one provided by a traditional linear transversal filter and that of the GRD system. For the population size of 10, the improvement is 5 times, whereas for 20 and 30 particles the improvement is about 2 orders of magnitude. This is a significant result, specially when comparing the population sizes (80 in the referenced work) and the computational complexity of the two approaches. Given the similar results obtained by population sizes of 20 and 30, it can be determined that 20 is an optimal size (within the chosen set of values).

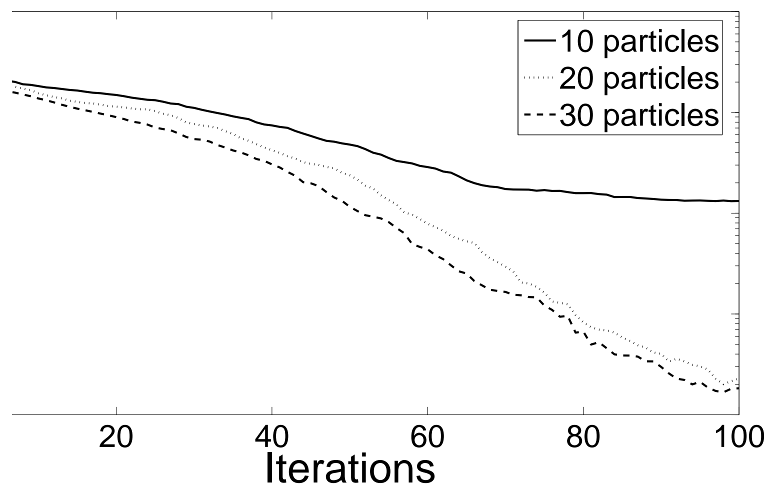


Figure 9.14 *Learning performance of the proposed equalizer for a SNR of 15 dB*

9.4 Conclusions

Pervasive reconfigurable systems are an application field well suited to bio-inspired reconfigurable systems. Their mobile nature requires them to be flexible enough to adapt to new and unknown environments. In this chapter, ROPES (Reconfigurable Object for Pervasive

Systems) has been presented, along with two application implementations targeting this prototyping board: a reconfigurable cryptographic application and an adaptive channel equalizer.

The presented self-reconfigurable cryptographic platform combines the flexibility of software and the high-performance of hardware, as shown in the results. The ability of the platform to communicate allows it to retrieve other bitstreams from other devices using Bluetooth, or to download bitstreams from a server able to generate new coprocessors. This feature permits a virtually infinite number of coprocessor configurations, allowing high performance for a wide array of applications.

Currently, the presented platform supports bitstream sharing through ethernet communication. However, it can easily be setup to benefit from Bluetooth wireless communications. One can envision, in this way, pervasive systems consisting of a population of ROPES accessing a common server or sharing coprocessors through a wireless channel.

Even though the cryptographic coprocessor exists as a pre-defined module not exhibiting any kind of adaptation, it is important to set up a whole system running an OS, able to self-reconfigure, and capable of exchanging configuration bitstreams with other units. These features are important for pervasive adaptive systems, since they allow a device to take advantage of the knowledge acquired by other devices by sharing, for instance, a network topology performing well in a certain environment.

Then, as an example, section 9.3 presents an adaptive system for a problem inherent to mobile wireless-communicating systems: an adaptive channel equalizer. As the adaptation mechanism PSODR has been proposed, a simple and efficient model for stochastic optimization that takes the concept of recombination from the evolutionary computation field and incorporates it into the general framework of particle swarm optimizers. When tested in benchmark optimization problems, the *gbest* and *lbest* PSODR variants show a better performance than the standard PSO algorithms. Most importantly, this improvement is not achieved by computationally complicating the algorithm, but by making it simpler.

The proposed computation engine consists of a population of very simple neural nets with material existence, which can be evaluated and reconfigured by means of a coprocessor running the proposed optimization algorithm. The population of nets along with the proposed optimization, have shown to be well suited to cope with the problem of channel equalization in a communication system for both the stationary and the non-stationary case.

Pervasive systems thus constitute, a promising application field for bio-inspired reconfigurable hardware systems. The high performance and high flexibility requirements of future pervasive systems will benefit greatly from the performance and flexibility features offered by current FPGA architectures. Additionally, the changing and unpredictable environment to which they are exposed will certainly allow them to exploit the adaptability provided by bio-inspired systems.

Chapter 10

Conclusions and Future Work

> Man must evolve for all human conflict a method which rejects revenge, aggression, and retaliation.
>
> Martin Luther King, Jr.

10.1 Summary

In this thesis, I have proposed several methodologies and techniques for the conception of bio-inspired self-reconfigurable systems by exploiting the current commercial FPGA's dynamic partial reconfigurability.

The methodology proposed in this thesis consists of two main components: a computation engine and an adaptation mechanism. The computation engine constitutes the bio-inspired architecture computing the solution. As examples, this thesis presented architectures for neural networks, spiking neurons, fuzzy systems, cellular automata, and random boolean networks.

The adaptation mechanism, on the other hand, allows the computation engine to adapt for performing a given computation; two basic types of adaptation have been proposed: parametric and structural. Parametric adaptation mainly deals with adjustment of functional parameters, while structural adaptation deals with topological modifications. The adaptation mechanism is basically presented in the form of evolutionary and learning algorithms.

The proposed system, along with the proposed bio-inspired architectures and algorithms, allows a system-on-chip to self-reconfigure in order to adapt the hardware supporting it, in a completely autonomous way. The main advantage of the proposed approach is that, unlike existing self-reconfigurable platforms, one is not required to specify every possible architecture to be implemented at design time, but it is the platform itself which determines it.

10.2 Original contributions

This thesis makes several contributions in the fields of bio-inspired architectures and dynamic self-reconfigurable systems. These contributions include methodologies, architectures, and design of prototyping platforms for potential application fields. Some of the original contributions are:

- A general framework for evolving hardware by partially reconfiguring Xilinx FPGAs.
- Three techniques for evolving hardware by using the two design flows proposed by Xilinx for partial reconfiguration (module-based and difference-based) and the direct bitstream manipulation flow.
- *Direct bitstream manipulation*, a new design flow for partially reconfigurable systems in Xilinx FPGAs, presenting several advantages over the existing design flows:
  - The generated bitstream is independent on Xilinx design tools.
  - The configuration bitstream can be generated on-line and on-chip with a low-cost processor.
  - Thanks to the low-level specification, the bitstream generation takes considerably less time than conventional design flows.
- A section of the configuration bitstream format of a Virtex-II, describing the configuration of LUTs and some multiplexers.
- A compact and performant architecture for a spiking neuron model with hebbian learning, and the characterization of the computational power of a network of them.
- A hardware implementation of the coevolutionary fuzzy system design technique Fuzzy Coco, where each one of the evolved species is independently reconfigured.
- A reconfigurable matrix array supporting random topological configurations, useful for digital hardware implementations of randomly connected networks, such as random boolean networks, echo state machines, liquid state machines, or for evolving networks with arbitrary connectionism.
- A new hardware-oriented PSO algorithm, which performs better than conventional PSO, and whose utilization has been tested in a channel equalization task.
- During this thesis I co-supervised the design of a new modular robot platform -YaMoR- whose most distinctive feature is the inclusion of an FPGA board and a Bluetooth board in each module.
- A framework for implementing partially reconfigurable controllers on the YaMoR platform.
- During this thesis I co-supervised the design of a prototyping platform for reconfigurable pervasive systems, along with a system setup for implementing a reconfigurable cryptographic coprocessor.

10.3 Future work

This thesis aims to help filling the gap between current bio-inspired computing techniques and their respective implementation in current commercial devices. To fulfill these objectives, this thesis involves several topics, and consequently, they generate a diversity of future research lines that may be followed in an independent or a complementary manner.

10.3.1 Hardware substrate

It is clear that commercial FPGAs are not the perfect platform for the type of systems presented in this thesis. Their general purpose architecture aims to be used for supporting applications other than bio-inspired cellular architectures. However, future trends in commercial FPGA architectures will certainly improve the features required for the type of systems described in this thesis.

Power consumption is one of the more critical issues. If we want a device to be autonomous for self-reconfiguring with the purpose of adapting to a changing environment, we can imagine that this device must be mobile, and consequently must be powered by a battery. Applications constrained by power consumption such as modular robots, pervasive systems, and wireless sensor networks are the first candidates for using self-reconfigurable adaptive technologies.

FPGA manufacturers are making significant efforts in this direction. Xilinx is investing in power saving technologies. For instance, the Spartan-3L family uses an internal power supply of 1.2 volts (instead of 1.5 volts, as other families) and provides a hibernate mode. The recent Virtex-5 family, built with a technology of 65nm and a core voltage of 1.0 volt, achieves power saving of around 35% with respect to Virtex-4 FPGAs, and has more embedded IPs. In the same way, Altera is also working hard at reducing the power consumption of their FPGAs. By using a 90nm technology, they are working on decreasing core voltage, increasing threshold voltage, increasing transistor length, and lowering pin capacitance. In addition, they are also exploring for more efficient clocking structures.

Concerning reconfigurability features, Xilinx and Atmel have a considerable advantage over other manufacturers. Only they provide the possibility of performing dynamic partial reconfiguration in their devices. There are three reconfigurability features in particular that would be of great interest for reconfigurable bio-inspired systems: granularity, multi-context configuration memories, and distributed self-reconfigurability.

- **Granularity.** Xilinx proposes two design flows for partial reconfiguration, among which only the module-based is actually used by system designers. However, in Virtex-II and Spartan-3 families, the configuration logic supports a finer granularity. A frame constitutes the minimum reconfigurable portion, where a frame has a variable length depending on the device size, and contains the configuration information for a set of elements going from the top to the bottom of the FPGA.

  In a dynamically adaptive system, when modifying a multiplexer's selection or an ANN synaptic weight, it would be desirable to reconfigure just the bits involved, instead of the whole frame, requiring a very fine-grained reconfiguration. In this sense, the Virtex-4 family configuration bitstream offers a finer granularity than the Virtex-II. A Virtex-4

frame has a fixed size of 41 words (for a Virtex-II it ranges between 26 and 286 words), which is independent of the device size. This can be seen as an improvement of commercial FPGAs toward the requirements of our systems; however, configuration granularity is not among the needs of commercial FPGA systems, so future improvements in this direction will be hardly possible.

- **Multi-context configuration memories.** Currently, dynamic reconfigurable systems are not among industrially competitive solutions. Because of this, FPGA manufacturers do not target these reconfigurability features when they design their devices. A consequence of this is the absence of multi-context configuration memories in commercial FPGAs. This fact makes the reconfiguration a slow and inefficient process, instead of updating a given coprocessor in a single clock cycle as is possible with multi-context reconfigurable devices.

  However, during the last years researchers have been studying reconfigurable architectures. By using both custom and commercial devices, dynamic reconfigurable systems have been shown to perform well in several tasks. Future commercial FPGAs will certainly provide enhanced reconfigurability features, such as multi-context configuration memories and faster configuration access ports.

- **Distributed self-reconfigurability.** In a real cellular system, a cell is able to self-modify without requesting that an external entity does it. Biological organisms' self-repair mechanism is not a centralized process, but a completely distributed process where cells are able to self-reproduce in an autonomous way, while interacting with other cells. State-of-the-art FPGAs require a configuration access port for modifying the configuration bitstream (e.g. Jtag or ICAP in the case of Xilinx FPGAs). The self-reconfigurable system presented in this thesis requires the ICAP port for self-reconfiguring, instead of allowing each neuron or automata to directly self-reconfigure.

  The interest of FPGAs' manufacturers in including logic cells with self-reconfiguration capabilities in commercial devices is certainly minimal. This reconfigurability enhancement does not represent a performance increase or a desired feature, on the contrary it would require very complex design tools for supporting these distributed self-reconfigurable systems. Consequently, at least for the next years we will not have commercial devices supporting this feature, unless we are able to show significant performance breakthroughs achieved with it.

Another interesting aspect is the concept of neighborhood in cellular systems. Biological cellular systems are 3-dimensional structures, whose neighborhood concept is much more complex than the one exhibited by any artificial cellular system. Chemical interactions among cells are not limited to cells having physical contact, but it extends to every cell present in a certain radius. In the same way, the concept of neighborhood in neural systems is not necessarily related to the neuron's position; for instance, a neuron's axon can measure several decimeters (a typical soma diameter is 10μm) allowing the output of a neuron to be connected to other neuron's dendrites which are very far from it.

Efficient hardware implementations of artificial cellular systems able to mimic the above described biological cellular systems will require enhanced routing resources. Current trends

on FPGA design are enlarging the concept of neighborhood in a logic cell; recent Virtex-5 devices from Xilinx feature a new diagonal routing [243] instead of the traditional segmented routing used until Virtex-4. The diagonal routing increases the physical neighborhood connectivity of a logic cell. A second and more relevant limitation is the 2-dimensional nature of current IC design technologies. However, current research in nanotechnologies is aiming to build 3-dimensional circuits, which would allow one to mimic in a more accurate way a real 3-dimensional biological organism.

10.3.2 Dynamic topology architectures

A living being's structure is very dynamic, the cortical plasticity exhibited by the human brain during the first years of life is a clear example of it. A child's early interactions have a direct impact on the way in which his brain gets physically connected initially. Another example could be the morphological modifications exhibited by a species during centuries of evolution, or simply the cellular replacement experienced by an organism after an injury.

Whichever biological structure is to be mimicked by a hardware system, topological flexibility is always a desired feature. There are not many platforms or design techniques addressing this issue, given that it is a problem almost exclusive to bio-inspired architectures. The POEtic tissue is an example of an FPGA supporting the dynamic reconfigurability of its routing resources [203, 204]: inter-cellular communications can be safely re-routed by reconfiguring the multiplexers defining logic cell's inputs.

Commercial FPGAs do not facilitate the implementation of such dynamic routing, it must be manually setup by the bio-inspired system designer. In section 7.2 of this thesis, a reconfigurable interconnection matrix is presented, which is able to perform it in a safe way. It is the first reconfigurable adaptable topology platform, which actually reconfigures the commercial FPGA supporting it.

Future reconfigurable topologies must support a wider neighborhood, and must also target 3-dimensional (or more) cellular systems. Future nano-technology developments will certainly provide the technological processes to achieve systems with a considerably larger number of cells and interconnections.

10.3.3 Self-reconfigurable adaptive systems

Self-reconfigurable systems are relatively new. Most of them have a basic structure where a processor reconfigures a piece of logic in the form of a coprocessor, a peripheral, or a functional unit. Furthermore, existing implementations are still constrained to a certain number of pre-designed configurations.

In these aspects, biological systems differ substantially from engineered self-reconfigurable systems. Additionally, we need also the algorithms for determining how to deal with distributed self-reconfiguration, as discussed in subsection 10.3.1, to provide the hardware support for performing it. What mechanism can allow a cell to determine its own genetic description? A possible answer is proposed by the cellular programming algorithm presented in subsection 3.2.4; however, this algorithm remains very simplistic and is still far from mimicking a real biological system or achieving any ”useful” solution.

Summarizing, there is still a lot of work to do in defining self-reconfigurable architectures where the configuration is not simply driven by an internal processor (which can be seen as simply moving the processor from the PC to the chip). In future self-reconfigurable architectures the cell array itself must have the possibility to self-reconfigure in a distributed and autonomous way. Moreover, how to cope with this reconfigurability for allowing a system featuring such architecture to perform any ”useful” computation remains an open question.

10.3.4 Evolvable hardware substrates

Future trends in nanotechnology are guiding us to "Avogadro computers", massively parallel devices with 10^{23} transistors. What to do with such a large number of transistors? How to use, interconnect and program these machines goes beyond the current engineering knowledge. EHW architectures and algorithms arise as a promising solution for dealing with the design complexity proposed by the construction of these machines.

This thesis has focused on implementations on silicon circuits, which reflects the main developments performed by the EHW community. However, other types of substrates have also been evolved which extend the domain and constitute new directions for EHW. At NASA, researchers have been working on evolving antennas for space missions [108, 109]. Miller is currently working on evolving liquid crystal (LC) [129]: by applying electric fields mapped from a genome he modifies the LC molecular alignment for implementing a desired function. Molecular circuit design constitutes another promising evolvable substrate: Masiero et al. [121] report the use of a GA for tuning component parameters in a molecular circuit. Quantum circuit synthesis is another potential substrate for EHW [194], given that designing circuits in such a substrate will require new design paradigms.

Bibliography

[1] H. A. Abbass. Speeding up backpropagation using multiobjective evolutionary algorithms. *Neural Computation*, 15(11):2705–2726, 2003.

[2] Actel Corp. Axelerator family FPGAs. www.actel.com, Dec, 2005.

[3] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):288–298, 2004.

[4] Altera Corp. Stratix II device handbook. www.altera.com, Apr., 2006.

[5] P. J. Angeline. Evolutionary optimization versus particle swarm optimization: Philosophy and performance differences. In *Evolutionary Programming*, pages 601–610, 1998.

[6] J. W. Atmar. *Speculation on the evolution of intelligence and its possible realization in machine form.* PhD thesis, New Mexico State University, 1976.

[7] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP - a self-reconfigurable data processing architecture. *Journal of Supercomputing*, 26(2):167–184, 2003.

[8] T. Bäck. *Evolutionary algorithms in theory and practice : evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, New York, 1996.

[9] J. Becker, T. Pionteck, and M. Glesner. DReAM: A dynamically reconfigurable architecture for future mobile communication applications. *Proceedings of Field-Programmable Logic and Applications. The Roadmap to Reconfigurable Computing: 10th International Conference, FPL 2000, Villach, Austria, LNCS*, 1896:312–321, 2000.

[10] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In John McCanny, John McWhirter, and Earl Swartzlander, editors, *Systolic Array Processors*, pages 301–309. Prentice Hall, 1989.

[11] J. Beutel and O. Kasten. A minimal Bluetooth-based computing and communication platform. Technical report, Technical Report, Computer Engineering and Networks Lab, Swiss Federal Institute of Technology (ETH) Zurich, 2001.

[12] H. G. Beyer and H. P. Schwefel. Evolution strategies - a comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[13] Biologically Inspired Robotics Group (BIRG). YaMoR modular robot webpage. http://birg2.epfl.ch/yamor.

[14] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan. A self-reconfiguring platform. 2778:565–574, 2003.

[15] R. A. Brooks. New approaches to robotics. *Science*, 253:1227–1232, 1991.

[16] S. Brown and J. Rose. FPGA and CPLD architectures: a tutorial. *IEEE Design & Test of Computers*, 13(2):42–57, 1996.

[17] J. Buck and E. Buck. Synchronous fireflies. *Scientific American*, pages 74–85, 1976.

[18] R.O. Canham and A. Tyrrell. Evolved fault tolerance in evolvable hardware. *In proceedings of the Congress on Evolutionary Computation (CEC2002)*, pages 1267–1272, 2002.

[19] Y. U. Cao, A. S. Fukunaga, and A. B. Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4(1):7–27, 1997.

[20] M. Capcarrere, A. Tettamanzi, M. Tomassini, and M. Sipper. Studying parallel evolutionary algorithms: The cellular programming case. In *Parallel Problem Solving from Nature - PPSN V*, volume 1498, pages 573–582. Springer-Verlag, 1998.

[21] J.W. Carter. *Digital Designing with Programmable Logic Devices*. Prentice-Hall,Inc., Upper Saddle River, NJ, USA, 1997.

[22] A. Castano, A. Behar, and P.M. Will. The conro modules for reconfigurable robots. *IEEE/ASME Transactions on Mechatronics*, 7(4):403–409, December 2002.

[23] D.C. Chen and J.M. Rabaey. A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths. *Solid-State Circuits, IEEE Journal of*, 27(12):1895–1904, 1992.

[24] S. B. Cho and K. Shimohara. Evolutionary learning of modular neural networks with genetic programming. *Applied Intelligence*, 9(3):191–200, 1998.

[25] C. A. Coello Coello, E. Hernández Luna, and A. Hernández Aguirre. Use of particle swarm optimization to design combinational logic circuits. In *ICES*, pages 398–409. Springer Verlag, 2003.

[26] H. Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, New York, NY, USA, 1997.

[27] A. Costa, A. De Gloria, P. Faraboschi, A. Pagni, and G. Rizzotto. Hardware solutions for fuzzy control. *Proceedings of the IEEE*, 83(3):422–434, March 1995.

[28] H. de Garis. Evolvable hardware: Genetic programming of a darwin machine. *Artificial Neural Nets and Genetic Algorithms, Proceedings of the International Conference in Innsbruck, Austria*, pages 441–449, 1993.

[29] H. de Garis. Growing an artificial brain with a million neural net modules inside a trillion cell cellular automaton machine. *Proc. of the Fourth International Symposium on Micro Machine and Computer Science*, pages 211–214, 1993.

[30] S. Donthi and R.L. Haggard. A survey of dynamically reconfigurable fpga devices. In *Proceedings of the 35th Southeastern Symposium on System Theory*, pages 422–426, 2003.

[31] J. Dumoulin, J. A. Foster, J. F. Frenzel, and S. McGrew. Special purpose image convolution with evolvable hardware. In *Real-World Applications of Evolutionary Computing, EvoWorkshops 2000, Proceedings, LNCS*, volume 1803, pages 1–11. Springer-Verlag, 2000.

[32] L. Durbeck and N. Macias. Defect-tolerant, fine-grained parallel testing of a cell matrix. *Proceedings of SPIE ITCom 2002 Series*, 4867:71–85, 2002.

[33] C. Ebeling, D. Cronquists, and P. Franlin. Configurable computing: The catalyst for high performance architectures. In *Proceedings of the IEEE International Conference on Application Specific, Systems, Architectures, and Processors*, pages 364–373, July, 1997.

[34] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science, MHS'95*, volume 2778, pages 39–43, 1995.

[35] D. Floreano and J. Urzelai. Evolutionary robots with on-line self-organization and behavioral fitness. *Neural Networks*, 13(4-5):431–443, 2000.

[36] D. B. Fogel. *Evolutionary computation : toward a new philosophy of machine intelligence*. IEEE Press, New York, 2nd edition, 2000.

[37] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial intelligence through simulated evolution*. Wiley, New York, NY, 1966.

[38] J. Gaisler. The LEON processor user's manual, January 2003.

[39] C. Gershenson. Classification of random boolean networks. *Artificial Life VIII: Proceedings of the Eight International Conference on Artificial Life.*, pages 1–8, 2002.

[40] W. Gerstner and W. Kistler. *Spiking Neuron Models. Single Neurons, Populations, Plasticity*. Cambridge University Press, Cambridge, 2002.

[41] K. Glette and J. Torresen. A flexible on-chip evolution system implemented on a Xilinx Virtex-II-PRO device. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 3637, pages 66–75. Springer-Verlag, 2005.

[42] M. Goeke, M. Sipper, D. Mange, A. Stauffer, E. Sanchez, and M. Tomassini. Online autonomous evolware. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 1259, pages 96–106. Springer-Verlag, 1997.

[43] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1):81–89, 1991.

[44] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Pub. Co., Reading, Mass., 1989.

[45] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[46] J. González-Gómez, E. Aguayo, and E. Boemo. Locomotion of a modular worm-like robot using a FPGA-based embedded MicroBlaze soft-processor. In *Proceedings CLAWAR05*, pages 3397–3402, 2004.

[47] A.A. Gray, C. Lee, P. Arabshahi, and J. Srinivasan. Object-oriented reconfigurable processing for wireless networks. In *IEEE International Conference on Communications, 2002. ICC 2002.*, pages 497–501, New York, USA, 2002.

[48] S. A. Guccione, D. Levi, and P. Sundararajan. JBits: A java-based interface for reconfigurable computing. *Proc. of the Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[49] V. G. Gudise and G. K. Venayagamoorthy. FPGA placement and routing using particle swarm optimization. In *ISVLSI*, pages 307–308, 2004.

[50] J.C. Haartsen. The Bluetooth radio system. *IEEE Personal Communications*, 7(1):28–36, 2000.

[51] P. Haddow and G. Tufte. Evolving a robot controller in hardware. *Proceedings of the Norwegian Computer Science Conference (NIK99)*, pages 141–150, 1999.

[52] P. Haddow and G. Tufte. Bridging the genotype-phenotype mapping for digital FPGAs. *Proceedings of the third NASA/DoD Workshop on Evolvable Hardware*, page 109, 2001.

[53] P.C. Haddow and G. Tufte. An evolvable hardware FPGA for adaptive hardware. *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 553–560, 2000.

[54] P.C. Haddow, G. Tufte, and P. Van Remortel. Shrinking the genotype :L-systems for EHW? In *ICES'01 : Proceedings of the 4th International Conference on Evolvable Systems : From Biology to Hardware*, pages 128–139, 2001.

[55] A. Hamilton, K. Papathanasiou, M. R. Tamplin, and T. Brandtner. Palmo: Field programmable analogue and mixed-signal VLSI for evolvable hardware. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 1478, pages 335–344. Springer-Verlag, 1998.

[56] L. K. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions in Pattern Analysis and Machine Intelligence*, 12(10):993–1001, 2002.

[57] B.L. Happel and J.M. Murre. Design and evolution of modular neural network architectures. *Neural Networks*, 7(6/7):985–1004, 1994.

[58] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In Kenneth L. Pocek and Jeffery Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96. IEEE Computer Society Press, 1997.

[59] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[60] S. Haykin. *Neural Networks, A Comprehensive Foundation*. Prentice-Hall, Inc, New Jersey, 2 edition, 1999.

[61] D. O. Hebb. *The Organization of Behavior*. John Wiley, New York, 1949.

[62] R. Hecht-Nielsen. Theory of the backpropagation neural network. In *Proceedings of the International Joint Conference on Neural Networks*, volume 1, pages 593–606, 1989.

[63] H. Hemmi, J. Mizoguchi, and K. Shimohara. Development and evolution of hardware behaviors. *Towards Evolvable Hardware, The Evolutionary Engineering Approach, LNCS*, 1062:250 – 265, 1996.

[64] E. Hernández Luna, A. Hernández Aguirre, and C. A. Coello Coello. On the use of a population-based particle swarm optimizer to design combinational logic circuits. In *2004 NASA/DoD Conference on Evolvable Hardware (EH'04)*, page 183, 2004.

[65] T. Higuchi, M. Iwata, I. Kajitani, H. Iba, Y. Hirao, Furaya T., and B. Manderick. Evolvable hardware and its application to pattern recognition and fault-tolerant systems. *Towards Evolvable Hardware, The Evolutionary Engineering Approach, LNCS*, 1062:118–135, 1996.

[66] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, M. Salami, N. Kajihara, and N. Otsu. Real-world applications of analog and digital evolvable hardware. *IEEE Transactions on Evolutionary Computation*, 3(3):220–235, 1999.

[67] T. Higuchi, M. Iwata, and W. Liu. *Evolvable systems: from biology to hardware: Proceedings of the First International Conference, ICES'96*, volume 1259 of *LNCS*. Springer-Verlag, Heidelberg, Germany, 1997.

[68] T. Higuchi, M. Iwata, H. Sakanashi, E. Takahashi, M. Murakawa, and I. Kajitani. Dynamic adaptive devices and their applications. *Bulletin of the Electrotechnical Laboratory. Special Issue: RWC Research-Toward Realization of Real World Intelligence*, 64(4/5):75–82, 2000.

[69] T. Higuchi, T. Niwa, Tanaka T., H. Iba, H. de Garis, and T. Furuya. Evolving hardware with genetic learning: A first step towards building a Darwin machine. *From animals to animats 2, Proceedings of the International Conference on Simulation of Adaptive Behavior*, pages 417–424, 1993.

[70] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *Acm Sigplan Notices*, 35(11):93–104, 2000.

[71] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology-London*, 117(4):500–544, 1952.

[72] G. Hollingworth, S. Smith, and A. Tyrrell. The intrinsic evolution of Virtex devices through internet reconfigurable logic. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 1801, pages 72–79. Springer-Verlag, 2000.

[73] G. Hollingworth, S. Smith, and A. Tyrrell. Safe intrinsic evolution of Virtex devices. *Proceedings of the 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 195–202, 2000.

[74] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximations. *Neural Networks*, 2:359–366, 1989.

[75] M. Hubner, K. Paulsson, M. Stitz, and J. Becker. Novel seamless design-flow for partial and dynamic reconfigurable systems with customized communication structures based on Xilinx Virtex-II FPGAs. *ARCS'05, System Aspects in Organic and Pervasive Computing, Workshop Proceedings*, pages 39–44, 2005.

[76] M. Husken, C. Igel, and M. Toussaint. Task-dependent evolution of modularity in neural networks. *Connection Science*, 14(3):219–229, 2002.

[77] C. Igel and M. Kreutz. Operator adaptation in evolutionary computation and its application to structure optimization of neural networks. *Neurocomputing*, 55(1-2):347–361, 2003.

[78] A. J. Ijspeert. A connectionist central pattern generator for the aquatic and terrestrial gaits of a simulated salamander. *Biological Cybernetics*, 84(5):331–348, 2001.

[79] A. J. Ijspeert. Vertebrate locomotion. In M. Arbib, editor, *The handbook of brain theory and neural networks*, pages 649–654. MIT Press, 2003.

[80] M. Iwata, I. Kajitani, Y. Liu, N. Kajihara, and T. Higuchi. Implementation of a gate-level evolvable hardware chip. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 2210, page 38. Springer-Verlag, 2001.

[81] M. Iwata, I. Kajitani, H. Yamada, H. Iba, and T. Higuchi. A pattern recognition system using evolvable hardware. In *Parallel Problem Solving from Nature (PPSN IV), LNCS*, volume 1141, pages 761–770. Springer-Verlag, 1996.

[82] H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004.

[83] K. Jarvinen, M. Tommiska, and J. Skytta. Comparative survey of high-performance cryptographic algorithm implementations on FPGAs. *IEE Proceedings on Information Security*, 152(1):3–12, 2005.

[84] B. C. Kahne. *A Genetic Algorithm-Based Place-and-Route Compiler For A Run-time Reconfigurable Computing System*. Master thesis, Virginia Polytechnic Institute and State University, 1997.

[85] I. Kajitani, T. Hoshino, N. Kajihara, M. Iwata, and T. Higuchi. An evolvable hardware chip and its application as a multi-function prosthetic hand controller. *Proceedings of the sixteenth national conference on Artificial intelligence*, pages 182–187, 1999.

[86] I. Kajitani, M. Iwata, M. Harada, and T. Higuchi. A myoelectric controlled prosthetic hand with an evolvable hardware LSI chip. *Technology and Disability, Special Issue: Advances in the Control of Prosthetic Arms*, 15(2):129–143, 2003.

[87] J. F. Kennedy, R. C. Eberhart, and Y. Shi. *Swarm intelligence*. Morgan Kaufmann Publishers, San Francisco, 2001.

[88] D. Keymeulen, M. Iwata, Y. Kuniyoshi, and T. Higuchi. Online evolution for a self-adapting robotic navigation system using evolvable hardware. *Artificial Life 4*, pages 359–393, 1998.

[89] D. Kim. An implementation of fuzzy logic controller on the reconfigurable FPGA system. *IEEE Transactions on Industrial Electronics*, 47(3):703–715, 2000.

[90] B. Kosko. Fuzzy systems as universal approximators. *IEEE Transactions on Computers*, 43(11):1329–1333, 1994.

[91] J. R. Koza. *Genetic programming : on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, Cambridge, Mass., 1992.

[92] J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane. Synthesis of topology and sizing of analog electrical circuits by means of genetic programming. *Computer Methods in Applied Mechanics and Engineering*, 186(2):459–482, 2000.

[93] J. R. Koza, F. H. Bennett, J. Hutchings, S. L. Bade, M. A. Keane, and D. Andre. Evolving sorting networks using genetic programming and rapidly reconfigurable field-programmable gate arrays. *Workshop on Evolvable Systems. International Joint Conference on Artificial Intelligence*, pages 27–32, 1997.

[94] R. Kruse, J. Gebhardt, and F. Klawonn. *Foundations of Fuzzy Systems*. John Wiley and Sons, New York, 1994.

[95] H. Kummert. The PPP Triple-DES Encryption Protocol (3DESE). RFC 2420, September 1998.

[96] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the internation symposium on Field programmable gate arrays*, pages 21–30, 2006.

[97] H. Kurokawa, A. Kamimura, E. Yoshida, K. Tomita, S. Kokaji, and S. Murata. M-TRAN II: metamorphosis from a four-legged walker to a caterpillar. In *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2454– 2459, 2003.

[98] A. Lagger, A. Upegui, E. Sanchez, and I. Gonzalez. Self-reconfigurable pervasive platform for cryptographic application. In *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (To appear)*, Madrid, Spain, 2006.

[99] C. G. Langton. *Artificial life : an overview*. Complex adaptive systems. MIT Press, Cambridge, Mass., 1995.

[100] Lattice Corp. LatticeSC family data sheet. www.latticesemi.com, Apr, 2006.

[101] P. Layzell. 'evolvable motherboard': A test platform for the research of intrinsic hardware evolution. *Tech rep. University of Sussex*, 1998.

[102] P. Layzell. A new research tool for intrinsic hardware evolution. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 1478, pages 47–56. Springer-Verlag, 1998.

[103] D. W. Lee, C. B. Ban, K. B. Sim, H. S. Seok, K. J. Lee, and B. T. Zhang. Behavior evolution of autonomous mobile robot using genetic programming based on evolvable hardware. *Proceeding of 2000 IEEE International Conference on Systems, Man, and Cybernetics*, 5:3835–3840, 2000.

[104] D. Levi and S. A. Guccione. GeneticFPGA: Evolving stable circuits on mainstream FPGA devices. *Proceedings of The First NASA/DOD Workshop on Evolvable Hardware*, page 12, 1999.

[105] C. Lindgren and M. Nordahl. Universal computation in simple one dimensional cellular automata. *Complex Systems*, 4:299–318, 1990.

[106] Y. Liu. *Evolvable systems: from biology to hardware: Proceedings of the 4th international conference, ICES 2001, Tokyo, Japan, October 3-5, 2001*, volume 2210 of *LNCS*. Springer-Verlag, Heidelberg, Germany, 2001.

[107] Y. Liu and X. Yao. Evolving modular neural networks which generalize well. *Proceedings of the IEEE Conference on Evolutionary Computation*, pages 605–610, 1997.

[108] J. Lohn, J. Crawford, A. Globus, G. Hornby, W. Kraus, G. Larchev, A. Pryor, and D. Srivastava. Evolvable systems for space applications. *Proceedings of the International Conference on Space Mission Challenges for Information Technology (SMC-IT 2003)*, 2003.

[109] J. Lohn, D. Linden, G. Hornby, W. Kraus, and A. Rodriguez-Arroyo. Evolutionary design of an X-band antenna for NASA's space technology 5 mission. *Proceedings of the 2003 NASA/DoD Conference on Evolvable Hardware (EH'03)*, page 155, 2003.

[110] J. D. Lohn, D. Gwaltney, G. Hornby, R. Zebulum, D. Keymeulen, and A. Stoica. *Proceedings of the 2005 NASA/DOD Conference on Evolvable Hardware : Washington,D.C., June 29 - July 1, 2005*. IEEE Computer Society, Los Alamitos, Calif., 2005.

[111] G. Lu, E. M.C. Filho, V. Castro Alves, H. Singh, M. H. Lee, N. Bagherzadeh, and F.J. Kurdahi. The morphosys dynamically reconfigurable system-on-chip. *Proceedings of The First NASA/DOD Workshop on Evolvable Hardware*, page 152, 1999.

[112] T. A. Ly and J. T. Mowchenko. Applying simulated evolution to high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(3):389–409, 1993.

[113] W. Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997.

[114] W. Maass and Ch. Bishop. *Pulsed Neural Networks*. The MIT Press, Massachusetts, 1999.

[115] W. Maass and H. Markram. On the computational power of recurrent circuits of spiking neurons. *Journal of Computer and System Sciences*, 69(4):593–616, 2004.

[116] W. Maass, T. Natschlager, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[117] N. Macias. The PIG paradigm: The design and use of a massively parallel fine grained self-reconfigurable infinitely scalable architecture. *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, page 175, 1999.

[118] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti. Toward robust integrated circuits: The embryonics approach. *Proceedings of the IEEE*, 88(4):516–540, April 2000.

[119] D. Marbach and A.J. Ijspeert. Online optimization of modular robot locomotion. In *Proceedings of the IEEE Int. Conference on Mechatronics and Automation (ICMA 2005)*, pages 248–253, 2005.

[120] T. Martinek and L. Sekanina. An evolvable image filter: Experimental evaluation of a complete hardware implementation in FPGA. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 3637, pages 76–85. Springer-Verlag, 2005.

[121] L. P. Masiero, M. Pacheco, C.R. Hall, and C. Santini. Molecular circuit design. *Proceedings of the 2005 NASA/DOD Conference on Evolvable Hardware : Wasshington,D.C., June 29 - July 1, 2005*, page 307, 2005.

[122] MathStar Corp. Field programmable object arrays. www.mathstar.com, Aug, 2004.

[123] D. Mattsson and M. Christensson. *Evaluation of synthesizable CPU cores*. Master thesis, Chalmers University Of Technology, 2004.

[124] C. Maxfield. *The Design Warrior's Guide to FPGAs*. Elsevier, Oxford,UK, 2004.

[125] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in neural nets. *Bulletin of Math. Biophys.*, 5:115–137, 1943.

[126] R. Mendes. *Population Topologies and Their Influence in Particle Swarm Performance*. PhD thesis, Escola de Engenharia, Universidade do Minho, May 2004.

[127] G. Mermoud, A. Upegui, C. A. Pena, and E. Sanchez. A dynamically-reconfigurable FPGA platform for evolving fuzzy systems. In *Computational Intelligence and Bioinspired Systems, LNCS*, volume 3512, pages 572–581. Springer-Verlag, 2005.

[128] J. Miller. *Evolvable systems: from biology to hardware: Proceedings of the third International Conference, ICES 2000, Edinburgh, Scotland, UK, April 17-19, 2000*, volume 1801 of *LNCS*. Springer, Heidelberg, Germany, 2000.

[129] J. F. Miller and K. Downing. Evolution in materio: looking beyond the silicon box. *Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware*, pages 167–176, 2002.

[130] M. Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, Mass., 1996.

[131] M. Mitchell, J. P. Crutchfield, and P. T. Hraber. Evolving cellular-automata to perform computations - mechanisms and impediments. *Physica D*, 75(1-3):361–391, 1994.

[132] S. Mitra and Y. Hayashi. Neuro-fuzzy rule generation: Survey in soft computing framework. *IEEE Transactions on Neural Networks*, 11(3):748–768, 2000.

[133] T. Miyamori and K. Olukotun. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[134] R. Moeckel, C. Jaquier, K. Drapel, E. Dittrich, A. Upegui, and A. Ijspeert. YaMoR and bluemove - an autonomous modular robot with Bluetooth interface for exploring adaptive locomotion. In *Proceedings CLAWAR05*, pages 685–692, 2005.

[135] R. Moeckel, C. Jaquier, K. Drapel, E. Dittrich, A. Upegui, and A.J. Ijspeert. Exploring adaptive locomotion with YaMoR, a novel autonomous modular robot with Bluetooth interface. *Industrial Robot*, 33(4):285–290, 2006.

[136] P. D. Moerland and E. Fiesler. Neural network adaptations to hardware implementations. In E. Fiesler and R. Beale, editors, *Handbook of Neural Computation*, pages 1–13. Institute of Physics Publishing and Oxford University Publishing, New York, 1997.

[137] P. Moore and G. K. Venayagamoorthy. Evolving combinational logic circuits using a hybrid quantum evolution and particle swarm inspired algorithm. In *Evolvable Hardware*, pages 97–102. Springer Verlag, 2005.

[138] F. J. Morales, J. P. Crutchfield, and M. Mitchell. Evolving two-dimensional cellular automata to perform density classification: A report on work in progress. *Parallel Computing*, 27(5):571–585, 2001.

[139] J. M. Moreno, J. Cabestany, J. Madrenas, E. Canto, J. Faura, and J. M. Insenser. Approaching evolvable hardware to reality: The role of dynamic reconfiguration and virtual meso-structures. *Proceedings of the Seventh International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems*, pages 163–170, 1999.

[140] J. M. Moreno, J. Madrenas, and J. Cosp. *Evolvable systems: from biology to hardware: Proceedings of the 6th international conference, ICES 2005, Sitges, Spain, September 12-14, 2005*, volume 3637 of *LNCS*. Springer, Heidelberg, Germany, 2005.

[141] M. Murakawa, S. Yoshizawa, and T. Higuchi. Adaptive equalization of digital communication channels using evolvable hardware. In *ICES*, pages 379–389, 1996.

[142] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. Hardware evolution at function level. *Parallel Problem Solving from Nature (PPSN IV), LNCS*, 1141:62–71, 1996.

[143] M. Murakawa, S. Yoshizawa, I. Kajitani, X. Yao, N. Kajihara, M. Iwata, and T. Higuchi. The GRD chip: Genetic reconfiguration of DSPs for neural network processing. *IEEE Transactions on Computers*, 48(6):628–639, 1999.

[144] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji. M-TRAN: Self-reconfigurable modular robotic system. *IEEE-ASME Transactions on Mechatronics*, 7(4):431–441, 2002.

[145] Y. Niv, D. Joel, I. Meilijson, and E. Ruppin. Evolution of reinforcement learning in uncertain environments: A simple explanation for complex foraging behaviors. *Adaptive Behavior*, 10(1):5–24, 2002.

[146] OpenCores. OpenRISC 1000 architecture manual, Jan, 2003.

[147] D.W. Opitz and J.W. Shavlik. Generating accurate and diverse members of a neural-network ensemble. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 535–541. The MIT Press, 1996.

[148] P. Pavan, R. Bez, P. Olivo, and E. Zanoni. Flash memory cells-an overview. *Proceedings of the IEEE*, 85(8):1248–1271, 1997.

[149] J. Peña, A. Upegui, and E. Sanchez. Particle swarm optimization with discrete recombination: An online optimizer for evolvable hardware. In *Proceedings of the 1st NASA /ESA Conference on Adaptive Hardware and Systems(AHS-2006)*, pages 163–170, 2006.

[150] C. A. Peña Reyes. *Coevolutionnary Fuzzy Modeling*, volume 3204. Springer, Berlin, lecture notes in computer science edition, 2004.

[151] M. Peng, C.L. Nikias, and J.G. Proakis. Adaptive equalization for PAM and QAM signals with neural networks. *Conference Record of the Twenty-Fifth Asilomar Conference on Signals, Systems and Computers, 1991*, 1:496–500, 1991.

[152] A. Perez-Uribe. *Structure-adaptable digital neural networks*. Phd thesis, EPFL, 1999.

[153] J. Pinter. *Global optimization in action (Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications)*. Kluwer Academic Publishers, Dordrecht / Boston / London, 1996.

[154] C. Plessl, R. Enzler, H. Walder, J. Beutel, M. Platzner, and L. Thiele. Reconfigurable hardware in wearable computing nodes. In *Sixth International Symposium on Wearable Computers*, pages 215–222, Seattle, Washington, 2002.

[155] C. Plessl and M. Platzner. Zippy - a coarse-grained reconfigurable array with support for hardware virtualization. In *Proceedings of the 16th IEEE International Conference on Application-Specific Systems, Architecture Processors - ASAP*, pages 213–218, 2005.

[156] J.B. Pollack. *On Connectionist Models of Natural Language Processing*. Phd thesis, University of Illinois -Urbana, 1987.

[157] B. Prince. *Semiconductor Memories: A Handbook of Design, Manufacture and Application*. John Wiley & Sons, 1997.

[158] Quicklogic Corp. μWatt FPGAs: Ultra-low power programmable logic soutions, Jul, 2005.

[159] S. Qureshi. Adaptive equalization. *IEEE Communications Magazine*, pages 9–16, March 1992.

[160] T.S. Ray. An approach to the synthesis of life. *Artificial Life II (SFI Studies in the Sciences of Complexity, vol X)*, pages 371–408, 1992.

[161] R. Reed. Pruning algorithms - a survey. *IEEE Transactions on Neural Networks*, 4(5):740–747, 1993.

[162] R. Rivest. The RC4 encryption algorithm (proprietary). *RSA Data Security Inc.*, Mar. 12, 1992.

[163] D. Roggen, S. Hofmann, Y. Thoma, and D. Floreano. Hardware spiking neural network with run-time reconfigurable connectivity. In *5th NASA / DoD Workshop on Evolvable Hardware (EH 2003)*, pages 199–208. IEEE Computer Society, 2003.

[164] E. Ronco and P.J. Gawthrop. Modular neural networks: a state of the art. *Technical Report CSC-95026, Center for System and Control, University of Glasgow*, May 1995.

[165] E. Ros, R. Agis, R. R. Carrillo, and E. M. Ortigosa. Post-synaptic time-dependent conductances in spiking neurons: FPGA implementation of a flexible cell model. In *Artificial Neural Nets Problem Solving Methods, Pt II*, volume 2687, pages 145–152. Springer-Verlag, 2003.

[166] D. Saha and A. Mukherjee. Pervasive computing: a paradigm for the 21st century. *Computer*, 36(3):25–31, 2003.

[167] H. Sakanashi, M. Iwata, and T. Higuchi. Evolvable hardware for lossless compression of very high resolution bi-level images. *IEE Proceedings-Computers and Digital Techniques*, 151(4):277–286, 2004.

[168] H. Sakanashi, M. Iwata, D. Keymulen, M. Murakawa, I. Kajitani, M. Tanaka, and T. Higuchi. Evolvable hardware chips and their applications. *Proceedings of International Conference on Systems, Man, and Cybernetics*, 5:559–564, 1999.

[169] E. Sanchez, D. Mange, M. Sipper, M. Tomassini, A. Perez-Uribe, and A. Stauffer. Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 1259, pages 35–54. Springer-Verlag, 1997.

[170] E. Sanchez, M. Sipper, J.O. Haenni, J.L. Beuchat, A. Stauffer, and A. Perez-Uribe. Static and dynamic configurable systems. *IEEE Transactions on Computers*, 48(6):556–564, 1999.

[171] E. Sanchez and M. Tomassini. *Towards Evolvable Hardware, The Evolutionary Engineering Approach*, volume 1062 of *LNCS*. Springer-Verlag, Heidelberg, Germany, 1996.

[172] L. Sekanina. Towards evolvable IP cores for FPGAs. *Proceedings of the 2003 NASA/DOD Conference on Evolvable Hardware : Chicago, Illinois, July 9-11, 2003*, pages 145–154, 2003.

[173] L. Sekanina. Virtual reconfigurable circuits for real-world applications of evolvable hardware. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 2606, pages 186–197. Springer-Verlag, 2003.

[174] L. Sekanina. *Evolvable components from theory to hardware implementations*. Springer, Berlin, 2004.

[175] L. Sekanina and V. Drabek. The concept of pseudo evolvable hardware. *Proceedings of the IFAC Workshop on Programmable Devices and Systems (PDS00)*, page 6, 2000.

[176] L. Sekanina and S. Friedl. On routine implementation of virtual evolvable devices using COMBO6. *Proceedings of the 2004 NASA/DOD Conference on Evolvable Hardware : Seattle, Wasshington, July 24-26, 2004*, pages 63–70, 2004.

[177] L. Sekanina and R. Ruzicka. Easily testable image operators: the class of circuits where evolution beats engineers. *Proceedings of the 2003 NASA/DOD Conference on Evolvable Hardware : Chicago, Illinois, July 9-11, 2003*, pages 135–144, 2003.

[178] Y. Shi, R. Eberhart, and Y. Chen. Implementation of evolutionary fuzzy systems. *Fuzzy Systems, IEEE Transactions on*, 7:109–119, 1999.

[179] Y. Shi and R. C. Eberhart. A modified particle swarm optimizer. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 69–73, May 1998, Anchorage, Alaska, USA.

[180] Y. Shi and R.C. Eberhart. Empirical study of particle swarm optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation (CEC 99)*, volume 3, pages 1945–1950, 1999.

[181] H. Siegelmann and E. Sontag. Turing computability with neural nets. *Appl. Math. Lett.*, 4(6):77–80, 1991.

[182] H. Singh, M.H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. Chaves Filho. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.*, 49(5):465–481, 2000.

[183] M. Sipper. Quasi-uniform computation-universal cellular automata. In *Proceedings of the European Conference on Artificial Life*, pages 544–554, 1995.

[184] M. Sipper. Virtual reconfigurable circuits for real-world applications of evolvable hardware. In *ECAL'95: 3rd. European Conference on Artificial Life*, pages 544–554. Springer-Verlag, 1995.

[185] M. Sipper. Co-evolving non-uniform cellular automata to perform computations. *Physica D-Nonlinear Phenomena*, 92(3-4):193–208, 1996.

[186] M. Sipper. *Evolution of parallel cellular machines the cellular programming approach*. Springer, Berlin, 1997.

[187] M. Sipper, M. Goeke, D. Mange, A. Stauffer, E. Sanchez, and M. Tomassini. The firefly machine: online evolware. *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 181–186, 1997.

[188] M. Sipper, D. Mange, and A. Pérez-Uribe. *Evolvable systems: from biology to hardware: Proceedings of the second International Conference, ICES 98, Lausanne, Switzerland, September 23-25, 1998*, volume 1478 of *LNCS*. Springer, Heidelberg, Germany, 1998.

[189] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Perez-Uribe, and A. Stauffer. A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. *IEEE Transactions on Evolutionary Computation*, 1(1):83–97, 1997.

[190] M. Sipper and M. Tomassini. Generating parallel random number generators by cellular programming. *International Journal of Modern Physics C*, 7(2):181–190, 1996.

[191] C. Slorach and K. Sharman. The design and implementation of custom architectures for evolvable hardware using off-the-shelf programmable devices. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 1801, pages 197–207. Springer-Verlag, 2000.

[192] M. E. Smid and D. K. Branstad. The Data Encryption Standard: Past and future. *Proc. of the IEEE*, 76:550–559, 1988.

[193] M.J.S. Smith. *Application-Specific Integrated Circuits*. Addison-Wesley Publihing Company, 1997.

[194] L. Spector, H. Barnum, H. J. Bernstein, and N. Swamy. Quantum computing applications of genetic programming. In *Advances in genetic programming*, volume 3, pages 135–160. MIT Press, 1999.

[195] A. Stoica, D. Keymeulen, and J. D. Lohn. *Proceedings of the First NASA/DOD Workshop on Evolvable Hardware : Pasadena, California, July 19-21, 1999*. IEEE Computer Society, Los Alamitos, Calif., 1999.

[196] A. Stoica, J. D. Lohn, R. Katz, D. Keymeulen, and R. Zebulum. *Proceedings of the 2002 NASA/DOD Conference on Evolvable Hardware : Alexandria, Virginia, July 15-18, 2002*. IEEE Computer Society, Los Alamitos, Calif., 2002.

[197] A. Stoica, R. Zebulum, M. Ferguson, D. Keymeulen, and V. Duong. Evolving circuits in seconds: Experiments with a stand-alone board-level evolvable system. *Proceedings of the 2002 NASA/DOD Conference on Evolvable Hardware : Alexandria, Virginia, July 15-18, 2002*, page 67, 2002.

[198] A. Stoica, R. Zebulum, D. Keymeulen, R. Tawel, T. Daud, and A. Thakoor. Reconfigurable VLSI architectures for evolvable hardware: From experimental field programmable transistor arrays to evolution-oriented chips. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):227–232, 2001.

[199] R. S. Sutton and A. G. Barto. *Reinforcement Learning*. MIT Press, 1998.

[200] E. Takahashi, M. Murakawa, Y. Kasai, and T. Higuchi. Power dissipation reductions with genetic algorithms. *Proceedings of the 2003 NASA/DoD Conference on Evolvable Hardware*, pages 111–116, 2003.

[201] E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. A first generation DPGA implementation. In *In Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, May, 1995.

[202] Y. Thoma. *Tissu Numérique Cellulaire à Routage et Configuration Dynamiques*. Phd thesis, EPFL, 2005.

[203] Y. Thoma and E. Sanchez. A reconfigurable chip for evolvable hardware. *Proc. Genetic and Evolutionary Computation Conference (GECCO 2004)*, 1:816–827, 2004.

[204] Y. Thoma, E. Sanchez, J. M. M. Arostegui, and G. Tempesti. A dynamic routing algorithm for a bio-inspired reconfigurable circuit. In *Field-Programmable Logic and Applications, LNCS*, volume 2778, pages 681–690. Springer-Verlag, 2003.

[205] Y. Thoma, G. Tempesti, E. Sanchez, and J. M. M. Arostegui. POEtic: an electronic tissue for bio-inspired cellular applications. *Biosystems*, 76(1-3):191–200, 2004.

[206] A. Thompson. Silicon evolution. *Proceedings of Genetic Programming (GP96), J.R. Koza et al. (Eds), MIT Press*, pages 444 – 452, 1996.

[207] A. Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 1259, pages 390–405. Springer-Verlag, 1997.

[208] A. Thompson, I. Harvey, and P. Husbands. Unconstrained evolution and hard consequences. *Towards Evolvable Hardware, The Evolutionary Engineering Approach, LNCS*, 1062:136–165, 1996.

[209] A. Thompson and P. Layzell. Evolution of robustness in an electronics design. In *Proc. of Evolvable Systems: From Biology to Hardware, LNCS*, volume 1801, pages 218–228. Springer-Verlag, 2000.

[210] T. Toffoli and N. Margolus. *Cellular automata machines : a new environment for modeling*. MIT Press series in scientific computation. MIT Press, Cambridge, Mass., 1987.

[211] O. Torres, J. Eriksson, J. M. Moreno, and A. Villa. Hardware optimization of a novel spiking neuron model for the POEtic tissue. In *Artificial Neural Nets Problem Solving Methods, Pt II*, volume 2687, pages 113–120. Springer-Verlag, 2003.

[212] S. Trimberger. *Field-programmable gate array technology*. Kluwer Academic Publishers, Boston, 1994.

[213] G. Tufte and P. C. Haddow. Biologically-inspired: A rule-based self-reconfiguration of a Virtex chip. In *Computational Science - Iccs 2004, Pt 3, Proceedings*, volume 3038, pages 1249–1256. Springer-Verlag, 2004.

[214] A. M. Tyrrell, P. C. Haddow, and J. Torresen. *Evolvable systems: from biology to hardware: Proceedings of the 5th International Conference, ICES 2003, Trondheim, Norway, March 17-20, 2003*, volume 2606 of *LNCS*. Springer, Heidelberg, Germany, 2003.

[215] A. M. Tyrrell, R. A. Krohling, and Y. Zhou. Evolutionary algorithm for the promotion of evolvable hardware. *IEE Proceedings-Computers and Digital Techniques*, 151(4):267–275, 2004.

[216] A. Upegui, R. Moeckel, E. Dittrich, A. Ijspeert, and E. Sanchez. An FPGA dynamically reconfigurable framework for modular robotics. *ARCS'05, System Aspects in Organic and Pervasive Computing, Workshop Proceedings*, pages 83–89, 2005.

[217] A. Upegui, C. A. Peña-Reyes, and E. Sanchez. A methodology for evolving spiking neural-network topologies on line using partial dynamic reconfiguration. In *ICCI - International Conference on Computational Intelligence*, Medellin, Colombia, 2003.

[218] A. Upegui, C. A. Peña-Reyes, and E. Sanchez. An FPGA platform for on-line topology exploration of spiking neural networks. *Microprocessors and Microsystems*, 29(5):211–223, 2005.

[219] A. Upegui, C. A. Pena-Reyes, and E. Sanchez. A functional spiking neuron hardware oriented model. In *Computational Methods in Neural Modeling I, LNCS*, volume 2686, pages 136–143. Springer-Verlag, 2003.

[220] A. Upegui, C. A. Pena-Reyes, and E. Sanchez. A hardware implementation of a network of functional spiking neurons with hebbian learning. In *Biologically Inspired Approaches to Advanced Information Technology, LNCS*, volume 3141, pages 233–243. Springer-Verlag, 2004.

[221] A. Upegui and E. Sanchez. Evolving hardware by dynamically reconfiguring Xilinx FPGAs. In *Evolvable Systems: From Biology to Hardware, LNCS*, volume 3637, pages 56–65, 2005.

[222] A. Upegui and E. Sanchez. Evolving hardware with self-reconfigurable connectivity in Xilinx FPGAs. In *Proceedings of the 1st NASA /ESA Conference on Adaptive Hardware and Systems(AHS-2006)*, pages 153–160, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[223] A. Upegui and E. Sanchez. On-chip and on-line self-reconfigurable adaptable platform: the non-uniform cellular automata case. *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS06)*, page 206, 2006.

[224] F. van den Bergh and A.P. Engelbrecht. Training product unit networks using cooperative particle swarm optimisers. In *Proceedings of the International Joint Conference on Neural Networks, IJCNN'01*, volume 1, pages 126–131, 2001.

[225] G.K. Venayagamoorthy and V.G. Gudise. Swarm intelligence for digital circuits implementation on field programmable gate arrays platforms. *Proceedings of the 2004 NASA/DOD Conference on Evolvable Hardware : Seattle, Wasshington, July 24-26, 2004*, pages 83–86, 2004.

[226] K. Vinger and J. Torresen. Implementing evolution of FIR-filters efficiently in an FPGA. *Proceedings of the 2003 NASA/DOD Conference on Evolvable Hardware : Chicago, Illinois, July 9-11, 2003*, pages 26–29, 2003.

[227] M. D. Vose. *The simple genetic algorithm : foundations and theory*. Complex adaptive systems. MIT Press, Cambridge, Mass., 1999.

[228] P. Waldeck and N. Bergmann. Dynamic hardware-software partitioning on reconfigurable system-on-chip. In *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC'03)*, Calgary, Alberta, Canada, 2003.

[229] J. Williams. MicroBlaze uClinux project home page. http://www.itee.uq.edu.au/ jwilliams/mblaze-uclinux/.

[230] M. J. Wirthlin and B. L. Hutchings. DISC: the dynamic instruction set computer. In John Schewel, editor, *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607*, pages 92–103, Bellingham, WA, 1995.

[231] S. Wolfram. Computation theory of cellular automata. *Communications in Mathematical Physics*, 96(1):15–57, 1984.

[232] S. Wolfram. Universality and complexity in cellular automata. *Physica D Nonlinear Phenomena*, 10(1-2):1–35, 1984.

[233] S. Wolfram. *A new kind of science*. Wolfram Media, Champaign, IL, 2002.

[234] Xilinx Corp. MicroBlaze soft processor core. www.xilinx.com/microblaze.

[235] Xilinx Corp. *The programmable Logic Data Book*. San Jose, CA, 1996.

[236] Xilinx Corp. *The XC6200 data sheet v.1.7*. San Jose, CA, 1996.

[237] Xilinx Corp. SystemACE CompactFlash solution, Apr, 2002.

[238] Xilinx Corp. XAPP503: SVF and XSVF file formats for Xilinx devices, Apr, 2002.

[239] Xilinx Corp. EDK OS and libraries reference manual, Aug, 2004.

[240] Xilinx Corp. Virtex-II platform FPGAs: Advance product specification, Dec, 2002.

[241] Xilinx Corp. Virtex-II platform FPGA user guide, March 2005.

[242] Xilinx Corp. XAPP529: Connecting customized IP to the MicroBlaze soft processor using the fast simplex link (FSL) channel, May, 2004.

[243] Xilinx Corp. Virtex-5 user guide, May 2006.

[244] Xilinx Corp. XAPP151: Virtex series configuration architecture user guide, Oct, 2004.

[245] Xilinx Corp. XAPP290: Two flows for partial reconfiguration: Module based or difference based, Sept, 2004.

[246] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[247] X. Yao and T. Higuchi. Promises and challenges of evolvable hardware. *IEEE Transactions on Systems Man and Cybernetics Part C-Applications and Reviews*, 29(1):87–97, 1999.

[248] M. Yim, D.G. Duff, and K.D. Roufas. PolyBot: a modular reconfigurable robot. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 1, pages 514–520, San Francisco, CA, USA, 2000.

[249] M. Yim, Y. Zhang, K. Roufas, D. Duff, and C. Eldershaw. Connecting and disconnecting for chain self-reconfiguration with PolyBot. *IEEE-ASME Transactions on Mechatronics*, 7(4):442–451, 2002.

[250] H. Ying and G. Chen. Necessary conditions for some typical fuzzy systems as universal approximators. *Automatica*, 33(7):1333–1338, 1997.

[251] R. Zebulum, D. Gwaltney, G. Hornby, D. Keymeulen, J. D. Lohn, and A. Stoica. *Proceedings of the 2004 NASA/DOD Conference on Evolvable Hardware : Seattle, Washington, July 24-26, 2004*. IEEE Computer Society, Los Alamitos, Calif., 2004.

[252] R. Zebulum, C. Santini, H. Takahiro, M. Pacheco, M. Vellasco, and M.. Szwarcman. A reconfigurable platform for the automatic synthesis of analog circuits. *Proceedings of the Second NASA/DoD Workshop on Evolvable Hardware (EH'00)*, page 91, 2000.

[253] Y. Zhang, S. Smith, and A. Tyrrell. Digital circuit design using intrinsic evolvable hardware. *Proceedings of the 2004 NASA/DOD Conference on Evolvable Hardware : Seattle, Wasshington, July 24-26, 2004*, pages 55–62, 2004.

Curriculum Vitae

PERSONAL DATA

Name	**Andres Upegui**
Birth	Medellín, Colombia, March 18, 1976
Nationality	Colombian
Prof. Address	Reconfigurable Digital Systems Group (RDSG) School of Computer and Communication Sciences - IC Ecole Polytechnique Fédérale de Lausanne (EPFL) Station 14 CH-1015 Lausanne Switzerland
Prof. Phone	(+41-21) 693 6714
Prof. Fax	(+41-21) 693 3705
E-mail	andres.upegui@epfl.ch
URL	http://lslwww.epfl.ch/˜upegui

EDUCATION

- Ph.D. in Computer Science [Expected: September 2006]
  Ecole Polytechnique Fédérale de Lausanne - EPFL, Lausanne, Switzerland.

- Graduate School in Computer Science
  Ecole Polytechnique Fédérale de Lausanne - EPFL, Lausanne, Switzerland, 2002

- Ingeniería Electrónica (Electronic engineering)
  Universidad Pontificia Bolivariana, Medellín, Colombia, 1999

ACADEMIC POSITIONS

- PhD Candidate and teaching assistant
  Logic Systems Laboratory (LSL)
  Ecole Polytechnique Fédérale de Lausanne - EPFL
  Lausanne, Switzerland, January 2003 - September 2006

- Assistant Instructor
  Universidad Pontificia Bolivariana
  Medellin, Colombia, September 2002 - December 2002

- Fellowship holder
  Ecole Polytechnique Fédérale de Lausanne - EPFL
  Lausanne, Switzerland, October 2001 - July 2002

- Research Assistant & Assistant Instructor
  Universidad Pontificia Bolivariana
  Medellin, Colombia, January 2000 - October 2001

OTHER PROFESSIONAL EXPERIENCE

- Consultant
  Accelogic LLC, FL - USA
  Hardware designer and developer.
  Jul/05 - Nov/05

RESEARCH INTERESTS

- Bio-inspired systems
- Bio-inspired hardware
- Evolvable hardware
- Evolutionary computation
- Adaptation, learning, and intelligence
- Autonomous robotics
- Digital systems design
- Processor Architectures
- Optimization Architectures

LANGUAGES

- Spanish: Native tongue
- English: Very good
- French: Very good

PUBLICATIONS

1. Andres Upegui and Eduardo Sanchez, *"Evolvable FPGAs"*, book chapter in "Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation", edited by Scott Hauck and Andre DeHon, to appear.
2. Arnaud Lagger, Andres Upegui, and Eduardo Sanchez, *"Self-Reconfigurable Pervasive Platform for Cryptographic Aplication"*, in Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL-06), to appear, Madrid - Spain, 2006.
3. Rico Moeckel, Cyril Jaquier, Kevin Drapel, Elmar Dittrich, Andres Upegui, and Auke Jan Ijspeert. *"Exploring adaptive locomotion with YaMoR, a novel autonomous modular robot with Bluetooth interface"*, Industrial Robot, vol 44, n. 4, pages 285-290, 2006.
4. Andres Upegui and Eduardo Sanchez, *"Evolving Hardware with Self-Reconfigurable Connectivity in Xilinx FPGAs"*, in Proceedings of the 1st NASA /ESA Conference on Adaptive Hardware and Systems(AHS-2006), Istanbul - Turkey, pages 153-160, IEEE Computer Society, 2006.
5. Jorge Peña, Andres Upegui, and Eduardo Sanchez, *"Particle Swarm Optimization with Discrete Recombination: An Online Optimizer for Evolvable Hardware"*, in Proceedings of the 1st NASA /ESA Conference on Adaptive Hardware and Systems(AHS-2006), Istanbul - Turkey, pages 163-170, IEEE Computer Society, 2006.
6. Andres Upegui and Eduardo Sanchez, *"On-chip and On-line Self-Reconfigurable Adaptable Platform: the Non-Uniform Cellular Automata Case"*, in Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS06), Rhodes - Greece, page 206, IEEE Computer Society, 2006.
7. Andrés Upegui, Carlos Andrés Peña-Reyes, and Eduardo Sanchez. *"On-line Topology Exploration of Spiking Neural Networks on FPGAs"*. Microprocessors and Microsystems, vol 29 (2005), p 211-223.
8. Andres Upegui and Eduardo Sanchez, *"Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs"*, in Proceedings of the International Conference on Evolvable Systems (ICES 05), Sitges- Spain, September 2005.
9. Rico Moeckel, Cyril Jaquier, Kévin Drapel, Elmar Dittrich, Andres Upegui, and Auke Ijspeert, *"YaMoR and Bluemove - an autonomous modular robot with Bluetooth interface for exploring adaptive locomotion"* in 8th International Conference on Climbing and Walking Robots (CLAWAR 05), London - UK, September 2005.
10. Grégory Mermoud, Andres Upegui, Carlos-Andres Peña, and Eduardo Sanchez *"A Dynamically-Reconfigurable FPGA Platform for Evolving Fuzzy Systems"* International Work-conference on Artificial and Natural Neural Networks IWANN2005, Villanova i la Geltru - Spain, June 2005.

11. Andres Upegui, Rico Moeckel, Elmar Dittrich, Auke Ijspeert, and Eduardo Sanchez *"An FPGA Dynamically Reconfigurable Framework for Modular Robotics"*. Workshop on Dynamical Reconfigurable Systems at the18th International Conference on Architecture of Computing Systems (ARCS 05). Innsbruck - Austria, March 2005.

12. Andres Upegui, Carlos Andrés Peña-Reyes, and Eduardo Sanchez. *"A hardware implementation of a network of functional spiking neurons with hebbian learning"*. BioAdit - International Workshop on Biologically Inspired Approaches to Advanced Information Technology. Lausanne - Switzerland. January 2004.

13. Andres Upegui, Carlos Andres Pena-Reyes, and Eduardo Sanchez. *"A Functional Spiking Neuron Hardware Oriented Model*. In J. Mira, J. R. Alvarez (eds.), Computational Methods in Neural Modeling, Volume 2686 of Lecture Notes in Computer Science, pp. 136-143, Springer, Berlin, 2003.

14. Andres Upegui, Carlos Andrés Peña-Reyes, and Eduardo Sanchez. *"A methodology for evolving spiking neural-network topologies on line using partial dynamic reconfiguration*. Proceedings of II ICCI - International Conference on Computational Intelligence. Medellin - Colombia. November 2003.

15. Andres Upegui, Ivan Herrera, Hernan Sánchez. *"Integracion del Multiprocesador MADE""*. Proceedings of the VII Workshop Iberchip. Montevideo - Uruguay. March 2001.

16. Ivan Herrera, Hernan Sánchez, Andres Upegui, and Lucas Restrepo. *"Modelo de simulación y Prueba para el Multiprocesador MADE"*. Proceedings of the VI Workshop Iberchip. Sao Paulo - Brasil. 2000.

17. Ivan Herrera, Hernan Sánchez, Andres Upegui, and Lucas Restrepo. *"MADE Multiprocessor"*. Proceedings of the 1st National Symposium of electronic and telecommunication research. Universidad de Los Andes. Bogotá - Colombia. November 1999.

PATENT APPLICATIONS

1. J. Gonzalez, A. Upegui, R. Nunez, "Customized functional parallel algorithms for global optimization",US patent app. 60/767128, filed 03/06/2006.

2. J. Gonzalez, A. Upegui, "Method and apparatus for increased parallel processing scalability in global optimization algorithms". US patent app. 60/767129, filed 03/06/2006.

3. J. Gonzalez, A. Upegui, R. Nunez, "Fast sorting algorithms for global optimization problems". US patent app. 60/767130, filed 03/06/2006.

INVITED LECTURES, TALKS, TUTORIALS, AND POSTERS

- *”High Performance Non- von Neumann Algorithms For Large-Scale Optimization”*, Poster at the ”31st Dayton-Cincinnati Aerospace Sciences Symposium”, Dayton-OH, USA, March 2006.
- *”Evolving Artificial Neural Networks”*, Invited lecture, - Prof. G. Tempesti and Prof. A. Ijspeert course ”Genetic and developmental computing architectures”, EPFL, Lausanne, Switzerland, December 2005.
- *”Tutorial in Dynamic Partial Reconfiguration of Xilinx FPGAs”*, Tutorial given at the ”Seminario Internacional sobre sistemas dinamicamente reconfigurables”, Universidad de Antioquia and UPB, Medellin, Colombia, Febrary 2005.
- *”Dynamic Partial Reconfiguration”*, Invited lecture, Prof. Eduardo Sanchez course ”Conception avancée de systèmes numériques”, EPFL, Lausanne, Switzerland, March 2004 and 2005.
- *”Evolving Neural Networks”*, Invited lecture, Prof Daniel Mange’s course ”Systèmes et programation génétiques”, EPFL, Lausanne, Switzerland, January 2004 and 2005.
- *”Topology Exploration of Spiking Neural Networks on FPGAs”*, Tutorial at ICCI - 2003, ”International Congress on Computational Inteligence”, Medellín, Colombia. November 6-8, 2003.
- *”Topology Evolution of Spiking Neural Networks”*, Talk at the ”Ciclo de conferencias sobre inteligencia artificial”, Rama estudiantil IEEE. Universidad Pontificia Bolivariana, Medellin, Colombia. November 5, 2003.
- *”Reconfigurable Computing”*, Lecture at the Course on ”Advanced Digital System Design”, Processor Architecture Laboratory (LAP), Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland. October 6-10, 2003.
- *”A functional spiking neuron hardware oriented model”*. Poster at the University Booth - SIGDA. In: 40th DAC. Anaheim, CA, USA. June 2003.
- *”Microboard: a low-cost prototyping platform for reconfigurable computing”*, presented at "INGENIAR ’01: Muestra de trabajos de ingenieria". UPB, Medellin-Colombia. October 2001.
- *”Seminar on FPGA based design”*. Course given at the Universidad Pontificia Bolivariana, Bucaramanga, Colombia. September 2001.
- *”Integration of the Event Driven Multiprocessor”*. Poster at the University Booth - SIGDA. In: 38th DAC. Las Vegas, NV, USA. June 2001.
- *”MADE Multiprocessor and Microboard”*, Poster at the University Booth - SIGDA. In: 37th DAC. Los Angeles, CA, USA. June 2000.

AWARDS AND SCHOLARSHIPS

- ”2005 President’s Innovator Award” delivered by *Intellectual Property Systems* in recognition of the contributions for the advancement of the company’s technological competitive advantage. FL-USA, November, 2005.
- ”The Industrial Robot Highly Commended Award” in the *8th International Conference on Climbing and Walking Robots (CLAWAR 2005)*. London-UK, September, 2005.
- ”Scholarship for the Graduate School in Computer Science” delivered by the *EPFL*. Lausanne-Switzerland, 2001-2002.
- ”Travel grant award”, granted by the *Special Interest Group on Design Automation (SIGDA)* for attending the *University Booth* at the *Design Automation Conference (DAC)*. Los Angeles-USA, 2000; Las Vegas-USA, 2001; Anaheim-USA, 2003.
- ”2nd Best project award: Muestra de Trabajos de Ingenieria”, delivered by the *UPB in INGENIAR ’01*. Medellin-Colombia, 2001.
- ”Scholarship for the course Interfacing Microsystems” granted by *Iberchip*. Montevideo-Uruguay, 2001.

REVIEW ACTIVITIES

Occasional reviewer for the following international journals:

- Microprocessors and Microsystems, *Elsevier*
- Neurocomputing, *Elsevier*
- Transaction on Neural Networks, *IEEE Computational Intelligence Society*
- Transaction on Neural Systems and Rehabilitation Engineering, *IEEE Engineering in Medicine and Biology Society*

STUDENT PROJECTS SUPERVISED

- Jorge Peña: *”On-chip and On-line Adaptive Hardware Using Particle Swarm Optimization with Discrete Recombination”* Master Thesis, ALaRI - University of Lugano, 2006
- Arnaud Lagger: *”Self-Reconfigurable Platform for Cryptographic Application”*, Master Thesis, EPFL, 2006
- Adamo Maddalena: *”YaMoR II - The Second Generation of the YaMoR Modular Robot”*, Master Thesis, EPFL, 2006
- Jerome Maye: *”Bluetooth Configuration of an FPGA: An Application to Modular Robotics”*, Semester Project, EPFL, 2005

- Gregory Mermoud: *"A Dynamically-Reconfigurable FPGA Platform for Evolving Fuzzy Systems"*, Semester Project, EPFL, 2005
- Cyril Jaquier, Kevin Drappel: *"Bluemove: Using Bluetooth to Control a YaMoR Modular Robot"*, Semester Project, EPFL, 2005.
- Christian Kunzler: *"Algorithmes d'Apprentissage pour des Réseaux de Neurones à Impulsions"*, Semester project, EPFL, 2005
- Valentin Longchamp: *"Linux for MicroBlaze, an FPGA-Based Platform"*, Semester project, EPFL, 2005.
- Rico Mockel: *"Design and Construction of an Autonomous Modular Robot Unit with Bluetooth and FPGA"*, Summer Internship, EPFL, 2004
- Elmar Dittrich: *"Modular Robot Unit - Characterization, Design and Realization"*, Summer Internship, EPFL, 2004.
- Jorge Peña, Jerónimo Castrillón, Daniel Giraldo: *"Aprendizaje por Refuerzo en Espacios Continuos para la Evasión de Obstáculos en un Robot Móvil"*, Diploma Project, UPB, 2004.
- Nicolas Frey: *"Linux for PowerPC, an FPGA-Based Platform"*, Semester project, EPFL, 2004.
- Barthélemy von Haller: *"Implémentation d'un Réseau de Neurones Reconfigurables sur FPGA avec JBits"*, Semester project, EPFL, 2004.
- Edouard Goupy, Guillaume Du Pasquier: *Synchronization of Nonlinear Oscillators via Bluetooth*, Semester Project, EPFL, 2003.
- Alejandra Gallón, Juan Carlos Vélez, Alex Echavarria: *"Chip para Adquisición y Edición de Video"*, Diploma Project, UPB, 2001.