A Practical Investigation into Achieving Bio-Plausibility in Evo-Devo Neural Microcircuits Feasible in an FPGA

Hooman Shayani

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University College London.

Department of Computer Science

University College London

2013

I, Hooman Shayani, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

_____

_____

This work is dedicated to nobel, courageous, and selfless men and women at BIHE (Baha'i Institute for Higher Education) who sacrificed everything, even their lives, to reclaim the right to education of many Baha'i youths in Iran.

Abstract

Many researchers has conjectured, argued, or in some cases demonstrated, that bio-plausibility can bring about emergent properties such as adaptability, scalability, fault-tolerance, self-repair, reliability, and autonomy to bio-inspired intelligent systems. Evolutionary-developmental (evo-devo) spiking neural networks are a very bio-plausible mixture of such bio-inspired intelligent systems that have been proposed and studied by a few researchers. However, the general trend is that the complexity and thus the computational cost grow with the bio-plausibility of the system. FPGAs (Field-Programmable Gate Arrays) have been used and proved to be one of the flexible and cost efficient hardware platforms for research and development of such evo-devo systems. However, mapping a bio-plausible evo-devo spiking neural network to an FPGA is a daunting task full of different constraints and trade-offs that makes it, if not infeasible, very challenging.

This thesis explores the challenges, trade-offs, constraints, practical issues, and some possible approaches in achieving bio-plausibility in creating evolutionary developmental spiking neural microcircuits in an FPGA through a practical investigation along with a series of case studies. In this study, the system performance, cost, reliability, scalability, availability, and design and testing time and complexity are defined as measures for feasibility of a system and structural accuracy and consistency with the current knowledge in biology as measures for bio-plausibility. Investigation of the challenges starts with the hardware platform selection and then neuron, cortex, and evo-devo models and integration of these models into a whole bio-inspired intelligent system are examined one by one. For further practical investigation, a new PLAQIF Digital Neuron model, a novel Cortex model, and a new multicellular LGRN evo-devo model are designed, implemented and tested as case studies. Results and their implications for the researchers, designers of such systems, and FPGA manufacturers are discussed and concluded in form of general trends, trade-offs, suggestions, and recommendations.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Peter Bentley, for careful support, great advice, many many helpful insights, and teaching me to be ambitious in my research. He is not only a very good PhD advisor but also a very good friend for his students.

I wish to thank my parents for their support and patience, my father who was the first one who instilled in me a passion for science and engineering, and my mother who alway encouraged me to pursue my goals. I would like to thank my wife, Shadi, for encouragement and support before and during my studies. Without her this work would have never been even started. I am grateful to my son, Adib, for making this PhD a more tolerable journey with his playful smiles and lively laughs.

I would also want to thank my external supervisor, Prof. Andy Tyrrell, for the great help and advice during the publications, and Prof. John Shawe-Taylor, Prof. Anthony Finkelstein, and Dr. Rob Smith for all the help and support during the course of my research, and finally, my examiners Dr. Julian Miller, and Dr. Stuart Flockton, for their helpful comments and for patiently noting many typographical errors in the thesis.

This work is gratefully dedicated to nobel, courageous, and selfless men and women at BIHE (Baha'i Institute for Higher Education) who sacrificed everything, even their lives, to reclaim the right to education of many Baha'i youths in Iran including me. Many of these people are still wrongfully in prison merely for providing higher education to those who are barred from it without any acceptable reason. I will always be indebted to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nature has always inspired man. We imitate nature to find solutions to our problems even with a minimum knowledge of the underlying principles. However, our engineered solutions are not the same as natural solutions. Otto Lilienthal for example, one of the pioneers of aviation, was clearly inspired by the way birds fly [220], but we do not see a Boeing 747 flying with moving feathery wings. A B747 has fixed metallic wings filled with aviation fuel to power its jet engines. Our engineered designs, though inspired by nature, are constrained and formed by our current technology. Similarly, the brain is clearly the source of inspiration in the report by John von Neumann, one of the pioneers of computing, where he first describes the basic architecture of today's computers [394]. The brain is arguably the most complex natural organ with parallel processing, distributed memory, stochastic computing, self-organisation, self-regulation, autonomy, learning, fault-tolerance, robustness and many other amazing features [51, 73, 200, 264]. Yet, von Neumann's architecture is ultimately a centralised, synchronous, sequential, precise and brittle design, which he, himself, was dissatisfied with until he died [393, 395]. His design was again a product of the engineering trade-offs, technological constraints, dominant mentality of his time [22], and his knowledge of neuroscience, and had in fact very little in common with how the brain works. Now, with the emergence of the new technologies such as reconfigurable electronic devices and advancements in biology and neuroscience, we have a better chance than before to bridge this gap between computers and the brain. Yet again we have these trade-offs. How similar should we make our designs to the biological solutions? What are these trade-offs? What is the right balance? And how to achieve that balance? It is the aim of this thesis to answer some of these questions.

Computation and computer architectures frequently challenge our ability to find the right balance between natural inspiration and engineering design. It is a common observation that almost everything in nature appears to be computing something [22]. Brains, immune systems, embryogenesis, evolution, swarming insects, and ecosystems, every one of these natural systems and phenomena has remarkable features that are very useful if we could reproduce and exploit them in our engineered computing solutions. Many pioneers in computing such as von-Neumann, Alan Turing, and Claude Shannon were well aware of these features, and were seeking to reproduce them into engineering designs (For example, Von-Neumann's self-reproducing automata [393], and Turing's chemical morphogenesis [368]). That is exactly what Bio-inspired Computing (Biologically Inspired Computing) is pursuing today.

Bio-inspired computing imitates natural processes such as evolution, development and learning and biological systems and organs such as the brain, chromosomes, and immune systems in order to replicate some of the features that are very useful in today's computing systems; properties such as adaptability, scalability, robustness, parallelism, distributed and stochastic processing, self-organisation, self-regulation, autonomy, fault-tolerance, regeneration, and self-repair.

Computing systems can benefit from these properties in many different ways. The intrinsic parallelism, and distributed processing of the bio-inspired computing solutions may simplify scalability challenges in multiprocessor and distributed systems and bring about robustness and fault tolerance. Some of these features may help computing systems in changing environments or tackling ill-defined problems. The no-free-lunch theorem [403] implies that for both static and time-dependent problems, there is no single algorithm that can perform better than all other algorithms on all problems. This is particularly evident in the case of traditional approaches to computing, which have difficulties solving natural problems such as learning, pattern recognition, optimisation, and automatic design using fixed, precise and deterministic algorithms. They can show an acceptable performance only on a very limited range of problems. For example, a statistical or heuristic object detection algorithm may perform very well for detecting faces in input images but generally cannot be easily adapted for detecting hands, tools or distorted faces.

In contrast, natural systems and to some extent bio-inspired computing solutions can adapt themselves to perform very well on a wider range of natural problems in changing environments. The main advantage of the natural systems is their intrinsic adaptability. Adaptability is the generic feature of the natural systems that manifests itself in different aspects and over various time scales. Looking closely, almost all other advantageous properties of the natural systems are different forms of adaptation. Natural systems are scalable, meaning that they can grow (through their life-time or evolve through generations) to employ more resources in order to tackle more difficult problems or deal with larger amount of work than before. They are fault-tolerant. That is to be able to cope with resource loss or failure and to regenerate (reorganise resources) to recover. Natural systems are robust, which means they are capable of graceful degradation in case of a change in their environment. Natural systems evolve, develop, regenerate, and learn to adapt to their changing environment. This constant and open-ended adaptation to an ever-changing ecosystem full of other adaptive systems brings about ever more complex self-sufficient systems that do not need human design, analysis or maintenance. The automatic design, self-sufficiency and autonomy are attractive features that may address the challenges of the ever-increasing complexity of the electronic systems.

1.1 Bio-plausibility

The bio-inspired design process involves answering two important questions: Which processes and structures in the inspiring natural system give rise to the desired features? To what extent do the quality of these features depends on the details of those processes and structures? Looking at a natural system from a hierarchical point of view, these questions lead to a principal question in design of bio-inspired systems: which level of abstraction is the right level in modelling of natural systems? Since natural

systems are evolved and developed in a bottom-up manner, each layer depends on the lower layers and it is reasonable to assume that all the details are more or less relevant to the general function of the system. For example, the interactions between the atoms in the process of gene expression contribute a lot to higher-order processes of development and evolution and it is beneficial to include them in a model of the evolution. However, it is physically impossible to include all these details in a bio-inspired system. Thus some abstractions and therefore some structural inaccuracies are inevitable. Moreover, we do not have a complete understanding of all the details in biology. Therefore, it not only matters how abstract and inaccurate our model of reality is but also how plausible our model is from a biologist's point of view. We can call this combined measure of inaccuracy and the plausibility of the abstracted model "bio-plausibility" for short.

While many researchers have pointed out that bio-inspired computing needs to go beyond simplistic, superficial and heavily abstracted models of natural processes and a higher level of bio-plausibility is beneficial [16, 365, 89], the complexity and the massive processing power and resources needed for such detailed models stop them from creating such bio-plausible systems. In practice, designers need to trade bio-plausibility of the model off against feasibility factors such as size, speed, energy consumption, heat dissipation, reliability, cost, and time-to-market. However, the emergence and development of new technologies such as multicore processors, FPGAs (Field-Programmable Gate Arrays), and GPUs (Graphics Processing Units) is pushing back the boundaries of feasible bio-plausible models and there are numerous new things to explore. This thesis focuses on the challenges and some potentials of creating a bio-plausible intelligent system similar to the brain using such new technologies.

1.2 An Artificial Brain in Silicon

Intelligent systems are used in all aspects of day-to-day life, from fraud detection, market forecasting and epidemic prediction to HCI (Human-Computer Interaction) and bio-informatics. You can find intelligent controllers in our car engines, washing machines, and robotic vacuum cleaners. Many of these intelligent systems are based on the traditional von Neumann-Turing model of computing, which is seriously challenged by the parallel, stochastic, ever-changing nature of some real-life problems and environments. While von Neumann-Turing paradigm lends itself to precise calculations based on fixed and deterministic algorithms for well-defined problems and processes, it is not quite suitable for solving intractable and ill-defined problems in changing environments [13, 423, 356]. Globalisation, climate change and the rise of the natural disasters have created a very dynamic and erratic social, economical, and physical environment with many unpredictable trend shifts. Intelligent systems used in such a dynamic environment need to be adaptable. Otherwise, every time new trends emerge in their environment, they may simply fail or need adjustment, revision or even redesign. Imagine a company that has developed a market forecasting system for decades and suddenly, in the middle of an unprecedented global financial crisis, when needed the most, it fails. Moreover, it is rendered useless, as it can never adapt to the new financial regime and requires costly radical changes. Or, consider a mobile network traffic management system that at the time of a natural catastrophe collapses and leaves people incommunicado. A bio-inspired adaptable intelligent network management system distributed over the network nodes or hubs may bene-

fit from a small local disaster to learn how to deal with a global catastrophe. Adaptability of bio-inspired computing is a very desired feature for such situations.

Among different fields of bio-inspired computing aiming at creating intelligent systems by imitating the brain, neural networks are one of the most famous topics. The first models of artificial neural networks proved to be too simplistic and not similar to the brain [108]. Nevertheless, they inspired statisticians and computer scientists to develop very successful non-linear statistical models and learning algorithms that comprise an important part of the machine learning techniques today [108]. Biologically more plausible models of neural networks such as recurrent spiking neural networks proved to be even more useful and even more promising in processing spatiotemporal data than traditional artificial neural networks [241]. Such recurrent networks can be very robust to noise and other changes in the environment and have an intrinsic fault-tolerance [241]. Evolving such networks using evolutionary computing can adapt them to a changing environment or optimise them to enhance the solution for specific problems. Adding bio-plausible details of neurodevelopment to the evolutionary system can bring about features such as fault-tolerance, self-organisation, regeneration, and self-repair and meanwhile may improve the evolvability and scalability of the system [128, 311]. We can call such a system, which incorporates three natural processes of learning, development and evolution, an evolutionary developmental (evo-devo for short) recurrent spiking neural network. Nevertheless, including all these complexities necessitates a very powerful yet malleable hardware platform.

There are many promising hardware technologies for implementing such evo-devo systems. Among them, maybe FPGAs (Field Programmable Gate Arrays) are the most practical solution for different reasons. An FPGA is essentially a pool of fundamental digital circuit elements that can be reconfigured in numerous ways to implement virtually any possible digital circuit. While they inherit the maturity of the digital VLSI silicon fabrication technology, they are particularly best suited to the asynchronous, parallel and distributed nature of bio-inspired computing. It is possible to achieve fine-grain parallelism with a network of small specific-purpose processors on a single FPGA, which allows much higher computational throughput than multi-core general-purpose processor architectures. FPGAs allow different distributed memory architectures to avoid memory bottlenecks typical of GPU (Graphics Processing Unit) architectures. While FPGA clock speeds, power consumptions, and capacities are no match for those of ASICs (Application Specific Integrated Circuits), their highly parallel architectures, short time-to-market and low NRE (nonrecurring engineering) costs make them very appealing for many applications including research. They can be dynamically reconfigured to modify one part of the circuit while the rest of the circuit is running. This is a very desirable feature for implementing a developmental regenerative neural network. Moreover, the intrinsic fault-tolerance of such a neural network can only appear in a truly distributed and parallel architecture, which is possible on an FPGA. Therefore, the focus here is on the implementation of evo-devo recurrent spiking neural networks on FPGAs. We call such an evolutionary developmental recurrent spiking neural network implemented in an FPGA an “evo-devo neural microcircuit” for short.

All the promised features of an evo-devo neural microcircuit, if achievable, are highly desirable in

different intelligent systems. Fault-tolerance is particularly needed in mission critical applications such as military, medical or aerospace systems where uninterrupted and reliable service is required. In cases such as satellites or other remote facilities where maintenance costs are very high, environments are unpredictably changing, and facilities have a long life cycle, a low-maintenance, fault-tolerant, robust and adaptable system can be very cost effective. Scalability of such systems allows them to exploit more hardware resources with a minimum effort. The possibility of automatic customisation of the system for specific problems or environments in the first place using evolution is another advantage of such system over fully human-designed systems. Depending on the accuracy of the models, such a bio-plausible neural system can also be used as a simulator in neuroscience research endeavours.

1.3 Research Problem

New advances in hardware technologies and better knowledge of neuroscience and neurodevelopment calls for a reassessment of the promise and challenges of achieving bio-plausibility using such new technologies. Digital hardware technologies are advancing very fast and the feasibility criteria are always changing and need to be updated. Figure 1.1 illustrates an abstraction of the expected general trade-off between feasibility and bio-plausibility in bio-inspired digital systems and how this trend might be changing by new digital technologies. The vertical axis of the graph represents the bio-plausibility of bio-inspired models from bio-irrelevant to bio-accurate. The horizontal axis represents all different dimensions of simulation feasibility (cost, speed, scale,...) as one dimension. Each curve shows the upper bound of feasibility and bio-plausibility using a specific generation of digital technology. On the extreme left of the graph, non-mathematical conceptual models reside that are completely impossible to simulate. Therefore, the existence of these models of natural systems are only bounded by current biological knowledge. This is demonstrated by the asymptote on the left-hand-side of the graph. Some famous brain simulation projects are depicted on the graph as examples.

The new findings of the neuroscience have also changed the landscape of bio-plausible models. Although, bio-plausible models of learning are still under study and the neurodevelopment process is still not very well understood, there are new discoveries that can be used to create speculative but plausible and useful models of learning and neurodevelopment processes.

Bio-plausibility has been viewed from a limited angle by many researchers in the field of bio-inspired computing. Bio-plausibility can have a broader sense, notably when it is viewed in the light of the interaction between different natural processes of development, evolution, learning, and all of them within an environment [157]. Usually in evolutionary computing an agent is viewed as a solution to a problem while biological agents are actually embodied in their challenging environments. Moreover, some models focus only on the bio-plausibility of the neuron model. Some extend this to the synapse model and even bio-plausible learning techniques. Very few studies have actually investigated the effect of using a bio-plausible evolutionary neurodevelopment model along with a bio-plausible neural model [191, 190, 192].

The prohibitive computational cost of having a bio-plausible evolutionary developmental algorithm on top of a neural model has practically precluded many researchers from investigating such

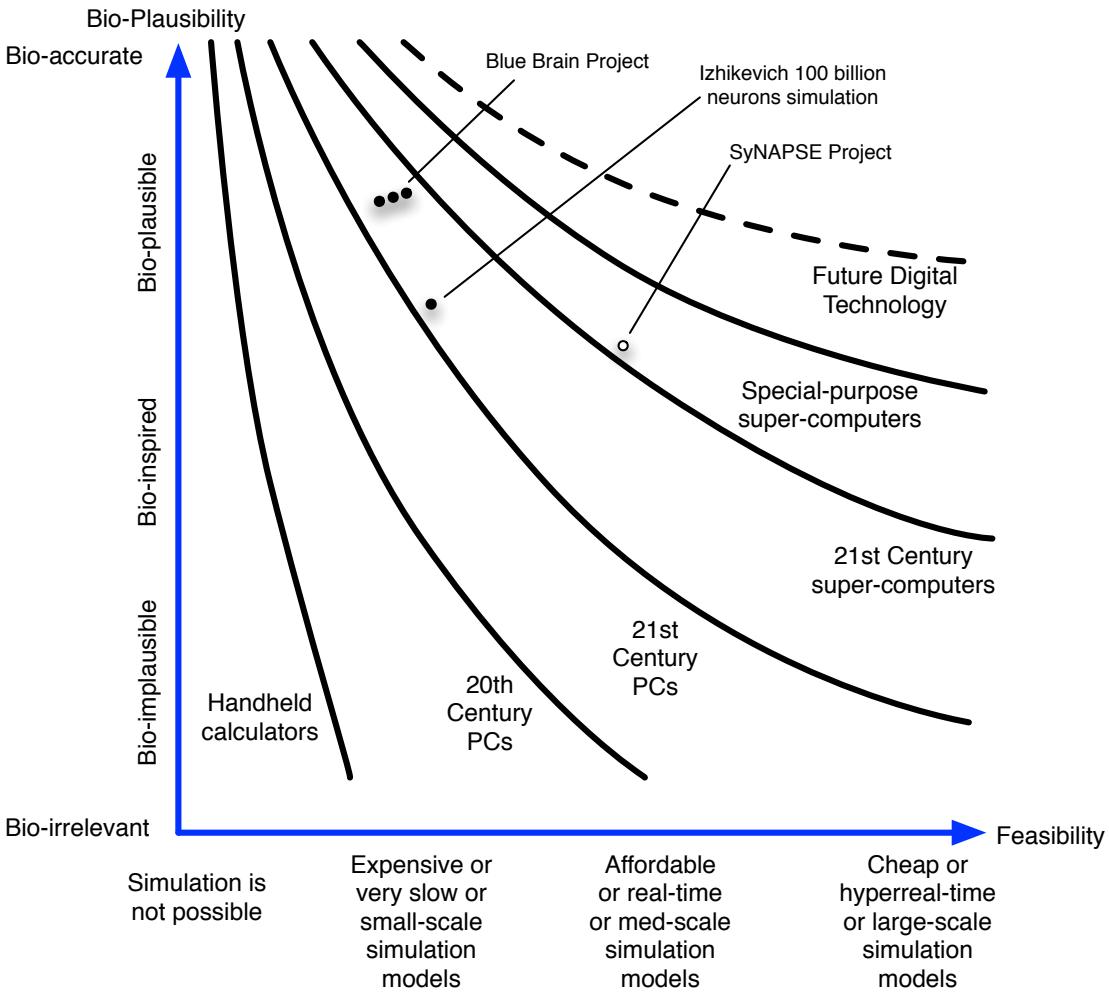


Figure 1.1: A schematic graph showing the trade-off between feasibility and bio-plausibility of bio-inspired systems and the expected effect of new technologies in changing this trend. The horizontal axis represents all the different dimensions of the simulation feasibility (cost, performance, scale,...) as one dimension. Each curve shows the upper band of feasibility and bio-plausibility using a generation of digital technology. Some famous brain simulation projects are depicted on the graph as examples

all-embracing bio-plausible models and those that succeeded did not have hardware (and specifically FPGA) implementation constraints in mind. There is certainly a need for a holistic reconsideration of the broader meanings of bio-plausibility in evo-devo neural microcircuits. It is only recently that POE systems have been studied in this light [370]. Nevertheless, for economical reasons, custom POEtic chips [6] are not practical and mature compared to FPGAs. Only small POEtic chips have been manufactured in small numbers or sub-optimally prototyped on FPGAs [310, 6].

Despite many advantages of FPGAs, they are not designed for evo-devo neural microcircuits. There are many challenges in conforming bio-inspired applications to FPGA architectures. It is to be investigated how to use current reconfigurable devices such as FPGAs effectively to achieve a better balance between feasibility and bio-plausibility, and how future reconfigurable devices can be designed to suit such bio-inspired applications. This also requires a better understanding of these trade-offs in the design of such systems.

The history of GPUs teaches us that it makes sense to begin with commercially ubiquitous and relatively cheaper platforms before designing custom chips. GPUs, fuelled by the gaming industry, were ubiquitous and relatively cheap and programmers started to use them as highly parallel processors for general computations. This led to GPGPU (General-Purpose computing on Graphics Processing Units) and emergence of standards, programming tools, and custom GPGPU devices later [143, 317, 316]. Aiming for higher levels of bio-plausibility on commercially available chips might similarly lead to the new design ideas, standards, potentials, and market force for large investments in custom bio-plausible chips.

Achieving all these interesting qualities in a bio-plausible evo-devo neural microcircuit is a daunting task that requires a plethora of exploration, investigation and experimentation with different models of neurodevelopment, evolution and learning. Moreover, these models must be more bio-plausible than models that researchers have already accepted as useful or feasible. The process of running an evo-devo neural microcircuit involves iterative nested loops of evolution, development, simulation and learning over a diverse training set from a problem class. This can be even more time consuming in an experimental setting where there are dozens of parameters to tune and many different techniques to investigate. There are many promises for amazing nature-like features in digital systems that justify the effort to explore the new landscape of feasibility and bio-plausibility of neural-microcircuits on the actual hardware. This work focuses on the investigation of the bio-plausibility of evo-devo neural microcircuits in FPGAs in a broader sense, highlighting the challenges of that level of bio-plausibility, and studying the trade-offs and constraints involved in the design of such systems in FPGAs during the design, implementation, and testing of such a system on a selected commercial FPGA.

1.4 Aim

The aim of this thesis is **to investigate the challenges of achieving bio-plausibility in evo-devo neural microcircuits feasible in an FPGA.**

1.4.1 Definitions

The word "model" is used throughout this work to refer to a simulated model of a biological system as an engineered solution that runs directly, or by means of software implementation, on digital hardware and does not refer to a theory used in scientific method or a simulation model that is used for development and testing of hypothesises unless explicitly stated.

A biologically plausible (bio-plausible for short) model is defined here as a bio-inspired model that "does not require unrealistic computations" [320, 400] and assumptions that are inconsistent with the current knowledge of the actual mechanism in the target biological system. Bio-plausibility is a qualitative property of a model that is weaker than structural accuracy, which requires strong evidence of similarity between the mechanisms in the model and the target system [Webb2001]. Bio-plausibility is discussed and defined in section 2.1 in detail.

A neural microcircuit is defined, in this thesis, as a bio-plausible heterogeneous recurrent neural network comprised of different types of neurons directly mapped to hardware, with no unrealistic assumptions for the neural coding. This terminology is used to contrast this type of neural networks with simplistic neural models that are homogeneous, feed-forward, non-spiking or loosely mapped to hardware or are completely implemented in software.

An evo-devo neural microcircuit is such a neural microcircuit that is evolved using evolutionary algorithms with a bio-plausible developmental genotype-phenotype mapping.

Feasibility is measured by different factors that make a solution practical as an engineered solution or as a platform for research toward such solutions. Factors can include design time and complexity, cost, speed, size, scalability, accessibility, and fabrication constraints in respect to the current technological limits in FPGAs and, when possible, those of foreseeable future technologies. These factors are discussed and defined in section 2.2 in detail.

1.5 Scope

The investigation is carried out through practical design, feasibility study and implementation of a case study solution. Analysis is used where different implementations or conducting many experiments are not feasible. This work focuses on the challenges caused by contradiction between hardware restrictions of FPGAs and requirements of an evo-devo system. It also investigates the possible trade-offs between speed, size, scalability, fault-tolerance, reliability, robustness, design time and complexity, and bio-plausibility, and seeks a balanced point in the design space suitable for a relatively bio-plausible evo-devo solution as the case study. The field of evo-devo neural microcircuits is still very young. It is too early for an exhaustive study of all the feasibility factors, bio-plausibility aspects, and trade-offs. Therefore other feasibility factors such as power consumption, heat dissipation, reliability, and human and economic factors are considered out of the scope of this thesis. However, some of these factors are inevitably assessed briefly during the research project planning. Similarly, this work focuses on investigation of some aspects of bio-plausibility and their trade-offs during the case study. The importance and effects of different factors and their relations are studied in a qualitative way. This is due to the novelty

of the field and lack of enough groundwork in the field (such as standards, metrics, and benchmarks) for a theoretical analysis or an experimental approach with statistical evidence. It is the aim of this work to provide the insight that can lead to more rigorous studies of the subject.

This work is a study of some of the significant possible design choices and solutions that are deemed relevant for the goal of achieving bio-plausibility in FPGA-based evo-devo neural microcircuits. Despite the existing standard design procedures and practices, the bio-inspired digital design process has remained a creative, artistic, and non-linear process to some extent. There is always a possibility that an ingenious design can radically change a trade-off equation resulting in a better solution or approach. However, we anticipate that the experience gained in this case study will provide insights for future designers of such systems.

1.6 Objectives

This research is aimed at providing insight into the challenges of achieving bio-plausibility in evo-devo neural microcircuits in FPGAs through analysis and practical design of a case study solution. To accomplish this aim, the following objectives are defined for the project:

1. Defining bio-plausibility and feasibility in the context of the bio-plausible evo-devo neural microcircuits in FPGAs

2. Reviewing the state of the art in FPGAs and related technologies, current bio-plausible models of the brain and neurodevelopment, and studies similar to this work

3. Investigating the challenges, constraints, and trade-offs in the hardware platform selection

4. Assessing the challenges, options, trade-offs, and constraints involved in the design, implementation, testing and, evaluation of a bio-plausible neuron model suitable for an evo-devo system on an FPGA

5. Assessing the challenges, options, trade-offs, and constraints involved in the design, implementation, testing and, evaluation of a bio-plausible reconfigurable structure on FPGA suitable for evo-devo neural microcircuits

6. Assessing the challenges, options, trade-offs, and constraints involved in the design, implementation, testing and, evaluation of a bio-plausible neurodevelopmental evolutionary model for growing neural microcircuits in FPGAs

7. Assessing the challenges, options, trade-offs, and constraints involved in the integration, end-to-end testing, and evaluation of a bio-plausible evo-devo neural microcircuit system

1.7 Publications

Different steps of this study have been peer reviewed, published and presented in relevant international conferences. A list of papers already published based on this work follows:

- Shayani, H. Bentley, P.J. and Tyrrell, A.M. (2009) A Multi-cellular Developmental Representation for Evolution of Adaptive Spiking Neural Microcircuits in an FPGA, International NASA/ESA Conference on Adaptive Hardware and Systems, 3-10, San Francisco, July, 2009.

- Krohn, J., Bentley, P. J., and Shayani, H. (2009) The Challenge of Irrationality: Fractal Protein Recipes for PI. In Proc of the Genetic and Evolutionary Computation Conference (GECCO 2009). July 8-12, 2009.

- Shayani, H., Bentley, P. J. and Tyrrell, A. (2008) A Cellular Structure for Online Routing of Digital Spiking Neuron Axons and Dendrites on FPGAs. In Proc of The 8th International Conference on Evolvable Systems: From Biology to Hardware (ICES 2008). Prague, September 21-24, 2008.

- Shayani, H., Bentley, P. J. and Tyrrell, A. (2008) Hardware Implementation of a Bio-Plausible Neuron Model for Evolution and Growth of Spiking Neural Networks on FPGA. In Proc of NASA/ESA Conference on Adaptive Hardware and Systems. IEEE Computer Society CPS. pp. 236-243.

- Shayani, H., Bentley, P. J. and Tyrrell, A. (2008) An FPGA-based Model suitable for Evolution and Development of Spiking Neural Networks. In Proc of 16th European Symposium on Artificial Neural Networks, Advances in Computational Intelligence and Learning. Bruges (Belgium), 23-25 April 2008. pp. 197-202.

1.8 Thesis Structure

The following chapter starts with a definition of bio-plausibility and feasibility (objective 1) and continues with a review of the relevant literature (objective 2). Chapter 3 discusses the hardware platform selection and its challenges (objective 3). Chapters 4 to 6, focus on the challenges, options and trade-offs in the design, implementation, testing and evaluation of bio-plausible neuron model, reconfigurable structure on FPGA, and evo-devo model (objectives 4-6) respectively. Chapter 7 discusses the integration and testing of the whole system (objective 7). The thesis is summarised and concluded in chapter 8.

Chapter 2

Background

In this chapter, related literature and research are reviewed and the latest FPGA-related technologies, methods, developments, and applications of spiking neural networks, evolvable hardware, hardware based evolutionary neural networks, similar studies, and related subjects are discussed. It starts with two separate sections on the definitions of bio-plausibility and feasibility in the context of this research.

2.1 Bio-plausibility

To be able to accomplish a focused literature review on feasibility of bio-plausible systems, first, it is essential to define bio-plausibility and feasibility as a ground for comparison in our specific context. In its original context of biomedicine, biological plausibility is the consistency of a hypothetical causal relationship with the current biological and medical knowledge about that relationship [159]. The same terminology and its shorter version - bio-plausibility - has been used in the context of modelling, simulation, robotics and artificial life to refer to the similarity of the behaviour and mechanism underlying the behaviour of a model, simulator, robot or artificial agent with the existing biological knowledge about those of the actual natural systems [115, 400].

In [400], Webb classifies different aspects of models of natural systems in seven dimensions:

1. Biological relevance: It shows if this model can be used to generate and test hypothesises about an identified biological system. This is important when the model is used for biological study rather than as an inspiration in engineering. From a biomedical and biological standpoint bio-plausibility can refer to the biological relevance of a model.

2. Level: What are the basic elements of the model that have no internal structure or their internal structures are ignored? For instance, a model can be based on the atoms and modelling their interactions while ignoring the internal structure of the atoms or might be a very high-level model that ignores all the internal structure of societies and only focuses on the interaction between societies.

3. Generality: How many different biological systems can be represented by this model? For example, a neural model can be used only to model a specific type of the biological neuron in human brain but another model is expected to represent different type of mammalian neurons. As different researchers has pointed out [400] this could be a result of higher level, abstraction, or even

detail in modelling, which might actually lead us to a significant finding in biology or a useful general solution in engineering.

4. Abstraction: The complexity of the model compared to the biological system and the amount of detail included in the model. Without abstraction modelling does not make sense. More abstract means less detail, fewer and simpler mechanisms than the target system. Abstraction makes the model understandable in science and feasible in engineering. This abstraction may or may not lead to generality. However, usually, general abstract models are interesting and very useful. Abstraction should not be confused with the level. Apart from the level of modelling, the complexity of a model also depends on how a modeller achieves the same behaviour in the model. For example, a high-level model of cognitive process in the brain may be more complex than a brain model based on the ion channel properties of neuron membranes, while both are showing the same behaviour.

5. Structural accuracy: The similarity of the mechanism behind the behaviour of the model to that of the target biological system. This is directly affected by our current knowledge of the actual mechanisms in biological systems. This is not necessarily proportional to the amount of details included in the model, as these details also need to be correct to contribute to the accuracy of the model. Similarly, accuracy is not directly related to the level of the model. For example, a high-level model could be very accurate up to that level while a very low-level model could be quite inaccurate on many levels. In [400] Webb explains that bio-plausibility can refer to the accuracy of a model.

6. Performance match: The similarity of the behaviour of the model to that of the target biological system.

7. Medium: The physical medium that has been used to implement the model.

On the matter of biological plausibility and its definitions, Webb mentions that "biological plausibility" is widely used to say that a model is "applicable to some real biological system"; or to refer to the biological accuracy of the assumptions that the model is based on. These definitions overlap with both biological relevance and structural accuracy in the above classification. It can also describe that the model "does not require biologically unrealistic computations" and is consistent with the current knowledge of the actual mechanism in the target biological system [320]. Webb prefers this latter interpretation of "plausibility" that weakly relates bio-plausibility to the structural accuracy of the model when there are not very strong reasons for accuracy but at the same time it is not implausible that the actual mechanism in the target biological system is similar to the model and compatible with its assumptions.

It must be noted that bio-plausibility and all the above dimensions can be viewed from two different perspectives:

1. Modelling as a tool in biology for developing theories and hypothesises and testing them [256];

2. Designing biologically inspired systems in an engineered application [103].

Webb acknowledges the distinction between these two methods and focuses on the former. However, it is clear that the dimensions she has introduced are very useful in bio-inspired engineering as well. From the definitions of the dimensions and the discussions in [400], it is clear that these dimensions are interrelated and not necessarily orthogonal. For example, while the complexity (abstraction) of the model largely depends on the modelling approach, it also depends on the level. The generality of a model can be increased by adding details (complexity) such as parameters, or by a valid abstraction that reflects the general properties of a group of biological targets. Figure 2.1 shows some of the interrelations between Webb's modelling dimensions that can be concluded from [400].

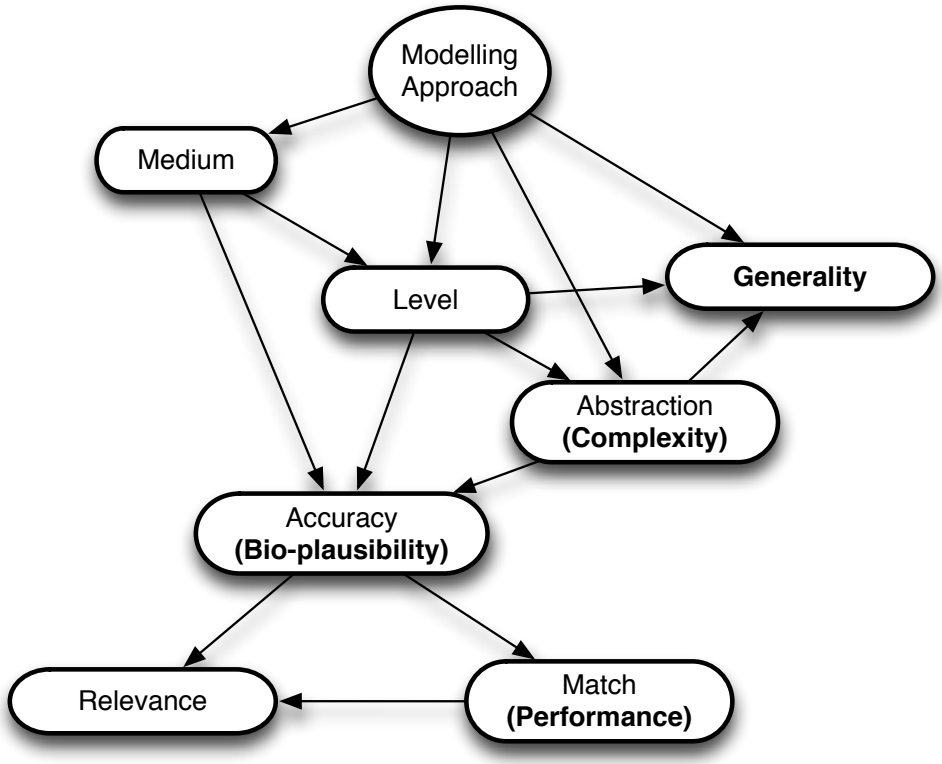


Figure 2.1: Some of the main interrelations between Webb's modelling dimensions concluded from [400].

In the former view to modelling, a certain level of relevance is necessary. This requires a certain level of performance match and accuracy. The goal is to use the abstraction to form an understandable hypothesis or test it. This may or may not lead to generality as a desired feature, which depends on the modelling approach and the fact that such generality existed in a group of target systems in the first place. A modeller chooses the medium, level, and abstraction in a way that a certain level of accuracy, and thus relevance, is reached, which makes the model useful in biology.

In contrast to this method, in the bio-inspired engineering approach to modelling, a certain level of performance is required and accuracy (bio-plausibility) is the means to match that performance. Abstraction contributes to the feasibility of the model as it reduces the complexity. Generality is again a desired extra feature of the model that may or may not be attained. In the bio-inspired engineering approach, medium, level, and abstraction are chosen in a way that the required performance is attained while the

model is kept feasible in terms of complexity.

We define bio-plausibility, in bio-inspired engineering context, using the same two definitions from Webb's taxonomy [400], as "**structural accuracy**" when detailed knowledge about the internal structure of the target system is available, or as "**consistency with the current knowledge**" when such details are not discovered yet. Focusing on the brains, neural microcircuits and neurodevelopment and evolution, it is clear that the current knowledge about the target system is far from adequate in many areas. However, the second definition of bio-plausibility, based on the consistency with the available knowledge, can be used effectively in those areas.

2.2 Feasibility

As the second important factor in this work is the feasibility of the bio-plausible models, it is imperative to define feasibility in such a way that allows comparison of different models in the literature. Feasibility is originally a binary measure that shows if a system is practical or impractical to design, build or use. It is affected by a set of different constraints. By sufficient relaxation of these constraints any solution would become feasible. Some of these constraints can be found in digital and embedded systems design and engineering literature [397, 202, 90, 98, 297]. In classical digital design or integrated circuit design textbooks, there is no mention of feasibility. They refer to quality measures of digital designs instead. For example [297] divides quality measures of digital integrated circuits into:

1. Costs

   Fixed or non-recurring costs: This is mainly design cost that is a function of the complexity, specification aggressiveness, productivity of the designer(s) plus indirect overhead of the company or laboratory.

   Variable cost: This is the cost of each manufactured product, which mainly consists of a quintic function of the silicon die area that is related to the complexity as well.

2. Functionality and robustness - reliability of the product

3. Performance - This is the computational power of the system. This depends on the latencies of the components and the maximum clock frequencies, etc.

4. Power and energy consumption: this is the amount of energy that the circuit needs to consume and relatively the heat that must be dissipated from the circuit.

Similar factors such as performance, cost, size, security, reliability, scalability, and power consumption are considered for computer systems in computer architecture and organisation textbooks [345, 288, 153]. In the general context of all different digital design approaches, the most important of all these factors are flexibility, performance, cost of ownership and running cost. Cost of ownership is mainly dominated by die area. The running cost is also mainly proportional to energy consumption. It is possible to asses different design approaches using power and area efficiency figures based on performance-cost ratios of milliWatt per Million Operations Per Second (mW/MOPS) and MOPS

per each square millimetre (MOPS/mm^2) [278]. Figure 2.2 from [278] shows how different design approaches are distributed over the 2D space of power and area efficiency. A general trend of increasing flexibility is also evident as we move from optimised ASICs (Application Specific Integrated Circuit) towards general-purpose processors. Figure 2.3 from [38] depicts the relation of flexibility, power dissipation and performance. Blume *et. al.* used reciprocal of the time needed for a new design for quantifying the flexibility in [38]. Figure 2.3 clearly demonstrates why using FPGAs can be a balanced solution to the flexibility-performance tradeoff.

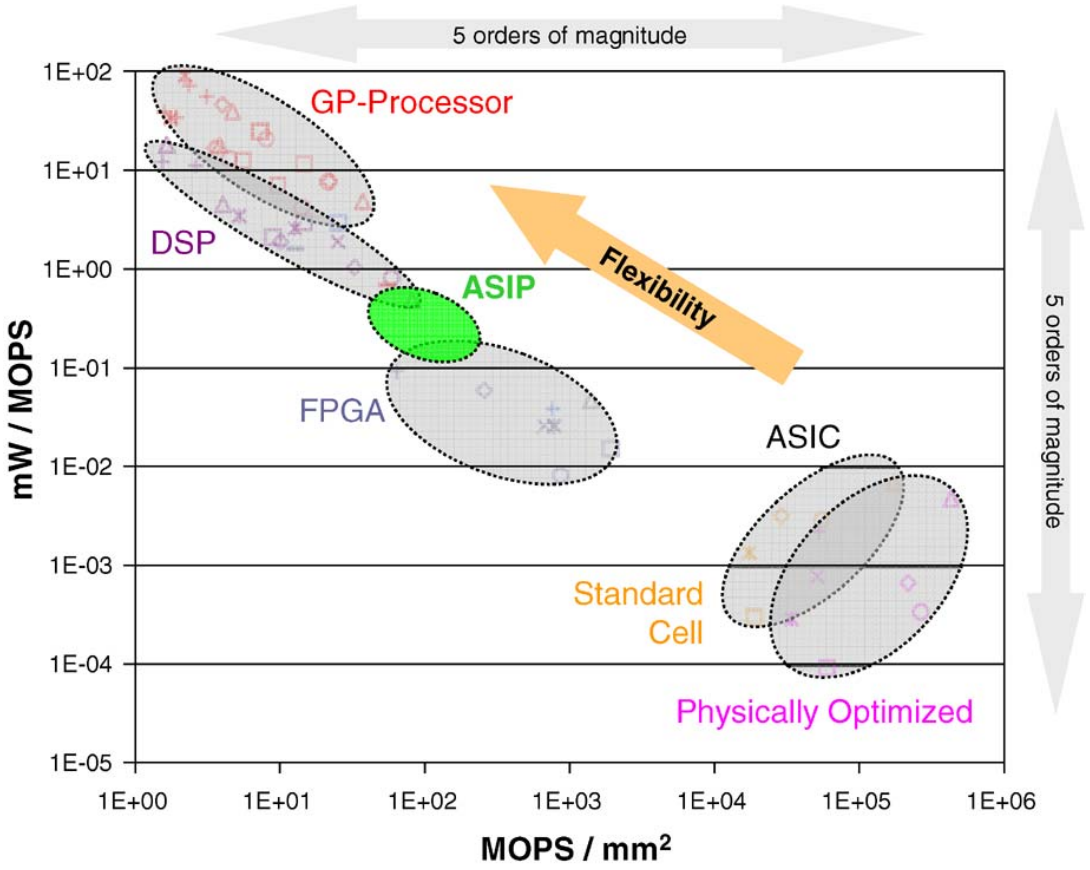


Figure 2.2: From [278], energy and area efficiency of different digital design approaches.

It is also possible to consider all these factors as different types of costs functions:

1. Design costs (Non-Recurring Engineering - NRE costs, a function of the design time and complexity)

2. Implementation costs (material and labour costs, implementation time, and complexity, availability)

3. Testing costs (time, complexity, accessibility)

4. Life-cycle costs (power consumption, heat dissipation, space, time, weight, reliability, maintenance, flexibility, scalability, performance, accessibility)

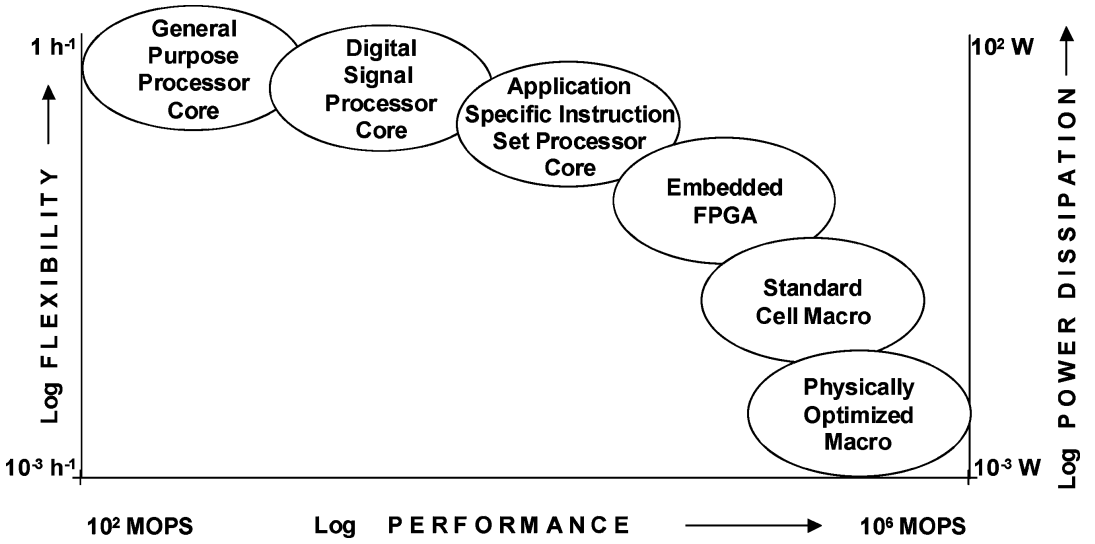


Figure 2.3: From[38], flexibility, power dissipation and performance of different digital design approaches. Flexibility is measured in the reciprocal of the design time.

The closest field to this study in the literature, discussing these factors and the tradeoffs between them, is reconfigurable computing [57, 278]. Although literature in this field cover many benefits of using FPGAs and reconfigurable computing, they mostly lack a review of the effects of bio-inspired approaches on these factors. Most of the studies talk about the power wall issue that stops manufacturers increasing the density of the power dissipation endlessly and how reconfigurable hardware can mitigate this problem. They also talk about performance benefits of parallelism and fault-tolerance in reconfigurable platforms. Not many studies focus on the potentials of using bio-inspired techniques at extremes towards solving these problems [371]. For example, the general tradeoff between performance and energy consumption in digital systems is a known fact and using many low-power processors in parallel as a scalable solution to the power wall issue has been suggested before [113]. However, the brain uses many billions of much slower and extremely low-powered processing elements (neurons and synapses) in parallel resulting a much higher efficiency. Another example is the limitation of the silicon die size in IC manufacturing. The probability of a defect in a larger die increases with die size, which drastically reduces the yield ratio [297]. Using effective bio-inspired techniques can bring fault-tolerance and self-repair to digital systems allowing much larger, denser, and cheaper integrated circuits [371, 283]. In the more specific context of spiking neural network simulation some studies look into these tradeoffs with emphasis on scalability and flexibility [113] rather than bio-plausibility. Schrauwen *et. al.* studied the tradeoff between scalability, area, and performance for a not very bio-plausible neuron model [330]. Tyrrell *et. al.*[371, 283] have also studied different aspects of the bio-inspired reconfigurable computing on custom devices with focus on fault-tolerance. There appears to be no readily available study that specifically focuses on the tradeoffs in the design of bio-plausible evo-devo neural microcircuits on FPGAs.

For the purpose of this study, feasibility is defined in the context of bio-inspired neural microcircuits in FPGAs. In this context, feasibility shows how tight are the constraints that can be satisfied by a design as an engineered product or as a platform for research towards such product. However, instead of looking for the whole Pareto front in the whole design space [90], this work focuses on the most important factors in this context, looking for potential new frontiers in the useful regions of the design space. Therefore, here, feasibility is mainly measured based on these seven factors:

1. Hardware cost (inversely proportional to compactness)
2. Performance (simulation and evolution speed)
3. Scalability (number of neurons, synapses, ...)
4. Design time and complexity (inversely proportional to simplicity)

   also includes flexibility (for research purposes)
5. Testing time and complexity (inversely proportional to simplicity)

   also includes observability (for debugging, testing, and research)
6. Availability
7. Reliability (robustness, fault-tolerance, etc.)

2.3 Field Programmable Gate Arrays (FPGAs)

A Field Programmable Gate Array (FPGA) [309] is a digital integrated circuit that consist of many Configurable Logic Blocks (CLBs), configurable input/output blocks, configurable routing blocks and many wires connecting these resources. Figure 2.4[30] shows the general concept of the FPGA architecture. In this figure, routing blocks are shown as switch boxes and connection boxes. Each CLB mainly comprises a few Look-Up-Tables (LUTs), flip-flops, adders, and multiplexers that can be configured to create a small combinational or sequential digital circuit. LUTs can be reconfigured in different modes to form shift registers or small RAM blocks as well. All these reconfigurable blocks can be electronically programmed (reconfigured) by a user or a designer after manufacturing to create virtually any arbitrary digital circuit. Recent devices have distributed static RAM and FIFO blocks, multiplier blocks, DSP (Digital Signal Processing) blocks, Hi-speed IO blocks, processors cores, different types of other hard IP Cores (Intellectual Property Cores - a predesigned module), and higher number of configurable logic blocks (CLBs) compared to the previous models. New devices with up to two million logic cells (approximately equal to 30 million gates) are already available off-the-shelf [412]. According to Moore's law even faster and larger FPGAs are on their way.

Although FPGAs have many disadvantages compared to ASICs, they proved to be useful and cost-effective in many areas. FPGAs are more expensive per chip as they need more silicon area, consume more power and offer lower clock rates than ASICs. An empirical study on different designs on FPGAs and ASICs [212] shows that on average a circuit on an FPGA needs 35 times more area, is 3.4 to

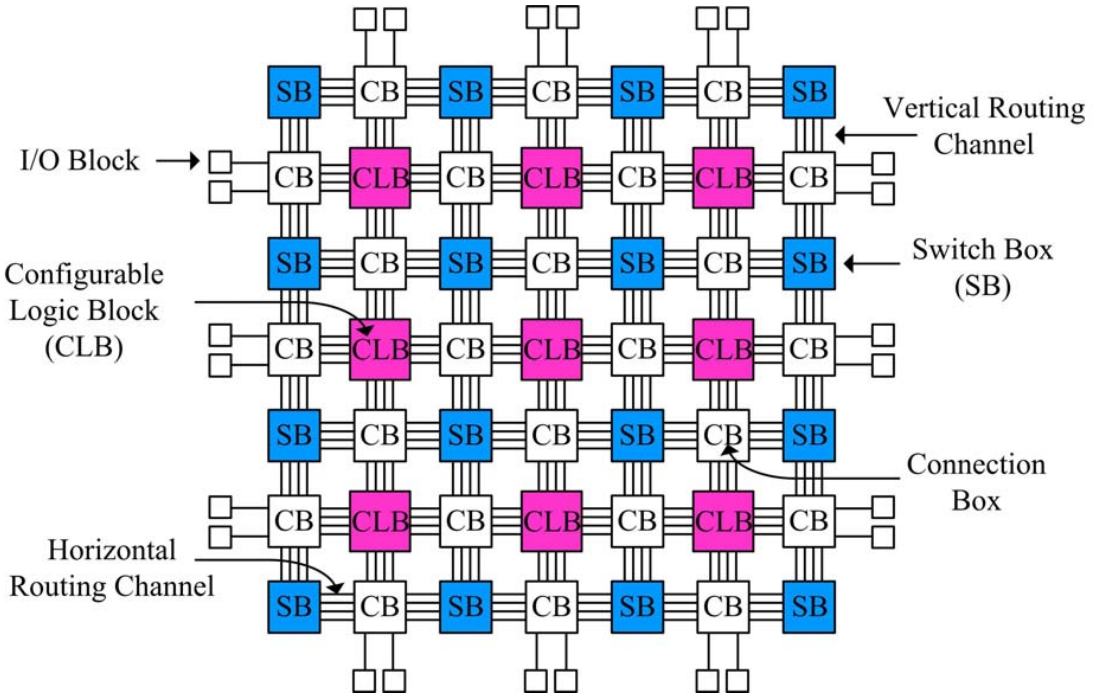


Figure 2.4: General conceptual of FPGA architecture [30]

4.6 times slower and consumes 14 times more power than an equivalent standard-cell implementation on ASIC. Nevertheless, the possibility of field-reconfiguration simplifies debugging and updating the design, and significantly reduces the time-to-market and Non-Recurring Engineering (NRE) costs, which make FPGAs popular particularly in low-volume applications and research. They have been successfully used in different applications [309] such as digital signal processing [333], reconfigurable computing, communication processing [141], and rapid system prototyping of ASIC designs [309]. They are also very useful in research and are commonly accepted as the main digital platform for intrinsic evolvable hardware[131].

There are two major FPGA vendors and competitors, Xilinx and Altera, that according to Wikinvest, together share more than 80% of the fast growing FPGA market. They are followed by Lattice Semiconductors Inc. with 11%. This is clearly a duopoly that indicates a matured industry and market. Other vendors mostly focus on non-SRAM-based (e.g. anti-fuse or Flash-based) FPGAs. New developments such as 3D FPGAs, stacking and time-multiplexed FPGAs [285] show a promising horizon.

2.3.1 Design

Although FPGA design tools and techniques and workflows are similar to those of standard-cell ASICs to some extent, each vendor has provided its own vendor-specific design suite for design and reconfiguration of their devices. For example, Xilinx provides the traditional ISE Design Suite that allows users to design using HDLs (Hardware Description Languages) and other high-level system descriptions, simulate, debug, and synthesise their designs, map, pack, place and rout them, verify them, and finally, generate reconfiguration bitstreams. Vendors provide designers with a set of IP cores (including

soft-core processors) to speed up and simplify the design process. Quite recently, Xilinx also introduced Vivado Design Suite that also allows designers to use C, C++ and System-C source codes as specifications of IP cores. Altera provides Quartus II as the design software solution. Design suites from both companies can work closely with third-party design, simulation, and synthesis tools such as Matlab and Simulink from Mathworks, and LeonardoSpectrum and ModelSim from Mentor Graphics.

2.3.2 Reconfiguration

Reconfiguration process [309] involves interfacing with the internal reconfiguration controller of the chip and writing configuration data into the configuration memory that controls the connections and functionality of the reconfigurable blocks. As the configuration memory is SRAM based and volatile, every time the FPGA is powered-up, the chip needs to be reconfigured. This is performed by sending a bitstream (a binary file) to the configuration controller of the chip through a port. FPGAs usually have different internal and external ports and modes for accessing the configuration memory. For example, Virtex-5 family of Xilinx FPGAs apart from supporting the standard serial JTAG/Boundary-scan port, have a serial/parallel port with master or slave mode called SelectMAP that can be used in different ways [411]. They also have an Internal Configuration Access Port (ICAP) that can be used by FPGA to reconfigure itself or read/verify its internal state [411]. For an FPGA to be able to reconfigure itself, it needs to support two other features: partial reconfiguration and dynamic reconfiguration.

2.3.3 Partial Reconfiguration

Partial Reconfiguration (PR) [20, 309, 411, 184] refers to the alteration of the state of only part of the configuration memory, thus functionality of a portion of the circuit, without touching the rest. This significantly reduces the length of the reconfiguration bitstream and reconfiguration delay [357]. Some FPGAs are capable of running without interruption while they are being partially reconfigured. This is known as DPR (Dynamic partial reconfiguration) or Run-Time Reconfiguration [309, 219, 184]. This feature makes it possible for part of the FPGA to reconfigure the rest of it. This is very useful as it allows designers to swap modules that are not used simultaneously[20]. Partial reconfiguration and dynamic partial reconfiguration are also very useful in evolvable hardware and bio-inspired designs as it speeds up the reconfiguration process and allows a circuit to adapt and develop in real time [54, 184, 373]. Only recently, Altera started to support dynamic and partial reconfiguration in its new 32nm FPGAs such as Stratix V [8]. Until few years ago Xilinx was the only major manufacturer of SRAM based FPGAs capable of partial dynamic reconfiguration.

Xilinx supports two design flows for PR (partial reconfiguration) using ISE design tools: module-based PR and difference-based PR [219, 406]. In module-based flow the swappable modules in the design are positioned in large blocks at specific locations in the FPGA and connected by interfacing resources called Bus Macros to the rest of the design in order to fix interfacing lines in place to guarantee that all the lines will be connected properly after reconfiguration [219, 184]. Module-based PR bitstreams contain only the reconfiguration data for the block that contains the module. In difference-based PR, the modification is usually very small and is affecting only a few places in the configuration memory. This technique can be used to modify the content of a block RAM or an LUT (look-up-table)

that changes the functionality of the circuit. In this case the bitstream contains the minimum number of frames (smallest reconfigurable data unit) needed to reconfigure those parts of the configuration memory and reconfiguration process is very fast [219, 406]. This is done by manually making small changes in the design using Xilinx's FPGA editor tool and saving the bitstream and then generating the difference-based bitstream by comparing before and after bitstreams using tools provided by Xilinx [406]. Both of these work flows are only useful when there are limited number of predefined compatible modules to swap or a predefined set of minor modifications needed.

For run-time and versatile reconfigurations, as needed in evolvable hardware, the reconfiguring agent (e.g. PC or an embedded processor) needs to generate bitstreams on-the-fly. This requires complete knowledge of the bitstream file formats. Although the general structure of the Xilinx bitstreams are well documented and released, the low-level specification of bitstream files for new families of Xilinx FPGAs are proprietary and not released. To address this need, Xilinx introduced JBits and JRoute APIs (Application Programming Interfaces) and a set of tools (called XHWIF) that allow reconfiguration of Virtex devices using these Java libraries and interfacing tools [279, 341]. However, these tools are not open source or properly maintained by Xilinx and they never supported any other FPGA devices beyond Virtex II [279]. Later, Xilinx introduced driver libraries for the OPBHWICAP and XPSHWICAP IP cores that can be used along with on-chip processor cores (e.g. MicroBlaze) to perform some of the useful partial reconfiguration tasks such as modifying LUT contents or flip-flop states in real-time. Unfortunately, both these drivers and the IP core are very limited in speed and functionality and are not portable to unlicensed processors. Other IP cores with orders of magnitude higher speed than original XPSHWICAP have been designed and benchmarked by different researchers for Xilinx Virtex II Pro, Virtex-4, and Virtex-5 family of devices [77, 226, 144, 31]. However, these cores are generally designed for module and difference-based PR and do not appear to necessarily work with Xilinx drivers for versatile reconfigurations such as LUT content modifications [77, 226, 144, 31].

Some attempts to reverse-engineer the bitstream file formats in order to directly generate or manipulate bitstreams have been very promising [279, 372]. It has been shown [279] that by using Xilinx tools and some statistical and logical inference it is possible to reverse engineer the bitstream file formats.

2.4 Neural Networks

Nature's solution to create an adaptable and embodied intelligent agent is a nervous system or a biological neural network [103]. Nervous systems mainly consist of two types of cells: neurons and glial cells. Neurons are the main processing elements [186, 73, 103, 121, 239, 150]. They consist of a body, called soma, with relatively long extensions called dendrites and axons. Dendrites are essentially the inputs of a neuron that gather all the signals from other neurons, mix them and send them to the soma. A single axon, which may also divide into branches at the end, sends the output of the neuron to other neurons or actuators (e.g. muscles) through small electrochemical devices called synapses at the contact point of the axons with dendrites or cell bodies. Glial cells provide support and nutrition for neurons and act as "glue" between them [213]. Recently, they were suspected to be also involved in the synapse formation as well as axon and dendrite development [293, 213, 67].

Neurons communicate by sending electrical pulses called Action Potentials (APs) or spikes [186]. These are electrochemical waves that travel through axons and when they reach to the synapses release special molecules, called neurotransmitters. These neurotransmitters can open very small gates, called ion channels, on the surface of the dendrite on the other side of the synapse. This allows electrically charged molecules, called ions, to pass the dendrite membrane and change the electrical potential across the membrane. These dendrite potentials mix and interact in complex ways and consequently affect the membrane potential of the soma [149]. Soma membrane is also covered with different types of ion channels that are sensitive to this voltage across the membrane. When the membrane potential goes higher than some level it affects more ion channels leading to an ion rush and a rapid increase (depolarisation) and then a quick drop (repolarisation) of the membrane potential that initiates an action potential that travels down the axons as a spike. There are different types of neurons with slightly different behaviours. Some neurons, called inhibitory neurons, release neurotransmitters that decrease or block the activation of other neurons. There are about 850,000 neurons in the brains of honeybees while the human brain comprises about 10^{11} neurons [113]. Neurons in the human brain are usually connected to 1000 to 10000 other neurons [113]. Neurons can fire (spike) up to 250 to 300 times a second [103].

2.4.1 Artificial Neural Networks

Artificial Neural Networks (ANN) or Neural Networks for short, are a set of bio-inspired computational models of the function or structure of the brains and nervous systems that are simulated in computer software or custom-designed hardware. They usually comprise a directed graph with a number of nodes (cells) representing the neurons, and many connections (links) representing the axons, dendrites and synapses between neurons[37, 103, 358].

Despite the extensive studies and brilliant achievements in using artificial neural networks, they have so far not been as successful as their biological counterparts [37, 358]. This could be partly due to extensive abstractions and over-simplifications in the artificial neural network models. The McCulloch-Pitts model [254], and sigmoid threshold neurons [37] are two classical neuron models of this kind in the literature. Limiting the architectures to feed-forward networks, modelling complex electrochemical signals and processes with relatively simple equations, and neglecting temporal dynamics of the signals and processes, are all examples of such simplifications to name a few. The field of neural networks has been largely formed by these type of simplistic rate models that neglect the timing and temporal dynamics of the neurons and networks [37, 358]. Recently, after many critiques and "hype cycles" and consequently periods of suspension in research, this field is attracting attention again thanks to introduction of new bio-plausible models that take into account some of the complexities of the biological neural networks. Using spiking neuron models and temporal coding have been proven to result in computationally more powerful networks [239]. Reservoir Computing is also a new bio-plausible method for design and training of recurrent neural networks that has been very successful in spatiotemporal pattern recognition [332, 390]. Hierarchical Temporal Memory (HTM) is another recent bio-plausible model that has attracted a lot of attention [150, 281]. In the following sections, each one of these new models

and methods are reviewed in more detail.

2.4.2 Spiking Neural Networks

Biological neurons communicate through their axons and dendrites by sending (arguably) identical spikes. While in simpler models (rate models [37]) only spike rates is considered, in Spiking Neural Networks (SNNs) [239, 121] the precise timing of each spike can also convey contextual information. There has been an endless debate about the importance of spike timings and whether only the spiking rate of the neurons matters in the brain. Although the coding scheme of the brain is not completely deciphered yet, there is enough evidence that delay, phase, and synchrony of the spikes can be used for communication in context of a spiking neural network [239, 121, 43].

Spiking neural networks have some advantages over other types of artificial neural network models in terms of computational power and capabilities. Spiking neurons, being more similar to their biological counterparts than rate neurons, are likely to be a good solution for creating embodied intelligent agents, as they are nature's solution to the same set of problems through billions years of trial and error by evolution. Moreover, the bio-plausibility of SNNs allows a mutual transfer of concepts, techniques, and results between neuroscience and artificial intelligence communities [106]. Spiking neural networks have more computational power than other artificial neural networks. Many functions exist that can be implemented by a single spiking neuron but take hundreds of hidden units on a sigmoidal neural network [237]. On the other hand, any function that can be computed on a small sigmoidal neural network can also be implemented using a small spiking one [238]. Even very noisy spiking neural networks can be used for computing a function to an arbitrary level of reliability [236]. Noisy spiking neural networks can simulate sigmoidal neural networks with the same number of nodes but with more computational power [238]. Spiking neurons have short term memory and can use a temporal coding for inter-neuronal communication. This allows them to process time series easily [241]. Temporal nature of the signals and processes in spiking neural networks provides useful information to a local learning process at each synapse. Spike Timing Dependent Plasticity (STDP) has been already observed in biological neural networks through experiment [35] and different timing dependent Hebbian algorithms can be used for unsupervised or reinforced learning. This makes spiking neural networks a very interesting option for implementing embodied agents where a labelled training dataset does not already exist. Moreover, locality of the learning processes allows a fully parallel implementation of the learning algorithm. Furthermore, the event-based nature of the signals with identical spikes makes digital hardware a good candidate for efficient parallel implementation of spiking neural networks [44].

As one of the most bio-plausible families of neural network models to date, spiking neural networks are the focus of this thesis, and different important and therefore useful (in this context) spiking neural network models are reviewed in the following sections.

2.4.3 Spiking Neuron Models

Results of Hodgkin and Huxley's extensive experiments on the giant axon of the squid, which resulted in the award of the Nobel prize in 1963 established a fundamental model of biological neurons [158]. Based on that, more plausible models for artificial neural networks were proposed, which take some

subtleties of biological neurons into account. These models, which are called spiking neuron models (or pulsed neural models), are also used in creating artificial Spiking Neural Networks (SNN) [239].

Hodgkin-Huxley Model

Hodgkin-Huxley neuron model [73, 158] is based on the ionic mechanisms underlying the initiation and propagation of action potentials in the neuron [73, 158]. In this model, the cell membrane is considered as a capacitor with capacitance C. Dynamics of the voltage across the membrane (u) and external driving current $I(t)$ are described by the differential equation [239]:

$$C\frac{du}{dt} = -\sum_k I_k + I(t) \tag{2.1}$$

where $\sum I_k$ is the sum of the ionic currents through the membrane, which consists of sodium and potassium ion channel contributions (indexed by Na and K) and leakage L [239]:

$$\sum_k I_k = g_{Na}m^3h(u - V_{Na}) + g_K n^4(u - V_K) + g_L(u - V_L) \tag{2.2}$$

where g_{Na}, g_K, and g_L are conductances of corresponding ion channels and membrane leakage; V_{Na}, V_K, and V_L are reversal potentials (modelling the diffusive flow of the ions); and m, n, and h are variables, which can be described with three additional differential equations [239]:

$$\begin{aligned} \dot{m} &= \alpha_m(u)(1-m) - \beta_m(u)m \\ \dot{n} &= \alpha_n(u)(1-n) - \beta_n(u)n \\ \dot{h} &= \alpha_h(u)(1-h) - \beta_h(u)h \end{aligned} \tag{2.3}$$

where α and β are in turn empirical functions of u.

As is clear from equations 2.1, 2.2, and 2.3 this is a complex model and computationally demanding [169]. However, it is still the most plausible model that is accepted as the reference model and all other models are based on this or are compared to this model in terms of bio-plausibility [169].

Multi-compartment Models

The Hodgkin-Huxley model is a single-compartment model that ignores the spatial electrical potential and current inside neurons and describes the membrane potential of a neuron by a single variable (u). multi-compartment models [73, 239] consider these details by approximating the shape of the neuron with many uniform cylindrical compartments using cable theory [73]. Computer simulations of a large-scale networks of neurons with these models are computationally intractable.

Leaky Integrate-and-Fire Model (LIF)

One of the most common models used in simulation of the spiking neuron models, specially in hardware implementations, is Leaky Integrate-and-Fire (LIF) model [121]. This is mainly because of its simplicity and because it is computationally cheaper to simulate than other models [169]. A neuron is simply modelled as a capacitor C in parallel with a leakage resistor R and input current $I(t)$. The input current and voltage of the membrane are then governed by equation:

$$I = \frac{u}{R} + C\frac{du}{dt} \tag{2.4}$$

which can be turned into a standard leaky integrator equation with time constant $\tau_m = RC$:

$$\tau_m \frac{du}{dt} = -u(t) + RI(t). \tag{2.5}$$

This equation does not account for the firing. So, a firing condition is added to this equation. When the membrane voltage $u(t)$ becomes greater than a threshold θ, a spike is emitted by the neuron and the membrane voltage is reset to $u_r < \theta$. As this reset value is far below the threshold voltage, neuron will not fire for a while even in presence of input current. This will create a relative refractory period. In a more detailed version of the model, an absolute refractory period can be added by keeping $u(t)$ at u_r for absolute refractory period Δ^{abs} after a firing at time $t^{(f)}$. Then integration restarts at time $t^{(f)} + \Delta^{abs}$.

When a neuron (indexed by j) fires at time $t_j^{(f)}$, it contributes to the input current of the downstream neuron (indexed by i) by $w_{ij}\alpha(t - t_j^{(f)})$ where w_{ij} is efficacy of the synapse between neuron i and neuron j, and $\alpha(s)$ is a pulse function. By adding an external current $I_i^{ext}(t)$ for sensory neurons, the input current for neuron i can be calculated by equation:

$$I_i(t) = \sum_j w_{ij} \sum_f \alpha(t - t_j^{(f)}) + I_i^{ext}(t). \tag{2.6}$$

The $\alpha(s)$ function can considered as Dirac pulse function, $\alpha(s) = q\delta(s)$. A more realistic choice for this function could be an exponential decay function with time constant τ_s:

$$\begin{aligned} \alpha(s) &= \frac{q}{\tau_s} \exp(-\frac{q}{\tau_s})\Theta(s) \\ \Theta(s) &= \begin{cases} 1, & s \geq 0 \\ 0, & otherwise \end{cases} \end{aligned} \tag{2.7}$$

More detailed versions of this model are available, which also include a finite rise time for $\alpha(s)$ and an axonal transmission delay [121]. This model captures the fundamental behaviour of the biological neurons and it is computationally less demanding than previous models. However, many features of Hodgkin-Huxley model are not supported by LIF model [169].

Spike Response Model (SRM)

Since a neuron can be assumed to be reset to the same state after each firing, in Spike Response Model [239, 121], it is possible to calculate the membrane voltage of a neuron using kernel functions (η, ϵ, and κ) of time after the last firing of the neuron ($t - \hat{t}_i$):

$$\begin{aligned} u_i(t) &= \eta(t - \hat{t}_i) + \sum_j w_{ij} \sum_f \epsilon_{ij}(t - \hat{t}_i, t - t_j^{(f)}) \\ &\quad + \int_0^\infty \kappa(t - \hat{t}_i, s) I^{ext}(t - s) ds. \end{aligned} \tag{2.8}$$

The function $\eta(t - \hat{t}_i)$ produces the spike form and the refractory period. The Kernel $\epsilon_{ij}(t - \hat{t}_i, t - t_j^{(f)})$ express the effect of a spike from pre-synaptic neuron j on the membrane voltage of the post-synaptic neuron i. The kernel $\kappa(t - \hat{t}_i, s)$ is response of the membrane voltage to external current I^{ext}. A neuron fires when the membrane voltage $u_i(t)$ exceeds a threshold θ. This model can become equivalent to LIF model for particular kernel functions. This model can also estimate Hodgkin-Huxley

model up to 90% of accuracy (in terms of firing coincidence) by selecting the right kernel functions. This is another simple model, which can be used in simulations. By neglecting the dependency of ϵ and κ on $(t - \hat{t}_i)$ and losing some accuracy, even a simpler model, called SRM_0, can be obtained.

Quadratic Integrate-and-Fire Model (QIF)

A biologically more plausible model than LIF is a non-linear model called Quadratic Integrate-and-Fire (QIF), also known as the theta-neuron or the Ermentrout-Kopell canonical model [121]. In this model the derivative of the membrane potential depends on a quadratic function of the membrane potential. The dynamics of this model are described by equation [121]:

$$\tau_m \frac{du}{dt} = -a(u(t) - u_{rest})(u(t) - u_{thres}) + RI(t). \tag{2.9}$$

where u_{rest} and u_{thres} are resting and threshold potential of the neuron respectively. Unlike LIF model, this model has a dynamic threshold and resting potential (u_{rest} and u_{thres} only when $I(t) = 0$), is capable of generating realistic spikes with latencies and has bistable states of tonic spiking and resting [169].

Izhikevich Model

Computationally simple models like LIF can be implemented efficiently in computer simulations but cannot display all the behaviours of complex and CPU-intensive models like Hodgkin-Huxley. Recently, Izhikevich proposed a simple model [168] with a reasonable computational complexity that can exhibit all the complex dynamics of the Hodgkin-Huxley model like bursting, chattering, adaptation, and resonance. The model consists of a 2D system of ordinary differential equations of the form:

$$\begin{aligned} \frac{du}{dt} &= mu^2 + nu + p - v + I(t) \\ \frac{dv}{dt} &= a(bu - v) \end{aligned} \tag{2.10}$$

where $I(t)$ is the sum of the all post-synaptic currents and the external input current $I_i^{ext}(t)$. In a pulsed-coupled model used in example of [168], the total input current of neuron i can be written as:

$$I_i(t) = \sum_{j \in F} w_{ij} + I_i^{ext}(t) \tag{2.11}$$

where F is the set of pre-synaptic neurons that fire at time t. The parameters m, n and p can be obtained by fitting the model with the behaviour of a cortical neuron so that we have membrane potential $u(t)$ in mV and time t in ms scale, which results in:

$$\begin{aligned} u' &= 0.04u^2 + 5u + 140 - v + I(t) \\ v' &= a(bu - v) \end{aligned} \tag{2.12}$$

with after spike resetting condition:

$$\text{if } u \geq 30\text{mV, then } \begin{cases} u \leftarrow c \\ v \leftarrow v + d \end{cases}. \tag{2.13}$$

The parameters a, b, c, and d can be set to values recommended in [168] to obtain bio-plausible models of different types of biological neurons.

This simple model can reproduce the rich behaviour of biological neurons, such as spiking, bursting, post-inhibitory spikes and bursts, continuous spiking with frequency adaptation, spike threshold variability, bi-stability of resting and spiking states, and sub-threshold oscillations and resonance. Izhikevich claims that his model is canonical and equivalent to the Hodgkin-Huxley model meaning that it deviate from those bio-plausible models only by coordinate change [168]. However, it consists of two equations with only one nonlinear term and therefore it is computationally inexpensive compared to the bio-plausible and accurate Hodgkin-Huxley models. The only disadvantage comparing to Hodgkin-Huxley model is that the parameters in Izhikevich model are not physically as meaningful as parameters of the Hodgkin-Huxley model.

Table 2.1: Biological features of different spiking neuron models and number of floating point operations needed for simulation of one millisecond of neuron activity from [169]. Empty squares indicate that it must be theoretically possible to produce the behaviour with that model, although Izhikevich did not find a parameter setting to produce it. + and - signs show that the behaviour is reproducible or not reproducible respectively by that model.

Models	biophysically meaningful	tonic spiking	phasic spiking	tonic bursting	phasic bursting	mixed mode	spike frequency adaptation	class 1 excitable	class 2 excitable	spike latency	subthreshold oscillations	resonator	integrator	rebound spike	rebound burst	threshold variability	bistability	DAP	accommodation	inhibition-indiced spiking	inhibition-induced bursting	chaos	# of FLOPS
integrate-and-fire	-	+	-	-	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	5
integrate-and-fire with adapt.	-	+	-	-	-	-	+	+	-	-	-	-	+	-	-	-	-	+	-	-	-	-	10
integrate-and-fire-or-burst	-	+	+		+	-	+	+	-	-	-	-	+	+	+	-	+	+	-	-	-		13
resonate-and-fire	-	+	+	-	-	-	-	+	+	-	+	+	+	+	-	-	+	+	+	-	-	+	10
quadratic integrate-and-fire	-	+	-	-	-	-	-	+	-	+	-	-	+	-	-	+	+	-	-	-	-	-	7
Izhikevich (2003)	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	13
FitzHugh-Nagumo	-	+	+	-		-	-	+	-	+	+	+	-	+	-	+	+	-	+	+	-	-	72
Hindmarsh-Rose	-	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	120
Morris-Lecar	+	+	+	-		-	-	+	+	+	+	+	+	+		+	+	-	+	+	-	-	600
Wilson	-	+	+	+			+	+	+	+	+	+	+	+	+		+	+					180
Hodgkin-Huxley	+	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+		+		1200

Bio-plausibility and feasibility of spiking neural models

Izhikevich carried out a model comparison [169] of main different spiking neuron models taking into account the computational cost and bio-plausibility of these models. He measured the computational cost of the models in number of floating point operations (FLOPs) needed for simulation of one millisecond of neuron activity. Bio-plausibility of the models were measured in number of features based on:

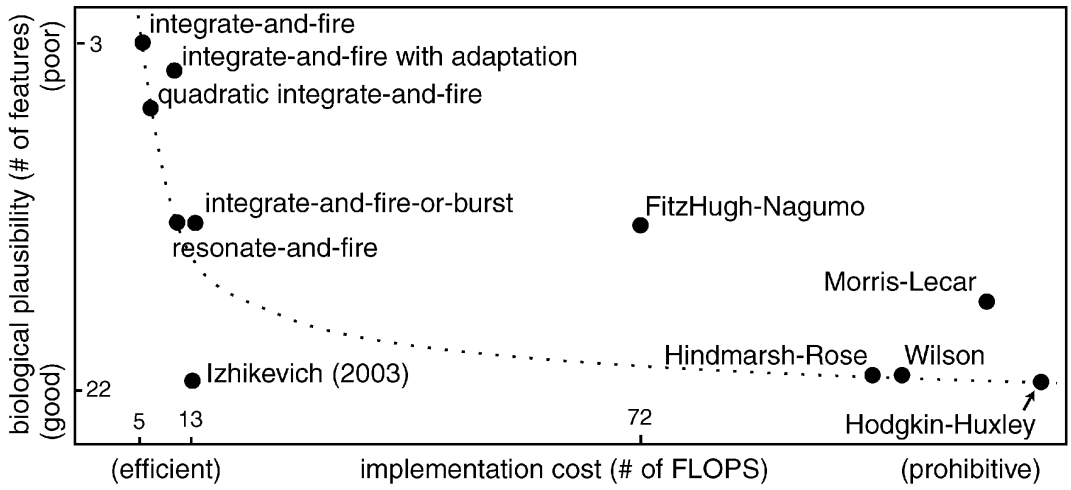


Figure 2.5: [169] Number of biological features of the spiking neuron models against number of floating point operations needed for simulation of one millisecond of neuron activity with each model [169] based on Table 2.1.

1. Biological features such as biophysical meaningfulness of the model and parameters,
2. Performance match of the model compared to the behaviour of biological neurons, and
3. Generality of the models, meaning the number of different behaviours of different types of biological neurons that can be represented by the model.

This comparison, summarised in Table 2.1[169] and Figure 2.5[169], shows that Izhikevich model is computationally the cheapest model with about the same bio-plausibility of the Hodgkin-Huxley model. By translating computational cost (time, T) to speed (frequency of operations, f), as one of the feasibility measures, using:

$$f = \frac{1}{T} \tag{2.14}$$

and plotting the data linearly, we arrive at Figure 2.6. Multi-compartment models of the neurons are also added to the chart with near zero feasibility and speculatively higher numbers of features. This is consistent with the general bio-plausibility-feasibility trade-off suggested in Chapter 1 (Figure 1.1).

Izhikevich's study does not include SRM model. This is probably because SRM is very general, meaning that with choosing different kernel functions it is possible to arrive at approximately equivalent of a broad range of neuron models from LIF to Hodgkin-Huxley. Speed of the SRM is also dependent on this kernel function selection. It can be seen that the bio-plausible selections of kernel function for SRM [121] can not produce a computationally cheaper model than Izhikevich model unless some approximations are involved.

2.4.4 Recurrent Neural Networks

A major part of the classical artificial neural networks are feed-forward with directed acyclic network graphs [73, 37]. In contrast, recurrent neural networks can contain directed cycles in their architecture leading to a much higher level of complexity and new features. Although many different problems have

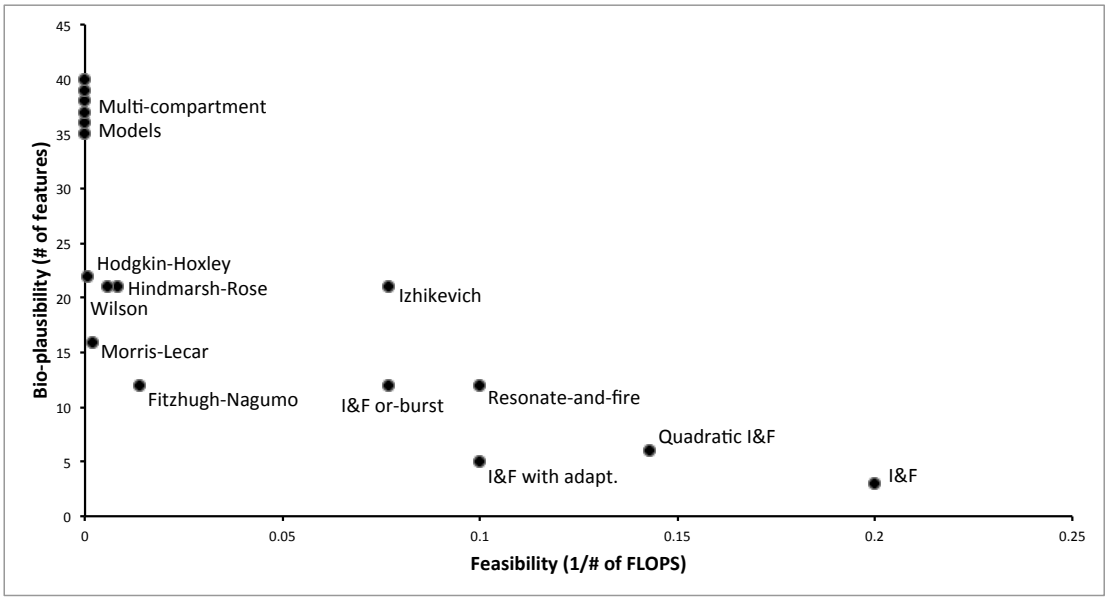


Figure 2.6: Bio-plausibility of the spiking neuron models (in number of features) against feasibility (speed)

been tackled successfully using feed-forward neural networks, the biological neural networks are not limited to those architectures.

Recurrent Neural Networks (RNNs) have shown great potential in solving different engineering problems such as classification, regression, prediction, control, and simulation of dynamic systems. They can naturally process temporal inputs while feed-forward networks need delay embedding and more parameters to be able to process temporal data [359]. RNNs are shown to be Turing equivalent [195] and universal approximators [110]. They can also approximate finite state automata [282]. However, the high computation cost and slow convergence and suboptimal solutions of RNN learning algorithms limited their wide real-word application [176].

Reservoir Computing (RC)

Recently, with independent works of Buonomano [53], Dominey (Temporal Recurrent Neural Networks) [84], Maass (Liquid State Machine - LSM [241]), Jaeger (Echo State Network - ESN [176]) and Steil (Back-Propagation Decorrelation - BPDC [353]) a new technique, collectively known as Reservoir Computing (RC) [332], emerged, which is claimed to be capable of processing analogue continuous-time inputs and to mitigate the shortcomings of the RNN learning algorithms.

The RC approach is generally based on a recurrent network of (usually) non-linear nodes. This recurrent network, which is called reservoir (also called liquid and dynamic filter) transforms the temporal dynamics of the recent input signals into a high-dimensional representation. This multi-dimensional trajectory can then be used as latent state variables by a simple linear regression/classification or a feed-forward layer (known as readout map or output layer) to extract the salient information from transient states of the dynamic filter and generate stable outputs. This is very similar to Kernel method [336, 70] used in Support Vectors Machines (SVM) [385, 70], which use a mapping function to map data points into a high-dimensional feature space and then linearly separate them into classes [240]. However, unlike

SVMs, RC has an intrinsic temporal nature.

Amazingly diverse choice of dynamic systems can be used as reservoirs: RNN [241], gene regulatory networks (GRN) [183], an animal brain [276], or even a bucket full of water [97]. The reservoir is traditionally a randomly generated RNN with fixed weights. Only the output layer is trained. Linear nature of the readout map dramatically decreases the computational cost and complexity of the training. Nevertheless, it has been shown that the topology, weights and the other parameters (*e.g.* bias, gain, threshold) of the reservoir elements can change the dynamics of the reservoir and thus affects the performance of the system [175, 232]. Therefore, a randomly generated reservoir is by definition not optimal.

Researchers tried to propose different measures and methods for generating and/or adapting reservoirs for a given problem or problem class [332]. However, there is none or very limited theoretical ground for specifying which reservoir is suited for a particular problem due to the non-linearity of the system [175]. Moreover, with only one positive result, in case of intrinsic plasticity [354], the development of unsupervised techniques for effective reservoir adaptation remains an open research question [332]. Another open question is the effect of the reservoir topology on the performance and dynamics of the system [175]. There is some evidence that hierarchical and structured topologies can significantly increase the performance [332]. Recently some deterministic formulations for generation of competitive reservoirs were suggested [307, 308].

Once the reservoir is optimally designed, tuned, or trained (in an unsupervised fashion), given a problem class, then different readout maps can be simply trained (in a supervised mode) for performing different tasks [241]. Computational power of the reservoir can be increased by just adding more neurons to the existing network [241]. The system is also very robust to noise [241].

RC is a very biologically plausible approach to RNNs [234, 233]. In Liquid State Machine (LSM) technique, generation recipe of the reservoir (liquid) follows biologically motivated topologies and metrics. The fact that the same reservoir can be used for inference of different outputs or that there are some deterministic architectures or some generic but not well-understood adaptations that can be used for many problems [307, 308, 234] show other promising similarities with the mammalian brain characteristics. RC also shows how temporal information can be represented spatially in the brain, providing a temporal context for perception of the current inputs [234]. As a matter of fact, LSMs have also been used as models of cortical microcircuits in cognitive and neuroscience studies [234] to explain processes in biological brains. However, there are still some other biologically not very accurate features in RC for example in terms of the accuracy of the neuron models used in RC (usually LIF neurons).

Hierarchical Temporal Memory (HTM)

Inspired by the structure and characteristics of the mammalian brain, Hawkins introduced a new model of the brain called Hierarchical Temporal Memory [150] in his book “On Intelligence”. Although he misleadingly claimed to solve “the global brain problem” [360] and made other questionable claims about artificial intelligence, the brain, and consciousness [292, 360, 96], he proposed a model that both explains many functions of the brain, and opens the way for creating new useful intelligent systems.

He builds upon the thesis [273] that many different functions of the neocortex such as vision, hearing, language, motor planning, etc. are based on a single common algorithm and structure. He proposed a general algorithm that explains how brain regions work together to store previous experiences and use them to predict and model future ones and their causes [150, 151]. Later, Hawkins and George presented a formal model of the theory [119] based on a belief propagation model. They used a hierarchical Bayesian network of sequence recognisers (based on Hidden Markov Model - HMM) in this model. They implemented and successfully tested it on a visual perception problem [119]. They also released a platform and a tool set [118] that allows other researchers and developers build upon their system. This attracted considerable attention and new products were released based on the platform. But the relation of the formal model to neural microcircuits in the brain was not clear and lower-level functioning of the regions (HMM) and the communication means between regions were far from those of the brain. Recently, a new model called HTM Cortical Learning Algorithms (CLA) was published [281] that fills these gaps by explaining the relation of the model to brain microcircuits already known to neuroscientists, and using a more bio-plausible neuron model with sparse coding. Very recently, they also introduced a cloud-based service [280] that allows researchers to evaluate the new model for online prediction and anomaly detection on temporal data streams.

The HTM model is bio-inspired and very biologically plausible compared to many other models with comparable capabilities [281]. Importance of the temporal persistence of the causes (objects) [151, 150], utilising sparse coding and RNNs for spatiotemporal pattern recognition, and other biological assumptions [281] are all examples of bio-plausibility of HTM. However, although a new neuron model is introduced in the CLA that is, in many aspects, much more bio-plausible than aforementioned spiking neuron models, it is in many other aspects heavily abstracted and simplified [281]. Current known CLA implementations are also still limited to a single layer [281]. There are no results from the CLA published yet.

2.4.5 Spiking Neural Networks Applications

Spiking neural networks are definitely still in their infancy and few real-world applications exist. This may be partly due to lack of analytical methods for designing such networks. A potential method is RC. In the academic context, RC has been used in temporal pattern classification and pattern generation, time series prediction, nonlinear systems control, timing, routing, and memorising. RC-based speech recognition systems shown good results, which compete with the state of the art systems [234]. Amazingly, in one case, the best performance was gained when a sound-to-spike coding front-end similar to the inner ear was used [389]. A multi-layer ESN-based system is being developed for handwriting recognition[234]. There are other stories of successful applications of RC-based systems in robotics, financial forecasting, epileptic seizure detection, Brain-Computer-Interfacing (BCI), Liquid State Machines were used in robust classification of other temporal or spatiotemporal patterns [332]. RC-based Spiking neural networks were also used in computational neuroscience and cognitive science [234].

Spiking neural networks are reported to be successfully used in clustering and classification of static data as well [45]. They showed a better performance compared to K-means and SOM (Self-Organizing

Map) in a number of real-life cases [45]. These networks are also useful in function approximation like applying them to iterative finding of roots [166]. Artificial spiking neural networks can be also used in neuroscience and studies on the brain. A comparison of different associative memory models including spiking neurons can be found in [199]. Spiking neuron models were used in visual sensory systems. Inference of depth from motion [416], image segmentation [69], robot vision [81, 109, 102] and many others [52] are examples of recent applications in this area.

2.4.6 Spiking Neural Network Simulators

Neural networks are often implemented and simulated using software running on a digital computer. However, implementing neural networks directly in hardware allows them to be massively parallel, fast, and asynchronous. As Moore's law has stayed valid for the last few decades, it is expected to realise very large scale neural networks directly in silicon in near future. Many many different design and implementations of hardware-based neural networks exist [268]. A large part of these are classified as classical, feed-forward, non-spiking, or other types of artificial neural networks (e.g. MLP, RBF, SOFM, etc.) [268]. Here the focus is on spiking neural networks. Many digital, analogue, and hybrid VLSI designs and models for spiking neural networks have been proposed by different researchers [239, 268]. Analogue systems are more power-efficient compared to digital systems but they are not flexible for research [113] and their design is difficult due to effect of noise and need for reliable non-volatile memory [396, 268]. Moreover, spiking neural networks lend themselves to digital communication [44, 113]. Therefore, digital and mixed-signal spiking neural network designs have been more successful. In the following sections, we focus on the digital designs and particularly those that can be implemented in FPGAs.

Depending on the goals and objectives of the system, different implementations tackle the problem differently. Usually when neural networks are implemented in special hardware it is to gain speed or scalability, either for neuroscience studies or engineering applications.

Many designs are aimed at large-scale simulations of the mammalian brain, or part of it, with reasonable speed, power consumption, flexibility and cost [114]. To this end, designers have turned to using software based systems based on very powerful super computers that are either custom-designed for neural simulation or are general-purpose supercomputers that are suitable for this application. Markram's Blue Brain project on IBM's Blue Gene is one important example of such endeavours to simulate very bio-accurate large-scale models of the mammalian brain [247]. Sporting 8192 CPUs, IBM's Blue Gene machine can provide up to 360 TFLOPS. This computation power is used to simulate accurate compartmental models of neurons. Blue Brain is currently aimed at scales of 100,000 very complex neurons or 100 million simpler neurons [247].

Izhikevich's simulation of a network of 10^{11} neurons with 10^{15} synapses (comparable with human brain) on a Beowulf cluster of 27 processors running at 3GHz is another example [170]. As simulation of one second of brain activity with 1ms resolution took 50 days on that cluster-computer, the system didn't have any application in neuroscience or engineering but provided valuable insights into the challenges of simulating such large-scale models. At that scale, it is not even possible to store all the synaptic weights

in the storage and this simulation was only possible by regenerating the weights and connections in every time step. He later used a similar cluster of 60 processors to simulate 1 million multi-compartmental neurons with half a billion synapses. It takes one minute to simulate 1 second of network activity with sub-millisecond resolution on that cluster [173].

Furber's SpiNNaker project [114] at the University of Manchester is another instance of efforts to design large-scale (billion neurons) [287] software-based spiking neural network simulators. SpiNNaker is based on a Torus 2D mesh packet-switch network of multi-processor network-on-chip nodes each containing up to 20 ARM cores. Using embedded processors is based on the energy efficiency (very low power consumption per MIPS of computation power) of these chips compared to high-end processors such as those used in IBM's Blue Gene. A special-purpose network-on-chip and router is added to each chip that are optimised for conveying spike-event packets. Recent evaluations [287] confirmed that SpiNNaker is scalable to one million cores with capacity to simulate one billion neurons with trillion synapses in real-time.

A series of projects at the Technical University of Berlin leading to SPINN Emulation Engine [152] are also good representatives of many efforts to design special hardware for large-scale simulation of spiking neural networks. This series of project are mostly based on designing special chips for neural simulation that can be added to standard PCs on accelerator boards. The next to last project in this series is SP^2INN [257] aiming at simulating one million LIF neurons with several million synapses on a custom-designed VLSI chip. Mehrtash *et. al.* [257] stated that although their design gained a speedup of 35 compared to a software-based simulator on previous generation of best workstations, the market force behind the development of general-purpose computers makes it very difficult to compete with software-based simulators. The performance and capacity of the general-purpose computers are growing more or less according to the Moore's Law and special-purpose VLSI chips cannot compete with their flexibility and performance. The authors conclude that FPGAs are the solution for a reasonably flexible system-on-chip (SOC) spiking neural network simulator [257]. Therefore, SPINN Emulation Engine project [152] focused on using FPGAs and gained up to 30 times speedup compared to software-based simulation on a state of the art PC. Their design, based on three Virtex II FPGA chips on the same board is capable of simulating up to half a million non-leaky integrate and fire (IFN) neurons and 800 million adaptive synapses. Hellmich also explained the memory requirements of such simulation systems and how academic and commercial FPGA platforms (boards) do not provide such capacity and bandwidth thus special design of the system with three FPGAs.

Researchers used Graphical Processing Units (GPUs) for simulating spiking neurons as well. Bhuiyan *et. al.* [33] evaluated simulating a 2-level network of up to 5.8 million Izhikevich and Hodgkin-Huxley point neurons for image processing. They reported a maximum speedup of 9.5x over the sequential implementation on the state of the art general-purpose processors for Izhikevich while parallel implementations on Xeon multicore processors reached to 58x for 1 million Izhikevich neurons and dropped to 17x for 5.8 million neurons [33]. However, GPU implementation outperformed all other parallel implementations for Hodgkin-Huxley model with a consistent speedup of 119x compared to the

sequential implementation while Xeon gained only a 80x speedup. They concluded that the ratio of processing to communication (FLOPS/Byte) of the neuron model is a good indicator for which architecture to choose [34, 33]. For neuron models with low FLOPS per Bytes (such as Izhikevich model) multicore processor architectures similar to Xeon perform better than GPUs and for neuron models with high Flops per Bytes (*e.g.* Hodgkin-Huxley) GPUs outperform other multicore architectures [34, 33]. In another paper [32] they implement three different spiking neuron models in a hardware-software solution based on a Xeon host and an accelerator with an Altera Stratix™II EP2S180 FPGA. They simulated a 2-level network of up to 2 million Izhikevich neurons showing the same trend regarding the FLOPS/Bytes ratio in GPUs [32]. Han *et. al.* [143] Investigated the performance of a cluster of GPU's in simulating similar 2-level network of up to 9 million izhikevich neurons with similar results. They also noted the biological implausibility of the network architecture used in their experiments. Fidjeland *et. al.*[99] Investigated the simulation of more bio-plausible networks (fully-clustered and uniformly connected) of up to 40,000 izhikevich neurons with 1000 synapse each on GPUs.

There are also other rather ambitious projects such as FACETS [100] and SyNAPSE [75] based on mixed-signal custom chips and utilising memristors for much larger or hyper-realtime simulations at expense of much higher costs. Accelerators based on Digital Signal Processors have also been used for spiking neural network simulation [242, 268]. Many many other designs similar to above examples exist that can be found in surveys and reviews in [91, 218, 268]. In general, implementation of spiking neural networks in digital hardware can be classified from different aspects:

- Parallelism: a spectrum of methods ranging from time sharing of one PE (processing element) between all neurons and synapses, to dedicating one PE for each neuron or even for each synapse
- communication method: direct mapping (from the neural network into hardware), network-based, memory-based, circuit switching, packet switching, etc.
- Storage: centralised, distributed or massively distributed
- Simulation method: event-driven or time-step
- Computation method: stochastic or deterministic
- Arithmetic method: parallel or serial
- Representation: floating-point, fixed-point or integer
- Programmability: Software-based, parametric, reconfigurable, or hard-wired
- Architecture: general-purpose computers, supercomputers, custom architectures based on embedded processors, GPU-based accelerators, custom chip (ASIC) accelerators, DSP-based accelerators, FPGA-based accelerators

Based on these choices, the resulting design may feature different levels of scalability, speed, flexibility, fault-tolerance, capital cost or energy efficiency (running cost). A key factor affecting many

other choices is the architecture. Using general-purpose processors provides the highest flexibility, programmability and ease with the lowest cost. However, it is not a scalable, fast and fault-tolerant solution. Supercomputers can provide capacity for relatively large-scale and rather flexible simulations at the cost of capital and energy efficiency [113]. It is possible to increase speed and fault-tolerance of the supercomputer-based solutions. Custom architectures based on embedded processors (such as SpiNNaker) have the same capital cost, speed and fault-tolerance of supercomputers with higher energy efficiency and a better scalability [113]. Systems based on GPUs can not scale well when network size and connectivity is increased due to their restricted communication and memory structures optimised for graphical or general purpose computing [301, 287]. GPU-based solutions are not fault-tolerant and not energy efficient to be scaled up in clusters [301]. However, they provide good flexibility in terms of the neuron model [301, 99] and reasonable speedups over general-purpose PCs [99, 34, 33].

Custom chips, depending on the technology used and internal architecture and their interfacing can provide high-speed, large-scale and energy efficiency, but they lack flexibility and fault-tolerance. But the main issue with bespoke chips is very high capital cost for design and fabrication of these chips. Even per unit cost of commercial neural chips with specific neuron models are high as they are not very popular as CPUs, GPUs and FPGAs.

DSPs usually provide more cost effective solutions than custom chips and may provide a higher level of flexibility in terms of neuron models. However, in almost every other aspect they are inferior to custom chips. Using FPGAs for simulating spiking neural networks shows a broad range of different results and trends. This is essentially due to their flexible architecture that can be used in each design in completely different way. Generally, FPGAs are considered to be more energy efficient than GPUs but not compared to custom architectures based on embedded processors [301]. They are not as fast as custom chips of the same generation and need much more silicon area to implement the same logic as custom chips. They are also stated to have routing limitations due to their inherent circuit-switched fabric [301]. However, creative designs by different researchers and many examples show that there might be a niche application for them as a platform for bio-plausible spiking neural networks. The following section is dedicated to a review of different examples and approaches to simulating spiking neural networks on FPGAs.

2.4.7 FPGA based Spiking Neural Networks

According to Johnston *et. al.* [182], spiking neural networks seem to be the most efficient class of neural networks in terms of hardware resources on FPGAs compared to RBF (Radial Basis Functions) and MLP (Multi-Layer Perceptron) neural networks. A plethora of different designs and implementation exist for realising different types of spiking neural networks on FPGAs. Table 2.2 gives an overview of important examples. Almost all these designs are based on time-step simulation technique. Event-base simulation is possible only with neuron models such as SRM. However, the computational complexity of the bio-plausible kernel functions needed for such neuron models is prohibitive [182]. A simpler form of the model known as SRM_0 relieves the model with the assumption of the kernels being independent of the time. Nevertheless, kernel functions have significant effect on speed and silicon area [182]. One

possible solution is to use lookup tables [140]. However, the size of the lookup table grows faster than super-exponentially with the accuracy, and separate lookup tables are needed for each kernel and for each processing element (PE) [140]. Some designs, however, use lookup tables or exponential kernel functions for synapse models [315, 314]. Integration and neuron state update time-steps (time resolution) for Izhikevich and LIF models range from 0.0625 to 1 millisecond. Shorter time steps are used when there is a bottleneck in the communication infrastructure and designers try to mitigate the network congestion by splitting each millisecond of neuron activity to many time steps and spreading the neuron activity over time slots. Otherwise a time-step of 0.5 to 1.0 millisecond is assumed to offer enough accuracy. For more detailed models such as Hodgkin-Huxley much shorter time steps are needed, which contribute to the inefficiency of these models.

Leaky Integrate and Fire (LIF) is the most common neuron model used in these designs. This can be explained with the relatively low computational complexity of this model [169]. Some designs [312, 374] use simplified linearised versions of the LIF model to save silicon and time. A few designs, aimed at higher bio-plausibility and neuroscience applications use Hodgkin-Huxley or other detailed models [314, 401, 130, 315]. A very popular bio-plausible but computationally cheap model that is introduced in the FPGA implementations recently is Izhikevich model [303, 269, 363]. Although Izhikevich model can offer a very higher degree of plausibility with a little more computation, it needs more parameters and state variables to be stored in (or fed into) PEs (Processing Elements), which affects the hardware complexity, speed, and scalability of the system. A few designs incorporate learning into the hardware. This is usually an unsupervised Hebbian learning or similar (STDP - Spike Time Dependent Potentiation) technique that can be performed locally at the synapse or with the minimum global data or feed back. As the learning process is occurring on a longer time-scale and mostly when a neuron fires, it is common to use less hardware resources or even software solutions for it.

[p]

Table 2.2: Examples of different design and implementations of spiking neural networks on FPGAs.

Ref.	Model	Learning	Connectivity	Parallelism	Scale	Scalability	FPGA	Objectives	Applied to	Speed	Step	Approach	Storage	Comm.	Arith.	Area
[21]	N/A	No	Fixed, Layered, feedforward	1 PE for each neuron	4 and 40	N/A	Xilinx Spartan II	bio-inspired robot navigation	Robot Navigation	1 (real-time)	1ms	Manual translation from C to VHDL	local distributed	directly mapped network	Parallel, fixed-point deterministic	N/A
[124]	LIF	STDP	Fixed, Layered, feedforward	Time-multiplexed Neurons, Synapses, and STDP	300 / 11200 and 4200 / 1.9 million	by adding more FPGAs May run into inter-chip communication bottlenecks	2 Xilinx Virtex II	Large-scale, bio-plausible neuron models	1D coordinate transformation	1/27 and 1 / 4237	0.125 ms	HW / SW co-design	External dedicated RAMs	shared memory	Parallel, fixed-point deterministic	N/A
[123]	LIF	No	fixed	1 PE for each neuron or each synapse	168 / 168 and 13 / 1300	No	Xilinx Virtex II	Speed, bio-plausibility	N/A	12500x real-time	0.125 ms	direct mapping of the network and use of Xilinx System Generator	local	directly mapped network	Parallel, fixed-point deterministic	63 slices per neuron and 33 slices per synapse

Table 2.2: Examples of different design and implementations of spiking neural networks on FPGAs.

Ref.	Model	Learning	Connectivity	Parallelism	Scale	Scalability	FPGA	Objectives	Applied to	Speed	Step	Approach	Storage	Comm.	Arith.	Area
[152]	non-leaky IF	Similar to Hebbian	4 or 8 nearest neighbour	3 PEs for all neurons	100,000 / 800,000 adaptive and half million / 800 million adaptive	N/A	2 Xilinx Virtex II + 1 Xilinx Virtex II Pro	large-scale, flexibility	Image processing	sub-realtime estimated 30x faster than SW	1ms	HW/SW co-design, better memory bandwidths	external dedicated RAMs	shared memory	parallel, fixed-point deterministic	N/A
[289]	Noisy LIF	No	Fixed, sparse	10 PEs each for 112 neurons and 912 synapses	1120 / 9120 and 1120 / 9120	No	Xilinx Virtex II	Large-scale, realtime	N/A	1 (real-time)	0.5ms	SIMD processor in FPGA	Copies of network activity for all PEs, Weights distributed over PEs	Shared buses between PEs	Parallel, fixed-point deterministic	N/A
[315]	Cond. based with SRM synapses	No	biological	1 to n PEs for all neurons	1024 / 4096 and 4096 / 16768	up to the size of the FPGA	Xilinx Virtex II	real-time	N/A	1 (real-time)	0.1ms	Pipelined PEs	dedicated memory for activity, states and parameters	shared memory	parallel (pipelined), deterministic	N/A

Table 2.2: Examples of different design and implementations of spiking neural networks on FPGAs.

Ref.	Model	Learning	Connectivity	Parallelism	Scale	Scalability	FPGA	Objectives	Applied to	Speed	Step	Approach	Storage	Comm.	Arith.	Area
[374]	simplified LIF	Hardware Hebbian	fully connected	1 PE for each neuron and its input synapses	30 / 900 and 30 /900	up to the size of the FPGA but with limited number of synapses per neuron (by time and area) and probably routing resources	Xilinx Spartan II	real-time, flexibility, evolving topology, low area	Frequency discriminator	about 1000	1ms (assumed)	local storage and processing for neuron and synapses with direct network mapping	distributed local memory for weights and states	directly mapped network	Parallel, fixed-point deterministic	53 slices per neuron with 30 synapses
[303]	Izhikevich	No	2-layer feed-forward	a set of PEs each for a subset of neurons, 1 PE for all synapses	624 and 9264	N/A	Virtex II Pro and Virtex-4	Large-scale bio-plausible neuron model	character recognition	Sub-realtime / 8.5 times SW	1ms	Pipelined modular PEs	local distributed	central access to distributed local memory	fixed-point, deterministic, parallel	N/A
[269]	Izhikevich	Hebbian	biological (Hippocampus)	one PE for each type of neuron	36 and 36	N/A	Xilinx Virtex-II Pro XC2VP30	real-time	maze navigation	1 (real-time)	0.5ms	HW/SW co-design	central	shared memory	fixed-point, deterministic, parallel	N/A
[61]	LIF	STDP	N/A	1 PE for each neuron and its input synapses	32 / 4096 and 64 / 8192	up to the size of the FPGA but limited by the AER bus bottleneck	Xilinx Spartan 3	flexibility, speed	Speech recognition	3125	0.0625ms	single address event bus/single weight memory, PE array	central global weight memory	Address-Event Representation (AER) common bus	fixed-point, deterministic, parallel	(estimated less than 180 slices per neuron

Table 2.2: Examples of different design and implementations of spiking neural networks on FPGAs.

Ref.	Model	Learning	Connectivity	Parallelism	Scale	Scalability	FPGA	Objectives	Applied to	Speed	Step	Approach	Storage	Comm.	Arith.	Area
[290]	LIF	No	programmable, restricted biological	10 PE each for 1/10 of the neurons and synapses	640 / 10240 and 640 / 10240	No	Xilinx Virtex II	real-time	N/A	1 (real-time)	0.5ms	SIMD processor in FPGA	Copies of network activity for all PEs, Weights distributed over PEs	Shared buses between PEs	fixed-point, deterministic, parallel	N/A
[130]	Hodgkin-Huxley	No	N/A	1 PE for each neuron or each compartment	1	No	Xilinx Virtex II	Multi-compartmental, bio-plausibility	N/A	40x realtime	0.001ms	Direct mapping	local	N/A	fixed-point, deterministic, parallel	very high
[401]	Hodgkin-Huxley	No	flexible, fully connected	1 PE for all neurons and synapses	40 / 40	up to the size of the FPGA	Xilinx Virtex-4	flexibility, speed	N/A	8.7x realtime	0.01ms	Pipelined, automatic generation of Xilinx System Generator modules	shared memory	shared memory	fixed-point, deterministic, parallel	(estimated 346 slices and 5 multiplier blocks per neuron and its synapses)

Table 2.2: Examples of different design and implementations of spiking neural networks on FPGAs.

Ref.	Model	Learning	Connectivity	Parallelism	Scale	Scalability	FPGA	Objectives	Applied to	Speed	Step	Approach	Storage	Comm.	Arith.	Area
[122]	LIF	No	flexible, layered feed-forward	4 PEs each for 1/4 of the neurons and synapses	1 million / 52 million and 1 million / 52 million	up to the size of the FPGA	Xilinx Virtex-4	Large-scale, flexibility	edge-detection	Sub-realtime / 14 times SW	N/A	local memory with shared spike network router	local memory for each PE	shared network router	fixed-point, deterministic, parallel	N/A
[134]	LIF	Hebbian (in software)	fixed, biological	1 PE for each neuron or each synapse	25 and 25	N/A	Xilinx Virtex II	speed, bio-plausibility	Odour classification	hyper-real time (N/A)	N/A	direct mapping of the network and use of modular design with cascaded synapse modules	local state and weight memories + global shared weight memory	directly mapped network	fixed-point, deterministic, parallel	N/A
[363]	Izhikevich	in software	fully connected	1024 synapse PEs each for all the synapses of the same presynaptic neuron and 1 neuron PE for all the neurons	1024 and 1024	N/A	Xilinx Virtex-5	bio-plausible neuron model, speed	N/A	118x real-time	1ms	pipelined weight integration systolic tree and pipelined neuron state update module	local weight memory for each synapse PE, global time-multiplexed state and parameter memories	time-multiplexed single bit bus	fixed and floating point, deterministic, parallel	N/A

Table 2.2: Examples of different design and implementations of spiking neural networks on FPGAs.

Ref.	Model	Learning	Connectivity	Parallelism	Scale	Scalability	FPGA	Objectives	Applied to	Speed	Step	Approach	Storage	Comm.	Arith.	Area
[312]	simplified LIF	no	reconfigurable (limited neighbourhood)	1 PE for each neuron and its input synapses	64 and 64	up to the size of the FPGA	Altera Apex 20KE	real-time, flexibility, evolving topology, low area	Obstacle avoidance	1200x real-time (assuming time-resolution of 1ms)	N/A	Cellular direct mapping of the network with time-multiplexed synapse modules	local state memories for each neuron	directly mapped network	fixed-point, deterministic, parallel	N/A
[314]	Cond. based with SRM synapses	in software	biological	4 PEs each for 1/4 of the neurons and synapses	1024 and 1024	No	Xilinx Virtex II	real-time	N/A	1 (real-time)	0.1ms	HW/SW co-design, pipelined, segmented memory	separate memories for states, parameters and weights	shared memory	fixed-point, deterministic, parallel	N/A
[331]	LIF with synapse model	No	fixed, sparse	1 PE for each neuron or each synapse	56 / 560	estimated 1400 neurons on larger FPGA	Xilinx Spartan 3	speed, small-scale, area efficiency	Speech recognition	2930x real-time	1ms (assumed)	pipelined synaptic integration tree	local memory for each PE	directly mapped network	fixed-point, deterministic, serial	N/A

A number of models are limited to fixed multi-layer feed-forward and biologically implausible network connectivity architectures. Others assume a fully-connected topology, using up resources for all the possible connections while only a fraction of those will be used after training the network. Few others have fixed but biologically plausible topologies inspired by the brain structures. Networks are of different scales ranging from few neurons and synapses to a million neurons with 52 million synapses. All the large-scale designs are of sub-realtime speeds and hyper-realtime designs are usually small-scale. Thomas *et. al.* [363] introduced a new architecture that allows simulating a fully-connected network of1024 Izhikevich neurons with 118 times realtime speed utilising a pipelined neuron update module and a systolic tree with local synaptic memories and dedicated memories for neuron states and parameters. While this is an impressive achievement, the speed of the design directly depends on the number of neurons, which is in turn limited by the FPGA hardware resources. This design particularly relies on limited resource of shift registers and FPGA memory blocks for synaptic weights, neuron states and parameters. Assuming a fully connected topology requires large number of weight memories and integration units that might be not used in practice. They already used the largest chip in the Virtex-5 family for this design [363]. Assuming there are enough hardware resources available on the FPGA a realtime speed would be expected for about 64,000 neurons.

As Furber noted [113], each neural system needs to balance its resource usage for three functions of processing, communication, and storage. Processing is what happens mainly in neurons and synapses. Storage mainly involves synaptic weights, neuron parameters and states such as membrane potential and ion densities. Communication is needed for neurons to send spikes to each other. Different approaches ranging from SIMD (Single-Instruction Multiple-Data) architectures to heavily pipelined and systolic tree architectures are used for parallelisation of the processing. It is clear that using higher number of PEs (Processing Elements) offers a higher performance but at the same time aggravates the problem of communication between PEs. A shared memory or bus or other network architectures are common solutions [330]. When one PE is used for each neurons, it is also common to use directly mapped communication, which connects each neuron output directly to the inputs of the other neurons using a dedicated signal (axon). Both techniques have scalability problems that affect the speed or connectivity. Reconfigurable routing resources are also needed for flexible directly mapped connections. For storage, using dedicated memories for synaptic weights, parameters and neuron states mitigate the bandwidth problem. Using local memories for each PE using distributed block RAMs available in most of the FPGAs is a successful approach [363]. Nevertheless, the number and size of these block RAMs are fixed, limited and not necessary scaling up in proportion with the design requirements. Most of the designs use Xilinx Virtex II as one of the popular reconfigurable platforms and Xilinx Spartan 3 as a cheap solution. Since 2008 researchers started using newer families of FPGAs such as Virtex-4 and Virtex-5 [122, 363].

Another important aspect of the designs is how calculations are performed in synapses and neurons. This can be serial or parallel, and stochastic or deterministic. There are also deterministic bit-stream computation methods that diverge from the traditional binary arithmetic. In the following, some of these

computation methods and their usage in spiking neural networks in FPGAs are briefly reviewed.

2.4.8 Computations using stochastic bit-streams

The seminal work of Gaines [116] opened the path for design and implementation of different stochastic computing systems on digital hardware. In stochastic computing, a continuous signal or variable is represented with a random bitstream [50, 4]. The probability of seeing 1s in the bitstream determines the current value of the signal or variable. Although it takes some time to sample enough bits to estimate the probability for a bitstream, complex computations can be carries out using a combination of very simple operations [50, 4]. For example an AND gate can be used to multiply two values in the range of $[0, 1]$. The probability of receiving 1s in the output of the AND gate is equal to the product of the probability of 1s in two inputs of the gate. Figure **??** shows this in a very simple example with random bitstreams of length eight.

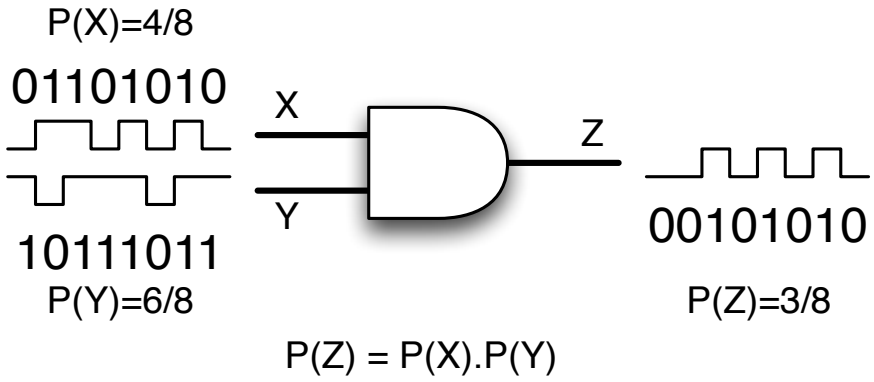


Figure 2.7: A simple example of computing product of two stochastic signals represented by 8-bit long random bitstreams using a single AND gate.

Unipolar and bipolar are two popular coding formats [50, 4]. In unipolar format, a signal value of $x \in [0, 1]$ is represented with the random bitstream X of probability $P(X = 1) = x$ while in bipolar format a signal value $x \in [-1, 1]$ is represented with the bitstream X where $P(X = 1) = \frac{x+1}{2}$. In [50, 4], Brown and Card focused on utilising these two formats for neural computations in digital circuits and showed some of the benefits of this computation technique. Numerous other stochastic formats can also be invented along with their respective computational circuits, each one with their own advantages and disadvantages. But in general, stochastic computing has many advantages over deterministic methods [50, 116]:

1. Simple hardware
2. Fault tolerance and robustness to noise
3. One-wire communication channel for each signal
4. High-clock frequencies due to simple hardware
5. Possibility of creating a trade-off between accuracy and computation speed with minimum hardware changes.

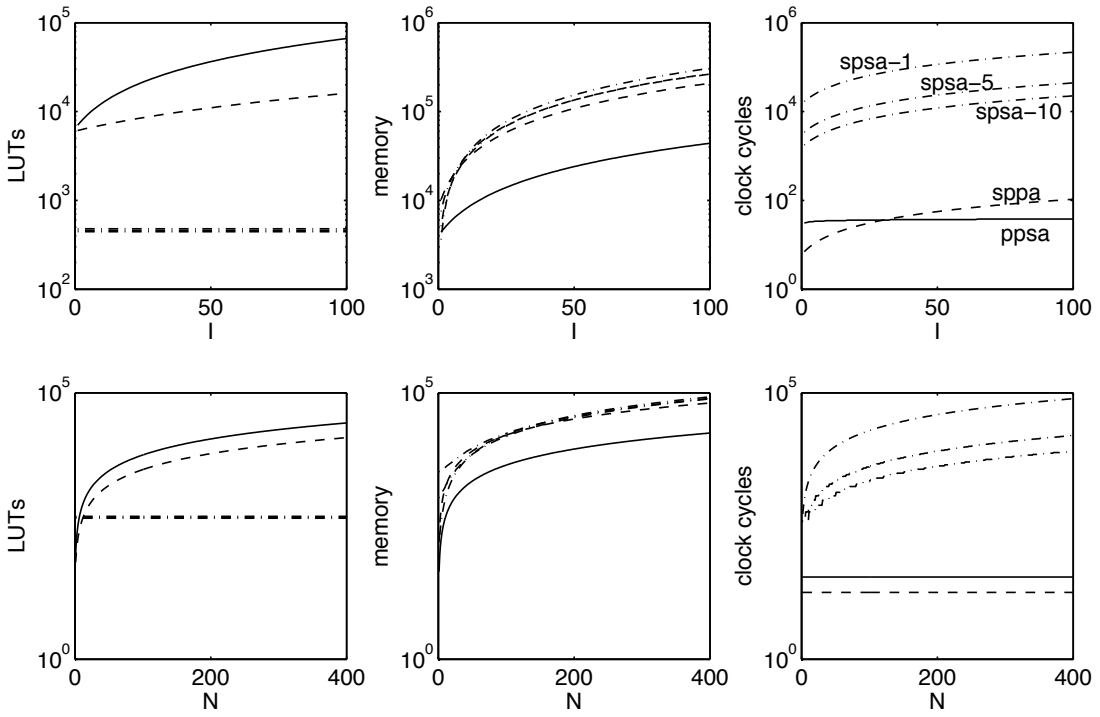


Figure 2.8: From [330], comparison of hardware resources and speed of different architectures (the SPSA architecture with 1, 5 and 10 PEs) with different number of inputs (I) and number of neurons (N).

There are also some disadvantages: the inherent variance in estimation of the signal value and need for longer integration periods for more accurate results. However, these can be usually mitigated to some extent by pipelining and using higher clock rates due to the very simple hardware.

Stochastic computing has been used in many non-spiking neural network models (for example [6, 178, 201]), and a few are designed specifically for FPGA implementation [14, 382, 249, 391, 421]. Pulse-mode neurons with stochastic arithmetic seem to be a very efficient choice (in terms of area) for implementation on FPGAs [14, 249, 50, 252, 82, 382, 383]. Similar stochastic computation techniques can be employed for spiking neural networks. This method can provide area efficiency with noise immunity and some bio-plausibility but with lower accuracy or speed compared to deterministic models due to the intrinsic trade-off between speed and accuracy of stochastic systems [14, 201, 50]. The accuracy of the stochastic signals is proportional to the square root of the length of the bit stream [14, 50, 201], which both affects the processing time and size of the counters and registers needed for storage and evaluation of the signal values. There are also deterministic bitstream solutions [49] that mitigate the accuracy problem to some extent with accuracy proportional to the length of the bitstream. In contrast, normal deterministic binary arithmetic need more hardware resources for processing but much shorter representations for the same accuracy.

2.4.9 Binary Arithmetic

Binary arithmetic can be performed in parallel (all bits at the same time) or serial (one bit at a time). Serial arithmetic needs less hardware resources but offers lower speeds depending on the representation length. [329]. Schrauwen *et. al.* [329, 331, 330] has reviewed and evaluated serial and parallel

arithmetic, serial and parallel integration (processing), their combinations and different interconnection architectures for implementing spiking neural networks in FPGAs with their examples and concluded that serial processing serial arithmetic (SPSA) is the most compact but the slowest and parallel processing serial arithmetic (PPSA) is the fastest with hardware resources growing logarithmically. Figure 2.8 shows the results from their investigation of different architectures with different number of neurons and inputs. Based on that premise, Schrauwen *et. al* [329, 331, 330] introduced a high-speed spiking neuron model for FPGA implementation based on the LIF model with serial arithmetic and parallel processing of the synapses utilising pipelining in a binary dendrite tree with 56 neurons achieving an impressive speed of 2930 times faster than realtime simulation and estimated even higher speeds and scales with better FPGA chips. They successfully tested the system for speech recognition using reservoir computing (RC).

One of the most compact implementations is Upegui's [381, 374] who synthesised 14, 30, and 62-synapse spiking neurons respectively in 17, 23, and 46 slices of a Xilinx Spartan II FPGA to explore different network architectures using evolution. However, this design is based on a simplified LIF model and uses a central memory for each neuron and does not scale well for large scale NNs as the number of inputs to each neurons is fixed and limited. There are other similar or slightly different designs that are not mentioned here but a good complementary coverage of spiking neural networks FPGA implementations can be found in [242, 62, 167]. Although a large part of existing FPGA implementation are for non-spiking neural networks [422, 250, 225], some of the techniques can be useful for spiking neural networks as well. There are also designs based on networks of FPGAs [415]. More recent design and implementations that also include evolution of the spiking neural networks such as [62, 272, 6, 381, 374, 312] will be reviewed in detail with other evolutionary SNN models on FPGAs in section 2.5.6.

2.5 Evolutionary Computing

Evolutionary Computation (EC) is a sub-field of Artificial Intelligence (AI), which can be loosely defined as the set of biologically-inspired techniques that are population based, parallel, of a random nature, and involving iterative progress, development or growth. Evolutionary Algorithms and Swarm Intelligence are two important subsets of evolutionary computation. Evolutionary Algorithms are best known with one of its popular techniques, Genetic Algorithms (GA) [160]. Other main techniques are Genetic Programming (GP) [205], Evolutionary Programming (EP), and Evolutionary Strategies (ES) . Evolutionary algorithms are known to be good meta-heuristic optimisation and search techniques where there is little or no knowledge about the search space [12]. Since they do not make any assumption about the fitness landscape, they are sometimes claimed to have a good performance for all problems. However, no-free-lunch theorem states that there is no such algorithm that can consistently perform well on all problems [403]. Bio-plausibility of evolutionary algorithms and empirical evidences shows that they can perform very well on those problems that are also faced in nature. This does not mean that they are easy to design or always fast. Need for designing a scalable representation and a suitable fitness function, their slow convergence to useful solutions, and too many parameters to tune are only a few issues that make it hard to use them in complex problems.

Despite of these drawbacks, evolutionary algorithms were effectively and widely used for search and optimisation in science, engineering and business [291, 107, 36, 68]. Using evolution to design physical structures has created a new paradigm called Evolvable Hardware.

2.5.1 Evolvable Hardware

Evolvable Hardware (EH or EHW) lies at the intersection of biology, computer science and engineering [127]. Generally speaking, Evolvable Hardware is automatic design and optimisation of any physical structure using evolutionary algorithms. These physical structures can be anything from optical lens sets to antennas, analogue filters or toddling robots. The term "evolvable hardware" was firstly used only for evolution of electronic circuits, but nowadays, people use it for any kind of "hardware" evolution. A recent review of evolvable hardware, in its broad sense, can be found in [231]. Haddow and Tyrrell also have a very comprehensive recent review of the field [137] noting the definition and boundaries of EH, the potentials and current challenges.

Some researchers name the final product "evolved hardware" when it is no longer evolving during its life time. On the other hand, a real evolvable hardware can be an embodied adaptable system that continues its evolution during its lifespan in its actual environment. This could be an instinct shift of a robot in response to the changes in its environment or a self-repairing satellite controller trying to evolve a circuit robust to different radiations and noises in space.

Evolvable Hardware can be classified in many ways according to the evolutionary algorithm, application area, adaptation method, evaluation process, and even the hardware platform used. Here we focus on its applications in design and adaptation of digital circuits and particularly spiking neural networks.

Evolving Digital Circuits

The most commonly used evolutionary algorithms for evolvable hardware are Genetic Algorithms (GA) [160, 127, 131]. However, Genetic Programming (GP) [206] and Cartesian Genetic Programming (CGP) [266] are also very popular. The whole process of evolving digital circuits by GA can be summarised in figure 2.9. Each circuit is represented by a binary string called chromosome (or a set of chromosomes in some cases), which comprises the genome of the potential solutions. The process starts with a population of random genomes (or sometimes an evolved seed population from previous runs). Each genome is mapped to a circuit through a direct or indirect mapping [127] process. In direct mapping, the chromosome explicitly describes the circuit at a switch, gate, or functional level. In contrast, indirect mapping uses the programs or derivation trees, which are encoded in the genome, to develop circuits [127]. The resulting circuits are evaluated according to the desired functionality using a fitness function. Circuits can be simulated in software or configured on a hardware platform for evaluation. The terms "extrinsic" and "intrinsic" are coined for these methods respectively [127]. The evaluation process involves feeding all the possible input vectors (or a subset of them when too many possibilities) and comparing the circuit output with the desired output to calculate a fitness score. After evaluation, a fitness score is assigned to each individual circuit. Fitter circuits are selected randomly as parents with a probability related to their fitness values. Less fit chromosomes are deleted from the population and new ones are reproduced instead using fitter chromosomes as parents. A set of recombination (crossover) and variation (mutation)

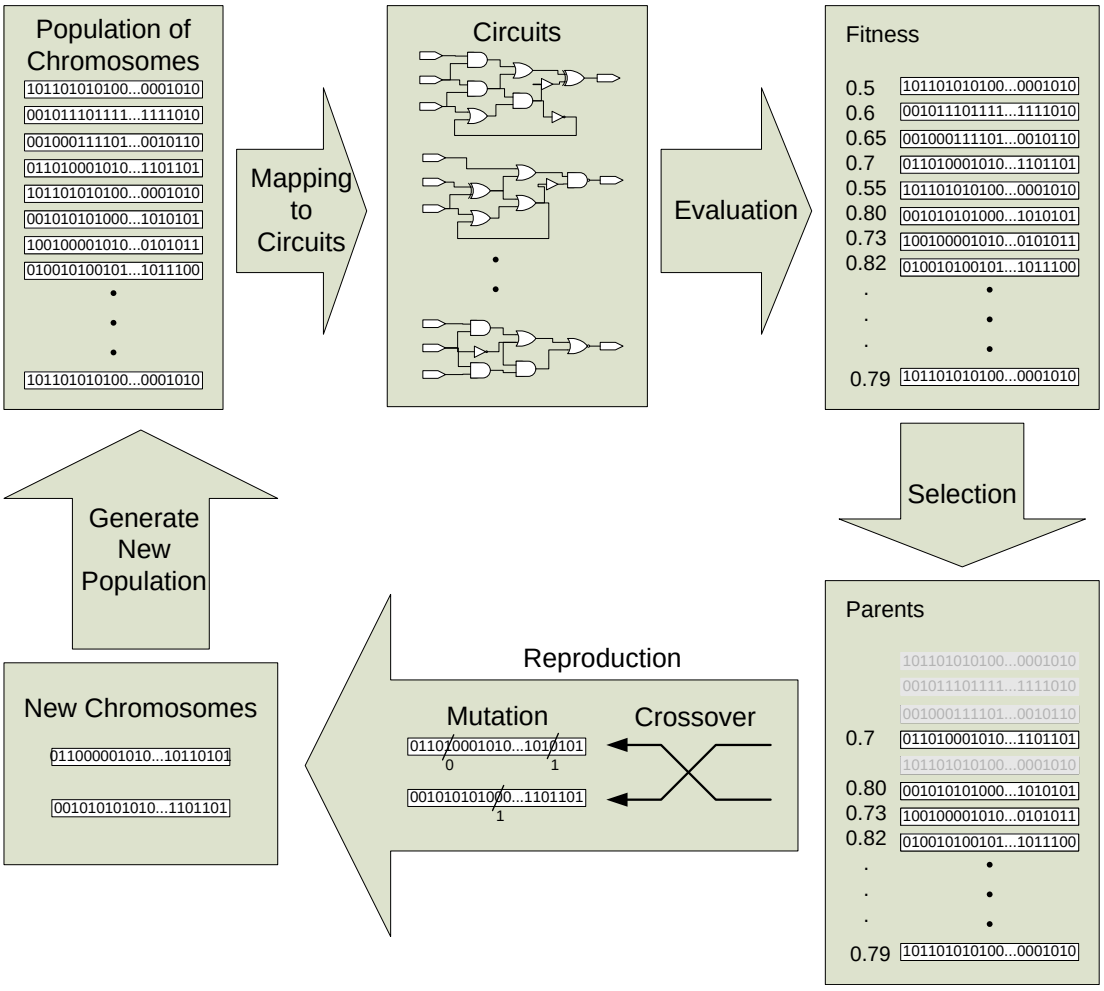


Figure 2.9: Summary of evolvable hardware process

operators (depending on the algorithm type and the genome representation) are used to reproduce a new genome from parent genomes. This process iterates and continues until a satisfactory circuit evolves or the stopping condition is met. This would effectively synthesise an evolved hardware solution. A similar process can be used for evolvable or adaptive hardware that is evaluated in the actual application environment and evolves through the product life-time. However, fail-safe mechanisms or other provisions are required to make sure that system will perform acceptably without damaging anything when an individual with a low fitness is evaluated.

Evolvable Hardware Platforms

Researchers used different reconfigurable devices for intrinsic evolvable hardware [156], including custom evolvable analogue and digital chips, Field Programmable Analogue Arrays (FPAA), Programmable Logic Arrays (PLA), Field Programmable Gate Arrays (FPGA), POEtic chip [271] and even Liquid Crystal Displays (LCD) [145]. Still, FPGAs are the most popular and *de facto* standard devices for intrinsic evolution of digital hardware [138]. A discussion of evolvable hardware platforms can be found in [138]. Also a recent review of different EH platforms can be found in [137]. Here we focus on FPGAs as the platform of this study.

FPGAs have been extensively used in evolvable hardware researches and applications. However, they have also some disadvantages. The constraints they impose on evolution for protecting the device from incorrect connections and damages, and unavailability of the configuration bitstream formats of some new commercial devices are two major problems in using FPGAs as evolvable hardware platforms.

To tackle these problems, three general approaches were previously applied to evolvable hardware on FPGAs. The first approach is to use an evolution-friendly FPGA (that cannot be damaged by contention due to incorrect configuration, *e.g.* XC6216) with an open configuration bitstream format allowing evolution to control the functional and routing resources of the FPGA at a low-level of abstraction [364]. This is an effective approach. However, those kind of devices are discontinued and new FPGA families are not evolution-friendly in that sense.

The second method is to design a virtual evolution-friendly FPGA on any FPGA [139]. This method is very flexible and allows designing the right type of reconfigurable cellular structure for each application but is very inefficient in terms of hardware resources [372]. Upegui reports that this method needs 4.5x more silicon area compared to the first method [372].

The third approach is to design the same cellular structure of the virtual FPGA and pre-route the connections between cells, but instead of dedicating some hardware resources to configuring the virtual FPGA, lock location and pin configurations of the LUTs, Muxs, and flip-flops in the cellular structure to pre-specified sites on the FPGA and then use partial reconfiguration for switching the routing and changing the function of each cell [377]. Undocumented configuration bitstream formats of new commercial FPGAs makes this a challenging task. However, it is always possible to modify the contents of a register, a RAM block, or a Look-Up Table (LUT) with a little bit of reverse engineering [372]. It is sometimes also possible to use specific APIs or design tools (*e.g.* Xilinx JBits or FPGA Editor) to make these changes in the configuration bitstream before each evaluation. However, this later method

is much slower than the former method of difference-based partial reconfiguration both due to longer reconfiguration times (if full reconfiguration is inevitable) and bitstream generation time. Xilinx APIs for embedded processors on FPGA do not allow changing the routings but they provide functions for changing the LUT and flip-flop contents. In this way, it is possible to use some of the LUTs as routing resources that can be partially reconfigured by the evolution.

Evolvable Hardware Promises

Evolvable hardware has been claimed (and sometimes proven) to minimise cost, create adaptive fault tolerant systems, able to explore new design spaces and synthesise implicitly defined circuits [127]. Evolvable hardware can be used to reduce the design or manufacturing costs by optimising a circuit or even automatic design of the circuits. Evolutionary algorithms have been used in routing [253] and layout [125] design of circuits. Successful application of evolvable hardware for optimising clock skew in chip manufacturing process leading to about 50% improvement in yield and evolutionary design of a fast low-cost controller are reported by Higuchi *et al.* [155, 185]. Adaptive evolvable hardware applications were reported in data compression [322], audio-visual filtering [335, 392], adaptive hashing [71], and ATM network scheduling [217, 227] and many others [137]. Fault tolerance has been demonstrated in an evolutionary robot controller [154]. Lohn *et al.* [230] also showed that evolution can recover the routing and logic of the circuits at the same time.

Sometimes evolvable hardware is used to synthesise circuits from an implicit high-level behavioural description or input-output examples. Pattern recognisers and classifier systems lie in this category. Input-output example vectors are given or can be obtained from behavioural description of the circuit. They are used to evaluate behaviour of the evolving circuits in form of a fitness function. Digital circuits can be evolved at the gate level or functional level. Higuchi *et al.* explored both of these areas in image recognition, classification of two intertwined spirals, the Iris data set, and 2D image rotation [418].

In traditional design, digital devices are used in the same way that they are designed for. But by relaxing this constraint, evolution is able to explore other innovative ways to use them. Evolution is able to exploit even parasitic properties of the components to satisfy its requirements. Thomson's famous experiment of intrinsic evolution of a tone discriminator on a reconfigurable digital device [364] demonstrated how evolution uses secondary properties of the digital components. In another experiment, he evolved a low-frequency (4 kHz) oscillator using simulated digital gates with random delays between 1 and 5 nanosecond. A recent review of EH success stories and potentials can be found in [137].

Evolvable Hardware Challenges

With all the success, advantages and potentials, EH is also facing serious challenges. One of the major problems with evolvable hardware is its scalability. Much longer chromosomes are needed for describing complex circuits, which implies longer evolution. Moreover, the fitness landscape for digital hardware evolution is generally very epistatic and rugged [129]. That means lots of local minima and deceptive clues for the evolution. Researchers are trying to devise more scalable representations to solve this problem. Koza [206] proposed GP (Genetic Programming) with Automatically Defined Functions (ADF) that may bring modularity and reuse to evolvable hardware. Torresen [367] demonstrated that dividing

the whole system into single-output subsystems can help evolution to solve more complex problems. Miller *et al.* introduced CGP (Cartesian Genetic Programming) [266, 387] and then ECGP (Embedded CGP) [398, 399, 265] and showed their superiority over GP for some problems. Alba *et al.* [5] proposed a parallel hybridisation of simulated annealing and evolutionary algorithms. Vasicek *et al.* used a SAT solver for formal verification of digital circuits solver and given a perfect initial solution were able to evolve simpler circuits and effectively optimise the silicon area needed for the function [386]. This method assumes that a perfect solution already exists. However, the scalability problem is still an obstacle in evolution of complex digital systems [365]. Gordon showed that developmental processes could be a solution to scalability problem in evolving digital hardware [129]. Developmental systems are reviewed in section 2.5.3. There are also other challenges such as fitness evaluation that involves an intractably large number of input vectors, measuring the feasibility factors of the solutions, and putting the solutions in realistic environments for evaluation and measurement [137]. A relatively recent review of these can be found in [137] and [127].

2.5.2 Evolving Neural Networks

It is also possible to use evolutionary algorithms to design, optimise and adapt artificial neural networks (sometimes known as Neuro-Evolution or NE [117]).

There are a few main advantages compared to traditional approaches of designing and training neural networks [101]. Describing a general performance function is usually easier and more flexible than defining an error or energy function (used in learning algorithms) particularly when dealing with recurrent neural networks. It is also possible to coevolve different properties of the neural networks at the same time, or use evolution and learning algorithms together, or even evolve new learning algorithms. These approaches have been pursued in three different ways (or a combination of them): evolving network architectures, evolving weights (evolutionary learning), and evolving the learning algorithms [417]. Experiment on different classification problems showed that evolving only the synaptic weights is a better algorithm than back propagation [103]. A comprehensive review of different methods of neuro-evolution can be found in [417, 103, 101]. In the following section we focus on the most important methods of neuro-evolution of RNNs.

Evolving Recurrent Neural Networks

The real power of neural networks (and particularly spiking neural networks) can be fully demonstrated when they are used in a recurrent network customised to a specific problem. However, no systematic and effective approach for designing recurrent neural networks for a given problem is proposed yet [417]. Therefore, some researchers pursued an evolutionary approach to design of the recurrent neural networks [417, 117]. These methods can be classified in different ways. One of the important aspects in classifying these methods is the representation or the genotype-phenotype mapping. It has a significant impact on efficacy of the genetic operators in finding solutions. Floreano *et. al.* [101] classified the current representation methods into: direct, implicit, and developmental. In direct representations there is a direct mapping between the genes and the properties of the neural network. For example values stored in the genes may directly represent the weights of the network connections. Finding the right

scaling for representing the weights and parameters is the first problem with a few solutions in the literature (such as dynamic encoding or centre of mass encoding) [101]. A more major problem is the issue of individuals with "competing conventions" [326]. This means that the same basic topology and functionality can be described with different order of weights (and consequently different genotypes) leading to unsuccessful crossovers or competing subpopulations. Premature convergence is another classical evolutionary algorithm problem that quickly reduces the diversity of the population and slows down the progress [101].

Sometimes, instead of synaptic weights, only the parameters controlling the neural network (such as number of neurons, probability of the connections, weight distribution, and activation functions) are evolved. This parametric method for evolution of neural networks is called indirect encoding [417].

Stanley and Miikkulainen devised an effective way of evolving topology and weights of increasingly complex recurrent neural networks called NEAT (Neuro-Evolution of Augmenting Topologies) [350]. It starts from a population of very simple topologies (no hidden neurons) and evolves by adding new nodes and connections using mutation, keeping track of the chronological order of the innovations as a solution to the "competing conventions" problem to allow useful crossovers between chromosomes of different lengths. It effectively uses these chronological gene markers to apply complexification and speciation principles of natural evolution. NEAT and its extensions (rtNEAT [350], HyperNEAT [349], and others [117]) was successfully used in a number of complex control [350, 352, 346], and pattern generation applications [347]. NEAT and its extensions are one set of the state of the art techniques in neuro-evolution. While they have many bio-inspired features and are effective in practice, they are not very bio-plausible. The chronological marks on the genes and the direct gene to neuron/connection mapping are two examples of such biologically implausible features. HyperNEAT [349] that uses an indirect mapping from genes to connections is a more bio-plausible method and will be discussed in more detail in section 2.5.5.

Durr *et. al.* [86] introduced another implicit encoding called Analog Genetic Encoding (AGE). AGE can be used for evolving any network and has been used for neuro-evolution. AGE is based on a more bio-plausible approach to neuro-evolution. It uses a variable length chromosome of a user-selected alphabet. Specific short strings represent tokens that mark start, end and regulatory regions of the genes along the chromosome. For, neuro-evolution, each valid gene starts with a device token then input terminal regulatory region, then a terminal token, output terminal regulatory region and ends with another terminal token. Valid genes are extracted from the chromosome the interaction between regulatory regions of the terminals of the neurons specifies how they will connect. Almost all the bio-plausible genetic operators such as deletion, insertion, substitution, duplication, transposition, and homologous crossover is easily applicable on AGE chromosomes. It also allows for noncoding regions in the chromosome. Durr *et. al.* [86] also showed that AGE can outperform NEAT on specific problems. However, it is clear that AGE needs more computation for decoding the chromosome, larger populations and restarting compared to NEAT. Although AGE has a much more plausible approach towards genetic operators and representation it is still far from being a bio-plausible neuro-evolution technique as it directly uses the

gene regulatory network described by the genotype as the neural network.

Other mixed methods of evolution and learning have been also studies. For example, evolutionary algorithms were used among other methods for optimising the performance of the reservoirs [384, 232], which led to some positive results. Chatzidimitriou *et. al.* [63] used NEAT to evolve echo state networks that are trained and evaluated for a problem. Evolino [327] is another example of evolving recurrent neural networks. There are also a plethora of other studies that mix two concepts of evolution and neural networks. Examples, surveys and reviews of this type of studies can be found in [417, 351, 79, 101, 103, 63, 117, 348]. Evolving artificial neural networks with Cartesian Genetic Programming (CGP) has recently ben shown to be more effective than many other neuro-evolutionary methods [193, 1, 244, 369, 194]. However, much work is still needed to evolve topologies, structures, and regularisation/adaptation/learning algorithms at the same time. To make this happen, the evolutionary algorithm must be free to change the size, topology, structure, node properties, and learning algorithm of the RNN or any part of the RNN. An evolutionary system needs emergent properties such as scalability and modularity to be able to work effectively on these different hierarchical levels at the same time. Developmental processes seem to be the nature's solution to these problems. What makes developmental systems special in the context of neuro-evolution, is their ability to grow and regenerate the network in situ that can bring adaptability, scalability, fault-tolerance and self-repair to these networks. The following sections provide a brief overview of the developmental systems with focus on their application in evolution of neural networks.

2.5.3 Developmental Systems

Development is the natural process of cell-division, differentiation, apoptosis, growth and morphogenesis [404, 211] that turns a single zygote cell into a mature organism and maintains it in a healthy structural and functional condition. In a sense, development can be seen as the machinery that determines the behaviour of an organism either directly or indirectly through generating some functional structures (such as nervous system). Development is a parallel, decentralised, and self-organised process [103]. Rust *et al.* count variation, adaptation, regulation, modularity, and robustness as emergent features of biological development [321].

As a more bio-plausible approach to evolutionary computation, artificial development (or computational development) is an abstracted model of biological development believed to be able to introduce emergent properties such as scalability, adaptability, robustness, self-organisation, modularity, growth, regularisation, regeneration, fault-tolerance, and self-repair to the products of evolutionary algorithms [103, 261, 211]. Such systems that use evolutionary algorithms and artificial development are sometimes called Evolutionary Developmental Systems or Evo-Devo systems, which come from the literature in biology [298].

Several mathematical and computational models of development have been introduced by researchers. Among them, Turing's Reaction-Diffusion systems [368], Meinhardt's Activator-Inhibitor models [258, 259], Kauffman's Random Boolean Networks [189], and Lindenmayer systems (L-systems) [221] have been studied extensively and applied to different domains [211].

Rewriting Systems

Lindenmayer systems (L-Systems) [221], its variations (such as bracketed, stochastic, parametric, and context-sensitive L-Systems), and similar rewriting systems are used for developing 1,2 and 3 dimensional arrays [222], which can be then interpreted into 1,2, and 3D shapes and structures, plants, graphs, circuits, hierarchical structures, and modular networks [296, 103, 260, 40].

Context-sensitive parametric bracketed L-Systems can be used to imitate cell signalling, regulation and protein diffusion [103]. One limitation of the rewriting systems is that modelling cell migration needs an external and implausible process (deleting one symbol and inserting it at another location) [103]. Cell migration can be modelled much easier in detailed models. Rewriting systems were originally devised for simulating the biological developmental processes in self-similar organism such as platens and they are not easy to design.

Cellular Systems

The cell is the fundamental building block of life. Each cell is separated from its environment and other cells by a membrane (and an extra cell wall in some cells) that defines its boundary and controls the matter and energy exchange with the environment. Each cell also has a genome that provides the instructions for functions and development of the cell and in turn maybe a whole multicellular organism. The densities of different chemical molecules inside the cell determine its state and the molecules that pass through the cell membrane can be used for signalling and nutrition [404]. Artificial cellular systems have a set of variables presenting the state of each cell [103]. A number of equations, instructions or other rules define the state transition function, describing how the cell state evolves through time. A set of rules also define how the cell communicates with its environment and other cells [103].

Cellular Automata (CA) is the simplest form of artificial cellular systems. It was first introduced by von Neumann in [393] and then studied extensively by Wolfram [402] and others. It is essentially a 1,2, or 3 (or more) dimensional array of cells with a discrete cell space of arbitrary neighbouring (grid, honeycomb, etc.), a finite state set, a discrete time variable, and a state transition function that defines the next states of the cell given the current state of the cell and states of a limited number of its neighbours. There are also variations of CA such as asynchronous, probabilistic, and non-homogenous CAs. In a non-homogenous CA, the state transition function can be time or space dependent. CAs have been used for modelling development among many different processes in nature and technology [103].

It is also possible to use finer resolutions or continuous time and state variables, signalling, and diffusion, or different levels of cellular systems that work together to model biological cellular systems more accurately [103, 211]. Cellular structures can be also used effectively in routing signals both in digital [364, 377] and analog [92, 93] systems. As diffusion and signalling are significant processes in biological development, their bio-plausible simulation is important in this study. Therefore, some diffusion models are reviewed here.

Diffusion Models

Bio-inspired multi-cellular developmental systems usually need to model the diffusion of proteins and other chemicals. These models can be analytical or numerical. In an analytical model, the concentration

of a protein can be calculated as a function of the position of all the protein diffusers (in vicinity of the point in question) and their strength, the diffusion coefficient of the protein in the substrate, and the decay rate of the protein. For example Kumar *et. al.* model the contribution of each diffuser to the concentration at a certain point (C) with Euclidian distance d from the source (diffuser) using an equation of the form [210]:

$$C = C_s \cdot e^{\frac{-d^2}{2D^2}} \tag{2.15}$$

where D is the diffusion coefficient and C_s is the concentration at the source, resulting in a Gaussian radial base function.

However, Rust *et al.* model diffusion using an equation of the general form [321]:

$$C = \frac{C_s}{d^\omega} \tag{2.16}$$

where $\omega \in \{1, 2, 3, 4\}$ is the decay factor (typically 1). Other models exist that also take time into consideration [89].

In numerical models the differential equations governing the process of diffusion is approximated using the Euler method by discretisation of time and/or space. For example Miller used the update rule of general form [261]:

$$C_{t+1} = \frac{C_t}{\omega} + \frac{1}{|\mathcal{N}|\omega} \sum_{i \in \mathcal{N}} C_t^i \tag{2.17}$$

where C_t is the concentration in a cell at time step t and $C_t^i, i \in \mathcal{N}$ are concentrations in the neighbouring cells.

Deterministic or stochastic computation in form of parallel or serial arithmetic implementations can be used to approximate such an update rule [15]. Sometimes very simplistic degenerated forms of this update rule are used to minimise computing resources. For example Roggen used this update rule [311]:

$$C_{t+1} = \max_{i \in \mathcal{N}} (C_t^i - 1, 0) \tag{2.18}$$

in a hardware-based implementation to avoid division operations and multiple additions.

These models are different in terms of bio-plausibility, computational cost and providing the concentration gradient information with a clear trade-off between computational complexity and bio-plausibility.

Cellular Developmental Systems

Cellular developmental systems are similar to context sensitive rewriting systems in many ways. Their state transition functions of neighbour states are similar to rewriting rules. Cell states and cell space in CA resembles the symbols alphabet and cells in rewriting systems. However, in CAs and most of the cellular systems the cell space and number of cells are predefined and fixed, while rewriting systems are capable of creating new cells and deleting old ones. Although this can be an advantage of the rewriting systems, it also makes it quite challenging to implement them in a parallel piece of hardware with limited resources. CAs (and to some extent other cellular systems) ,on the other hand, are very easy to implement in parallel hardware and particularly in FPGAs.

Biological development can be modelled at different levels of abstraction. One extreme is to look at very high-level dynamics of organ formation or cell duplication. Rewriting systems such as Lindenmayer systems (L-systems) [221] are very good examples of such high-level approaches. It is also possible to delve into the details of the biological development process looking for the mechanisms and structures and imitate them to achieve a higher level of bio-plausibility. Multicellular systems of gene regulatory networks (GRNs) with protein folding, diffusion, simulated chemistry, and gene expression and regulation, are examples of the later extreme. There is a spectrum of different models between these two extremes, which have been developed and applied to different applications with a diverse degree of success and performance match [103, 211]. Here we focus on artificial evolutionary developmental (evo-devo) models with an emphasis on bio-plausible models and those that are useful in developing circuits and particularly neural networks.

2.5.4 Evo-Devo Systems

Not only evolutionary computing can be utilised to synthesise developmental processes in a more effective way than synthesising them "by hand", but also, as many believe (for example [17]), developmental systems can also bring evolvability, and scalability, among other features, to evolutionary computing. Moreover, it is much more a bio-plausible solution to evolutionary computing than direct encoding of the phenotype.

Although there is no clear-cut borderline for what is considered to be an evo-devo and what is just an evolutionary algorithm, all the researchers unanimously consider a direct mapping from genotype to phenotype a sign of a non-developmental evolutionary system. Direct encoding aside, Bentley and Kumar [23] differentiated the developmental processes of evo-devo systems into three different classes of external, explicit, and implicit encoding. The first group consist of the indirect mapping that are designed, fixed and not evolved. The developmental process in the other two groups are evolved. Evo-devo systems with explicit encoding are those that evolve a data structure of instructions, which explicitly specify the developmental process. Evo-devo systems with implicit encoding, on the other hand, are those that use a set of rules (instructions) with no or a minimum predefined structure, which can be dynamically activated in parallel based on the context and environmental factors.

In a somewhat similar way, Floreano *et. al.* [103] classified evo-devo systems in the following four categories:

1. Parametric evo-devo systems with a fixed but nontrivial developmental or generative process that maps a set of evolved parameters from genotype to the phenotype: In this class of evo-devo systems, the developmental process is always fixed and not evolved. This will leave the very difficult problem of synthesising an evolvable and useful developmental process to the designer of the evo-devo system.

2. Evo-devo systems that iteratively execute developmental modifications in a fixed order for a fixed number of times while the modifications that are executed repeatedly are evolved. Parallel rewriting systems are good examples of this category with fixed rewriting cycles that are always exe-

cuted in the same order while the axiom and rewriting rules are defined by the genotype and can be evolved.

3. Evolutionary developmental programs that evolve programs using a set of fixed instructions or functions. Here, the order and interrelation of different instructions can be evolved while the basic instructions are fixed. Cellular Encoding [133, 132], HyperNEAT [349], and using GP and CGP for evolving developmental programs are good examples of this category that use a fixed set of basic functions and terminals but evolve graphs that defines how these basic function work together to develop the phenotype. Applications of Cellular Encoding, GP and CGP in evo-devo neural networks is discussed in more detail in section 2.5.5.

4. Evolutionary developmental processes: These are bio-plausible methods that allow evolving both the basic mechanisms and the general process of the developmental system and cannot be classified in any of the above categories. These are generally more detailed and less abstracted models with higher computational complexities. They generally have the potential for better scalability and evolvability but they also usually require evolving the developmental process from scratch that means much longer runtimes for evolutionary algorithm. However, using already evolved seed populations instead of random initial populations to provide the basic and generic mechanisms of development may mitigate this problem. They also allow for interaction of the environment with the developmental process leading to phenotype plasticity [277] during the lifetime of the individual that can bring other adaptive features such as fault-tolerance, regeneration, and self-repair to the evo-devo systems.

After Bentley and Kumar introduced the term "Embryogeny" [23], Stanley *et. al.* used the term "Artificial Embryogeny" in [351] referring to evo-devo systems and acknowledged the importance of both evolvability and feasibility of the developmental systems in the evaluation and selection of evo-devo models. They introduced a taxonomy of evo-devo systems based on the following five dimensions that are also very useful in evaluating the bio-plausibility of the evo-devo systems:

1. Cell fate: How many different methods can be used in the system for determination of the role of a cell in the matured phenotype. Systems with few determination methods are residing at one extreme of this dimension and systems with many methods (as in natural evo-devo systems) are at the other end.

2. Targeting: The way that each cell finds other cells that need to be connected (this is particularly important in neurodevelopment as neurons need to connect to each other in a controlled fashion). This can be done by targeting cells that are at a position relative to the source cell or by using specific chemical markers of the target cells. At one extreme of this dimension are systems that only use relative targeting while systems at the other extreme can only use specific targeting. Natural evo-devo and bio-plausible systems reside in the middle as they can use a mixture of both methods.

3. Heterochrony: How the genotype encoding allows evolutionary changes in the timing and order of the events in the development process. At one extreme of this dimension are systems that are not flexible and there is no way to change the steps or order of the developmental events. At the other end are natural evo-devo systems that can change the order or entirely skip some development stages by small changes in the genome.

4. Canalisation: That is evolution of robust developmental mechanisms that are not sensitive to mutations and can buffer the effect of mutations. At one extreme of this dimension reside systems with precise developmental processes that are sensitive to the mutations. On the other extreme are natural evo-devo systems that are able to utilise stochasticity, resource allocation [351], overproduction/apoptosis, self-regulation, and other means to buffer the effect of the mutations.

5. Complexification: This is the process of evolution starting from simpler phenotypes and incrementally add more features and details to the phenotype by using longer genomes. Variable length genomes, neutral gene-duplication mutations that can lead to two genes with different roles in the later generations, synapsis [351], and speciation are crucial for complexification. At one end of this dimension are the systems with fixed-length genomes with no complexification. At the other extremes reside systems with variable length genomes that are able to utilise useful crossovers, synapsis, and speciation.

They also mentioned a few different abstractions in the current models that made artificial evo-devo systems computationally more efficient than bio-accurate models:

- Cartesian coordinates for space (as in HyperNEAT and CGP)
- Using the real time in the developmental process as a regulating mechanism
- Historical marking of the genes to facilitate artificial synapsis (as in NEAT)
- Using a prepared canvas of cells for the development process to start with (as in most of the cell chemistry systems)

They also differentiated these dimensions and abstractions as design decisions from emergent properties and performance measures such as possibility of evolving complex structures, modularity, gene reuse, symmetry, and efficiency. Stanley *et. al.* [351] also proposed a few benchmarks for evo-devo systems such as evolving symmetry, a specific shape, a specific connectivity pattern, and a simple controller.

The classification of the current evo-devo systems by Stanley *et. al.* in [351] based on the above taxonomy does not include many new models and the ones that are included are not deeply analysed, and their full potentials are not appreciated. Moreover, those five dimensions are not covering all aspects of bio-plausibility of evo-devo systems (*e.g.* online development). Nevertheless, it clearly shows a few trends in the bio-plausibility of the current models. Their classification shows that although both grammar-based (rewriting systems) and cell chemistry systems are potentially able to close the gap

between the artificial and natural evo-devo systems in most of these five dimensions, the cell grammar-based systems were not very bio-plausible and rich in cell fate and canalisation dimensions; speciation and synapsis are usually neglected in the complexification dimension; grammar-based systems tend to use specific targeting while cell chemistry are more inclined towards relative targeting.

Two major issues in developmental evolutionary systems are evolvability and scalability. Many artificial evolutionary systems suffer from stagnation that limits their evolvability. They get trapped in local minima and can't find any progressive path to fitter solutions through subsequent mutations. Neutral networks [343] and gradual complexification of the evolving systems [245, 350], and speciation [243] seem to be possible solutions to this problem. Scalability is another emergent property of the developmental processes in nature. Using an example of overhanging blocks, Devert [78] showed that sometimes an explicit iterative development process is necessary for a scalable evo-devo system. Gordon [128] proposed an evo-devo system based on a simple cell chemistry (Outer Totalistic protein rules) for evolution of digital circuits and showed that a developmental process can in fact enhance the scalability of evolutionary digital circuit design.

Another important aspect of the evo-devo systems is the distinction between developed and developing systems. Similar to the difference between evolved and evolving systems, a developed system will eventually mature and stop developing and the genetic code and the developmental process will have no influence on the functioning of the product. On the other hand, in a developing system the developmental process continues to run during the lifetime of the individual that can play a major rule in the adaptivity and maintenance of the system. Roggen *et. al.* [311] called these systems online developmental systems and suggested that many potentials of evo-devo systems lie in this area. They, however, admitted that using offline development can save computational resources [311].

Backed by such insights, many researchers proposed richer and more bio-plausible models of evo-devo systems. Apart from those evo-devo neural network systems that will be reviewed in more detail in the next section, some of the important and relatively bio-plausible evo-devo systems with general applications are briefly reviewed here.

Liu, Tyrrell, and Miller [223, 224, 261] proposed an evo-devo model, based on cell chemistry using CGP for evolution of the cell dynamics, and applied it first to the french flag benchmark problem [261] and used it later for intrinsic evolution of digital circuits [223, 224]. Their model successfully exhibited fault tolerance to transient damages. In [223] they conjectured that using cell chemistry for long-distance signalling is necessary for achieving robustness in the solutions. Roggen *et. al.* [311] proposed an online distributed developmental system suitable for multicellular systems in hardware based on a simple cell chemistry and applied it to robust evolution of different types of patterns (uniform, checkerboard, Norwegian Flag, complex CA-generated) to show its scalability and robustness.

Bentley introduced Fractal Gene Regulatory Networks (FGRNs or Fractal Proteins) [25, 24], which uses a fractal-controlled genotype to GRN mapping. It is based on a variable-length genome of genes with two regions. The first region, cis-regulatory site, determines the conditions of the gene expression. The second region, coding region, encodes the protein that can be synthesised by this gene. In Fractal

Proteins system, each protein shape is defined using three real numbers, describing a square subset of the Mandelbrot set. Different protein shapes interact with each other in an artificial chemistry to result in a merged protein shape that will then interact with the cis-regulatory region of each gene (also coded as three real numbers and a couple of thresholds values) to give rise to a gene expression probability. When a gene is expressed, the concentration of the protein encoded in the coding region is increased by a function of the concentration of all proteins interacting with the cis-regulatory region of the gene. The probabilistic nature of the gene expression, complex shapes of the fractal subsets, and non-linear interactions of the protein shapes all seem to contribute to interesting properties of this bio-inspired system. Bentley compared human-designed programs, GP evolutionary programs, and FGRN developmental evolutionary programs facing damage and showed that the FGRN developmental program can exhibit graceful degradation [27]. Fractal Gene Regulatory Networks have been successfully applied to function regression and evolving controllers for robots [26, 419], single and joint pole balancing [208], and evolving algorithms for approximating π [207]. Fractal Gene Regulatory Networks [25] showed a high level of bio-plausibility and evolvability [24]. Using very rich domains of fractals for imitating the protein-protein and gene-protein interactions, dynamic indirect mapping of the genotype to the GRN that then controls the dynamics of the cell is much more bio-plausible than other methods using neural networks or GP for control of the cell dynamics.

Apart from cell chemistry and grammar-based developmental systems, there are also generative systems that abstract the explicit process of development and growth through time into a time-independent function. Methods such as CPPN (Compositional Pattern Producing Networks) [347] do not need to simulate all the developmental events through time to generate a pattern in space thus can be classified as an abstraction of the development using a generative process. CPPN-NEAT can be used to evolve a network of mathematical functions with co-ordinates as inputs and structures attributes in that coordinates as outputs. CPPN has a few advantages over developmental systems: computational efficiency, definition of structures in infinite resolution, perfect damage recovery, and possibility of using user-defined biases. Some adaptive features of online development, such as robustness and regeneration, can be produced by iterating the CPPN so that it reacts to the changes in the environment. However, it lacks a mechanism for using local environmental information other than what is already generated by CPPN itself, as demonstrated by Devert in [78]. Some also argue that the sequential nature of an explicit developmental process can provide evolution with useful fitness information during development [203].

A plethora or different evo-devo models for different applications have been proposed by different researchers. Famous other examples of grammar-based and rewriting evo-devo systems are Jacob's system for evolution of fractal shapes [174], Kitano's matrix rewriting systems used for evolution of neural networks [197], Koza's tree-based rewriting system for evolution of 2D shapes [204], and Hornby's system for evolving 3D shapes [162]. Among cellular and cell-chemistry evo-devo systems based on diffusion of morphogens, Astor and Adami's [10] and Hampton and Adami's [142] models for evolving neural networks, Bentley and Kumar's Implicit Encoding for evolving 3D shapes [23], Bentley's Fractal Proteins [25] for evolving GRNs (Gene Regulatory Networks), Bongard and Pfeifer's Artificial

Ontogeny[48] for evolution of embodied agents, Eggenberger's differential gene expression system for evolving 3D organisms [88], Kitano's model of neurogenesis [198], and Miller, Harding and Banzhaf's Developmental Cartesian Genetic Programming [263, 146] and Self-Modifying CGP [146] used in evolution of flags and in many other problems [146] are particularly notable. In the next section focus is on the evo-devo systems used for evolving neural networks.

2.5.5 Evo-Devo Neural Networks

For tackling the challenging task of evolving complex and scalable neural networks and artificial brains many researchers turned to evo-devo systems. Therefore a major part of the evo-devo systems are either designed for evolving neural networks or have been applied to this problem. The neural network evo-devo systems can be categorised into four different types of neurodevelopment: Abstract, Parametric, Explicit, and Implicit.

Abstract Neurodevelopment

The first and most abstracted model of evo-devo NN Systems use an implicit development process that abstracts out the iterative, time-dependent, distributed and local processes of development replacing it with an evolvable function of space (and sometimes time and local variables) describing the connectivity, parameters, and other properties of the neurons and the network, based on their distribution in space. HyperNEAT [349] and its extensions (Adaptive HyperNEAT [304], ES-HyperNEAT [305], HyperNEAT-LEO [388]) are representatives and the state of the art techniques in this group.

HyperNEAT uses NEAT with its historic marking of genes for synapsis, speciation and gradual complexification to evolve CPPNs (Compositional Pattern Producing Networks) [347] with 4 inputs as x,y coordinates of the pre and post-synaptic neurons. Each output of the CPPN can then give the weight of a synapse between two neurons in each layer of the network. Adaptive HyperNEAT [304] allows to evolve NN learning rules by using CPPNs as an evolvable function of previous synaptic weight, neuron locations, and activities. This shows how HyperNEAT can be extended to use local data to abstract local developmental processes using a function. However, as the authors noted, there is still a tradeoff between the generality of the model and the computation cost [304].

Instead of leaving it to the user to specify the locations of the neurons in the substrate (as in HyperNEAT), Evolvable-Substrate HyperNEAT (ES-HyperNEAT [305]) distribute the neurons based on the variance of the synaptic weight CPPN function. In HyperNEAT the expression of links (existence of synapses) between neurons is specified by a threshold on the weight, while in HyperNEAT with Link Expression Output (HyperNEAT-LEO [388]) a separate CPPN, which can be seeded with a biased towards modularity with local connections, controls the expressions of the links. All of these extensions of HyperNEAT can improve the performance, evolvability and scalability of the system for evolving NNs. A set of experiments showed that ES-HyperNEAT and ES-HyperNEAT-LEO can facilitate the evolution of modular and multimodal (for more than one task) NNs , outperforms the original HyperNEAT, and allows complexification of the NNs as well as CPPNs through evolution [306].

Although one of the main points of abstracting development into a CPPN was to avoid the computational complexity of the explicit iterative developmental processes, the practical, scalable, evolvable and

somewhat more bio-plausible versions of this technique such as ES-HyperNEAT-LEO add extra computation to the development process that reduces the computation cost gap between these techniques and explicit cell-chemistry methods. This group of evo-devo NNs rely on complex mathematical functions and assume that any connectivity is feasible. They are particularly targeted for software-based implementations of the neural or developmental processes and their hardware-based implementations may prove impractical.

Parametric or External Neurodevelopment

The second group are systems based on parametric development [103] or what Bentley and Kumar call External Embryogeny [23]. In these systems the process of neurodevelopment is not evolved and only the parameters that are fed into a hand crafted generative process are evolved. Manual synthesis of evolvable and scalable phenotype-genotype mappings for this group of evo-devo NNs are very difficult and most of the researchers avoid it when it comes to complex neural network systems. The parametric model for generation of NNs by Harp *et. al.* [148] is an example of such systems. A few more examples can be found in [417].

Explicit Neurodevelopment

The third group of evo-devo systems use a program that is encoded in a data structure and is run explicitly to generate and modify the NN. Bentley and Kumar classify these systems as Explicit Embryogenies [23]. Floreano *et. al.* divide this group into two subgroups [103]: The first subgroup include those systems that use a grammar-based rewriting system with a fixed order and number of operations while the basic operations (rewriting rules) can be evolved. The final product of the parallel rewriting cycles represent the NN connectivity and parameters. Kitano's Matrix Rewriting system for evolution of NNs [197] is a good example of such systems. The second subgroup includes those systems that evolve the program (that might also use rewriting rules, self-modification, recursion, or other techniques to achieve modularity), which in turn generates or modifies the NN using a fixed set of operations. Cellular Neural Encoding by Gruau [132] is a good example of such systems.

Implicit Neurodevelopment

Another group of evo-devo NN systems are based on cell-chemistry, morphogens, and evolution of GRNs or other similar dynamical systems that govern the development of the NNs. These systems are what Bentley and Kumar call Implicit Embryogenies [23].

Evo-devo NN systems by Cangelosi, Nolfi, and Parisi [56], Kitano [198], Dellaert and Beer [76], Eggenberger [87], Bongard and Pfeifer [48], and Jakobi [177] can all be considered examples of this type. Astor, Hampton and Adami also proposed a new developmental model for evolution of robust neural networks and reviewed some other developmental neural networks of this kind [10, 142]. All these models are relatively more similar to biology compared to the former types of evo-devo NNs in one way or the other. However, they use different levels of abstraction and complexity, show different performances, evolvability, scalability, and robustness, with different computation costs and they can not be easily compared due to very diverse applications and benchmarks researchers used to evaluate

these models. As a general trend it is clear that more detail (complexity) and lower levels of abstraction always increase the computation cost but not always lead to desired emergent properties or improved performance. Seeking modularity, allowing complexification (of genotype, phenotype, and fitness function), exploiting neutral mutations in the genotype-phenotype mapping, distributed local processes and interactions, and utilising useful environmental and network activity information are examples of good tips for an evo-devo NN designer.

Online Neurodevelopment

Online development has many advantages over using development as a generative process only to create a mature solution and then stoping it. Fault-tolerance, regeneration and self-repair, adaptability, effective use of fitness values during the development, and utilising the activity of the neurons and synapses to guide neurodevelopment are a few examples. Federici [95] used a recurrent neural network to control the cell dynamics for development of neural networks and showed that development can bring regeneration and fault tolerance to spiking neural network robot controllers.

Khan, Miller, and Halliday [191, 192] proposed using online development of CGP programs to evolve both electrical and developmental behaviour of the bio-plausible NNs and showed very interesting bio-plausible emergent properties in their results. This work is particularly unique in the sense that it takes into account different bio-plausible properties and behaviours of the biological neurons (such as nonlinear synaptic interactions and neuron health) instead of limiting the development to modification of synaptic weights with linear summative interactions.

Theoretically, many of the methods mentioned earlier can be also adapted to support online development. However, the computational cost of continuous running of the development process through the life time of the product is usually prohibitive and needs special attention.

Bio-plausibility in Evo-devo NNs

In [192], Khan, Miller, and Halliday argued that there are much more bio-plausible features than synaptic weights and connectivity that need to be included in evo-devo NNs. They also showed that by bringing many bio-plausible details using their method it is possible to control an agent playing arcade game with only a single neuron [190]. Also, Rust *et al.* claimed that to exploit the potential of artificial neural systems to the fullest extent, more bio-plausible details of the neural development should be modelled in the artificial developmental systems [321].

There are many other systems that are either the basis of the aforementioned newer models or similar. Reviews of many other evo-devo NN systems can be found in [417, 351, 55, 311, 310, 101, 103]. Among them are those that are designed or suited for hardware implementations, reviewed in the following section.

2.5.6 Hardware-based evo-devo NNs

Researchers have sought to create hardware-based evo-devo neural network systems capable of evolving, developing and learning *in situ*, adapting to the given problems and environments. These are called POE systems as they are aimed to show these capabilities in all three aspects of Phylogeny, Ontogeny and,

Epigenesis of an organism [342]. POE systems can use online development that brings fault-tolerance and self-repair features to the neural networks [370] and similar systems. Online development in these systems allows developmental process to use neural activity and other environmental factors as useful information for adaptivity. It can also provide evolutionary process with useful fitness information during development and learning. Evo-devo spiking neural networks that could be implemented on the POEtic chip [94, 366, 270, 271] would be good examples of such systems. PEOtic Chip is a custom-designed reconfigurable integrated circuit as a hardware platform for POE systems. It is based on a layered architecture known as POEtic tissue [370] that dedicates three separate layers to Phylogenic, Ontogenic and, Epigenesis processes.

One of the challenges in the field of hardware-based evo-devo systems is the difficulties of implementing cell devision in silicon [361]. Although reconfigurable chips and FPGAs provide some flexibility to achieve similar processes, cell devision remained a challenge. Most of the solutions implicitly allow new cells to take over hardware resources without explicitly dividing anything as such alteration of hardware substrate is physically not possible [361]. Many approaches abstract out the cell division process into the cell differentiation process or reduce it to the simpler process of self-replication [361]. Tempesti *et. al.* reviewed a number of usual approaches to hardware-based cell division and differentiation in [361].

Routing of the signals is another challenge in hardware-base evo-devo NNs. POEtic tissue address this by dedicating two set of routing resources on separate layers for local and global circuit switching and communication between cells [94, 366, 270, 271]. Thoma and Sanchez reviewed a number of hardware-based routing algorithms for circuit-switched communication between cells in [362]. Other systems tend to use packet-switched NOCs (Network-on-Chips) as they allow much higher connectivity density but may suffer from packet delays and jitter noise depending on the network activity [147, 272, 58, 62, 60, 59].

Evolvable hardware has been previously used for evolving spiking neural microcircuits in FPGAs. For example, Upegui et al. evolved a fully connected recurrent spiking neural network of 30 simplified LIF neurons with 30 synapses each on a Xilinx Spartan FPGA [374]. However, the number of neurons and synapses, general architecture of the network, and the neuron parameters were fixed during the evolution and no developmental process was used. There are also other hardware-based adaptive neural systems that may be modified for evo-devo neural networks [60, 62]. However these are not specifically designed or suited for intrinsic developmental processes.

Inspired by the seminal work of Thomson [364] with a cellular structure on Xilinx XC6264, several multi-cellular developmental systems for FPGAs have been designed by Haddow and Tufte, Liu, Miller and Tyrrell, Gordon, and many others, cited above and in [311]. One of the first reported works on hardware-based evo-devo neural networks is CAM-Brain project by DeGaris *et.al.* [120, 74], based on Cellular Automata and implemented in Xilinx XC6264 FPGAs. They report experiments on a system of about 1000 neurons.

Moreno *et. al.* [270, 271] reported implementation of a spiking neural network on POEtic tissue

consisting of a 4x4 array of POEtic chips [370]. Their model is based on an optimised serial implementation of LIF neuron model with bio-plausible synapse and STDP (Spike-Time-Dependent-Plasticity) learning [366] on the reconfigurable substrate of POEtic tissue that allows neighbouring cells to reconfigure each other for fault-tolerance and self-repair. Unfortunately, only one such neuron can be fit in a single POEtic chip due to size limitations of the actual POEtic chips. Therefore, time-multiplexing was used to simulate a network of 10,000 neurons. This, however requires loading the parameters and neuron variables into the chips every 150 cycles for 625 times for one network update. This was still fast enough for realtime processing of a 384x288 pixel video stream at 50 frames/sec. However, at this stage, the developmental and evolutionary potentials of the POEtic tissue was not exploited fully. Their implementation uses a parallel hardware implementation of breath-first search algorithm for dynamic routing of axons and dendrites that is initiated by the unconnected neuron input and outputs. Allen *et. al.* [6] also report implementing a very small network consisting of 3 spiking neurons capable of STDP learning and evolution on an FPGA and a POEtic chip.

Later, Roggen *et. al.* worked on the evo-devo features of the PEOtic tissue and presented a comprehensive review of the hardware-based evo-devo systems including neural networks in [311]. They introduced a new classification of developmental systems, and stressed the importance of hardware-based (intrinsic), online, cellular and distributed developmental systems arguing that a lot of desired features of developmental systems such as adaptivity, fault-tolerance, scalability, speed, and robustness are achievable with such evo-devo systems. They also proposed a hardware based evo-devo spiking neural network system, based on diffusion of morphogens with very simplistic cell chemistry and neuron model, and applied it to character recognition and robot navigation tasks successfully. The largest network they implemented comprises a 8x8 grid of 64 neurons and a maximum of 12 synapses (inputs) per neurons. They demonstrated improvements in fault-tolerance, and scalability of the system compared to using a directly encoded genome. Although their design was generic and could be implemented in any reconfigurable platform, their system was initially designed for POEtic chip [370], and prototyped on a FPGA for experiments [310, 311]. However, the range and pattern of neurons connectivity were limited to six fixed local connectivity patterns and a simplistic leaky integrate and fire soma model was used in their implementation [311].

Recently, Upegui *et al.* introduced a dynamic routing algorithm to produce nature inspired activity dependent synaptogenesis in the cellular bespoke reconfigurable chip, Ubichip [375, 380]. Ubichip is a reconfigurable platform, as part of a greater project call Perplexus, aiming at ubiquitous and embedded computing for complex systems [324, 379, 375, 378, 380]. They showed that the network activity information can be used effectively for neurogenesis and the resulting network architectures resembled those of biological neural networks. Although their hardware-based routing algorithm is a very fast implementation of its kind, their current design may face some scaling limitations (due to using long combinatorial signals sensitive to delays) [375, 380]. Moreover the number of possible input synapses for each neuron is fixed and defined *a priori* [375, 380].

Another notable study is EMBRACE-FPGA project [272, 62, 284], aiming at prototyping the EM-

BRACE mixed-signal reconfigurable custom SNN chip. The FPGA prototype of the EMBRACE chip is able to evolve synaptic weights and neuron thresholds of a pre-specified network architecture. The analog LIF neurons are modelled by soft core processors on the FPGA and a NOC (Network-on-Chip) is responsible for interneuron spike communications. The EMBRACE system successfully evolved spiking XOR, an inverted pendulum controller, and a classifier for Wisconsin breast cancer dataset. Although many feasibility measures such as scalability, fault-tolerance, robustness, performance and hardware cost are considered, bio-plausibility is not a priority in EMBRACE-FPGA project and the system does not use a developmental process or any bio-plausible features other than spiking neurons and a genetic algorithm for achieving scalability and fault-tolerance.

Yet none of these models are quite suitable for a developmental neural microcircuit capable of regeneration and growth on FPGA. Evidence suggests that structural plasticity [65] and wiring delays [66] play major roles in the brain. The placement and wiring of the neurons are also optimised for the high interconnectivity in the brain [64]. In contrast, most of the existing evolvable hardware neural network models (e.g. [311, 374, 380, 271]) are not capable of flexible neurite growth in silicon. They either are typically restricted in terms of number of inputs per neuron or impose constraints on the patterns of connectivity and/or placement on the actual chip mostly due to implementation issues. Some of them do not allow heterogeneous networks with flexible parametric neurons and learning rules as important bio-plausible features or use very simplified developmental processes. They either keep the silicon area low and gain a high speed by using excessively simplified neuron models or use bio-plausible models and quickly run out of silicon area forcing them to use time-multiplexing. Although most of them are originally designed for custom chips (Ubichip [375, 380], and POEtic chip [370] for example) they ended up being prototyped on FPGAs due to availability issues such as high NRE costs of ASICs and even after fabrication of the actual chips they are limited to small-scale chips mainly due to financial reasons.

2.6 Summary

Bio-plausible approaches to spiking neural network such as Liquid State Machines (LSM) and Hierarchical Temporal Memory (HTM) are gaining popularity and success. However, both of them need high computational power of direct hardware implementation for scalable solutions. There is still a lack of systematic method for designing the network architectures for given problems. However, evolutionary approaches have shown promising results.

A review of the neuron models shows that very bio-plausible computational models are available but they are very complex and computationally expensive. Recently, Izhikevich has proposed a flexible, parametric, computationally simple, behaviourally bio-plausible model for simulation of spiking neurons. However, almost all of the Liquid State Machines and all of the hardware-based neural systems use very simplistic and implausible models such as LIF or even degenerated implementation of it that are incapable of showing a vast part of biological neurons behaviours. There are few good hardware implementations of relatively bio-plausible synapse models and STDP learning with higher silicon areas.

In the field of evolutionary computing, although some bio-plausible neurodevelopmental systems

have been used for evolution of neural networks, they are usually very simple and not modelling details of the gene-protein and protein-protein interactions. On the other hand, the detailed and complex systems are very slow and computationally expensive. Despite some general trends, it is not yet quite clear incorporating which one of these complexities in the neuron models and developmental processes can lead to benefits both in terms of performance and efficiency and in terms of emergent properties such as evolvability, scalability, fault-tolerance, and robustness. The complexity of bio-plausible models has also prevented researchers from implementing such bio-plausible models in hardware. The most bio-plausible evolutionary neurodevelopmental model in hardware is based on very simple diffusion of morphogens on a very homogeneous neural substrate.

New findings in biology provide us with more accurate and better models of biological systems that can be used to enrich our bio-inspired designs. Although general trends point to promising results from pursuing a bio-plausible approach to evo-devo neural networks in hardware, endeavours were restricted to a few attempts on custom chips with limited success, mainly due to the restricted availability of such fabricated custom chips and their small sizes. Even though FPGAs are increasingly popular, cheaper, larger and powerful, it is yet to be investigated how these new potentials can contribute to feasibility of more bio-plausible evo-devo spiking neural networks. This has been the motivation for the present work to explore possibility of exploiting the latest biological models and new FPGA technologies to achieve higher levels of bio-plausibility in such systems.

Chapter 3

Hardware Platform

One of the first challenges in achieving bio-plausibility in an FPGA is the choice of the hardware platform. It is also an important factor in this study as it can impact both the applicability and generality of the challenges, trade-offs, and constraints discovered by the study. Hardware platform selection has been always one of the challenges in implementation of evo-devo systems and evolvable hardware in general as the limitation of the hardware directly impact all aspects of the whole system. In case of this specific study, the FPGA platform must be selected in a way that facilitates the feasibility study of such bio-plausible models on the latest widely available technology. At the same time, practical factors such as availability of the platform for this study and design time frames are also of importance. In the following sections the platform selection criteria and the trade-offs involved are discussed, and based on aims and limitations of this study, an FPGA and a hardware platform is selected.

The choice of hardware platform is a crucial decision. It has a substantial influence on the quality of the system and the size and complexity of the potential applications. With the maturity of digital VLSI technology and its continuous progress, digital technology seems to be a good candidate for realisation of neural networks. Compared to analogue technology, digital circuits are easier to fabricate and provide simpler solutions for storing synaptic weights [201]. Evo-devo neural networks on digital hardware has been implemented on two types of platforms: Bespoke chips (*e.g.* POEtic chip [370, 311]) and commercial FPGAs (*e.g.* [311]). The former, by definition, has a higher performance and better features but was limited to a few privileged institutes and typically small chips (compared to commercial FPGA sizes) due to high costs of VLSI design and fabrication. FPGAs are in contrast ubiquitous, cheaper than such custom chips and, as predicted by Moore's law, will be followed by future generations with new technologies and features. Practical evaluation of the current FPGA technology for implementing bio-plausible evo-devo neural microcircuits can provide recommendations for both FPGA manufacturers and designers of Bespoke chips. As this study is by definition focused on the FPGAs, in the following section, we discuss the factors, trade-offs, constraints and challenges in the selection of an FPGA based hardware platform.

3.1 Selection Criteria

To select the right platform a set of selection criteria should be defined. These criteria can be defined based on the factors already known to impact the bio-plausibility and feasibility of the system, trade-offs between these factors, and the corresponding constraints that are usually present in research studies or engineering design projects. A set of such factors are analysed here based on the experience and literature:

1. Cost: The total cost of the hardware platform is always an important and limiting factor. A large portion of the hardware cost is the unit cost of the FPGA device. There is a trade-off between FPGA cost, and its size, speed and its other useful features. Almost all of these desirable properties add to the cost. The cost of the hardware platform that is mainly determined by the per unit cost of the FPGA is constrained by the project budget (here, a maximum of £1000 for hardware platform).

2. Popularity and Prevalence: The only factor that has negative correlation with the cost of the FPGA is its popularity and prevalence as it can increase the demand and drop the per unit cost of the device. However, all those factors and features that can add to the cost can also increase the popularity and prevalence of a device. It is also affected by other factors outside of the scope of this study such as marketing strategies of the manufacture and popularity of the device in consumer electronics, etc. As a constraint for this study it is necessary that the platform or similar devices remain available to the research community at a reasonable cost in future. Discontinuity of the Xilinx XC6264 device is an example of such unfortunate faith for a family of devices so useful to evolvable hardware community. This factor directly contributes to the availability measure of the feasibility defined in section2.2.

3. FPGA Performance (Speed): Running speed of the FPGA (dictated by propagation delays and maximum clock frequencies) directly affects the simulation time and performance of the whole system as a feasibility measure. FPGA Performance has an impact on the computational power and thus FPGA capacity for more bio-plausible models. Performance has a fundamental trade-off with the power consumption of digital VLSI chips as well. Availability of the same chip in different speed grades is also notable, which can give designers more options to play with the trade-off between the performance and the hardware cost.

4. Size and scalability: Size of the FPGA limits the amount of hardware resources that can be dedicated to the bio-inspired system. It has an effect on the total computational power of the FPGA and its capacity for higher levels of bio-plausibility. The scalability of the hardware platform requires the availability of larger devices from the same family and/or possibility of interconnecting devices. Interconnecting devices might be also limited by the number of available device I/O pins. The size of the FPGA or using many interconnected FPGAs affect the scale of the application (here, number of neurons and synapses) as one of the feasibility measures defined in section 2.2. The hardware platform should provide enough FPGA resources for the experiments and application. Therefore it can pose constraints on the size and scalability of the application. Assuming

a high level of parallelism, size and scalability of the hardware platform can improve the simulation time. On the other hand, bigger devices take more time to reconfigure (even using partial reconfiguration) and therefore size can increase the reconfiguration time. Reconfiguration time and simulation time contribute to the total experiment time, as another feasibility measure (performance) defined in section 2.2.

5. Power consumption: Energy consumption of the hardware platform is a function of the size of the FPGA and number of interconnected FPGAs. This, in turn, affects the scalability of the whole system as one of the feasibility measures defined in section 2.2. However, this factor is usually constrained only for very large-scale systems.

6. (Dynamic) Partial Reconfiguration: These features allow us to speed up the process and perform the development and growth on the hardware by reconfiguring only part of the system while the rest is untouched or even running (in case of dynamic partial reconfiguration). Without partial reconfiguration, we are bound to reconfiguring the whole FPGA for every new solution (or for each development step), which dramatically increases the experiment time.

7. Reconfiguration speed: Since for evaluating the fitness of every single solution during the evolution, the FPGA will be reconfigured (even if partially), this will directly affect the reconfiguration time and contribute to the experiment time.

8. Data communication bandwidth: An FPGA platform is usually connected to a PC or host computer that reconfigures the device for the first time and might be needed for managing the input/output vectors through the evaluation of the solutions. Therefore, data transfer speed between the host computer and the FPGA platform can also have an impact on both reconfiguration and simulation times. High-bandwidth communication interfaces naturally add to the hardware cost.

9. Interfacing: Ease of connection to a PC and/or other I/O devices both for reconfiguration and evaluation of the neural network. For evaluation of the system on different problems it would be useful to feed different data sources through a PC or directly through I/O devices (*e.g.* webcams, sensors, etc.). These interfacing devices can be part of the FPGA (as hard IP cores on the FPGA chip) or on the hardware platform. In both cases a wide range of interfacing options increases the flexibility of the system but adds to the hardware cost.

10. Indestructibility or validity checking: Meaning that it is impossible to damage the chip using an incorrect configuration or it is possible to check the validity of the configuration bitstream before programming the chip. It effectively simplifies the evolvable hardware processes and relaxes some of the constraints.

11. Embedded Processing: Availability of processor core(s) on the FPGA itself enables the FPGA to potentially run some of the evolutionary or developmental processes on-chip and perform complex I/O tasks. This increases the flexibility of the system and contributes to the ease of use. Some FPGAs include a hard processor core or allow to implement soft processor cores on the FPGA

fabric. Using soft processor cores instead of hard processor cores allows us to scale the processor and customise its features depending on our needs. However, soft processor cores are usually 5 to 10 times slower than their hard IP-core versions.

12. Observability: Possibility of monitoring and debugging the behaviour of the FPGA is an important factor both for verification and evaluation, and in experiments on fault-tolerance and robustness.

13. Reliability: Reliability of the whole system, as one of the feasibility measures, is affected by the reliability of the FPGA device among other design and environmental factors. Some FPGA families are also available in radiation-hardened (rad-hard) versions that are resistant to damages and errors caused by high-energy electromagnetic radiations and particles. These chips are usually used in aerospace applications or systems in harsh environments. Different physical and design measures are taken to introduce this feature into FPGA chips, which significantly affect the cost, power, size, and performance of the device. Although these techniques significantly increase the reliability of the FPGA at the physical level and therefore adds to the reliability of the whole system, the rad-hard versions of the FPGA devices are usually available much later than the rest of the device family, which affects the availability of the system as a feasibility measure.

14. Ease of use: A fast learning curve for using the platform and software tools, availability of documentation and support, or previous familiarity can significantly reduce the design and testing time and complexity.

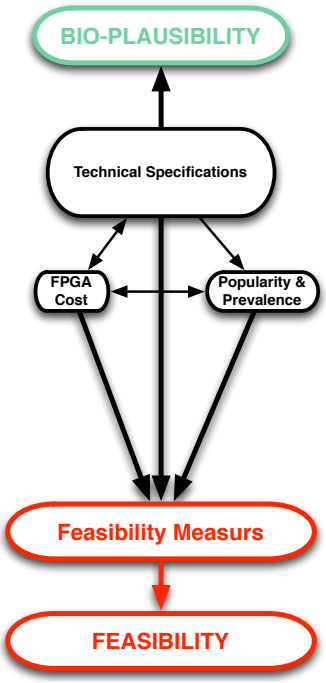


Figure 3.1: Trade-offs and interactions between cost, popularity, and technical specifications of an FPGA, and how they are related to feasibility and bio-plausibility of the whole system.

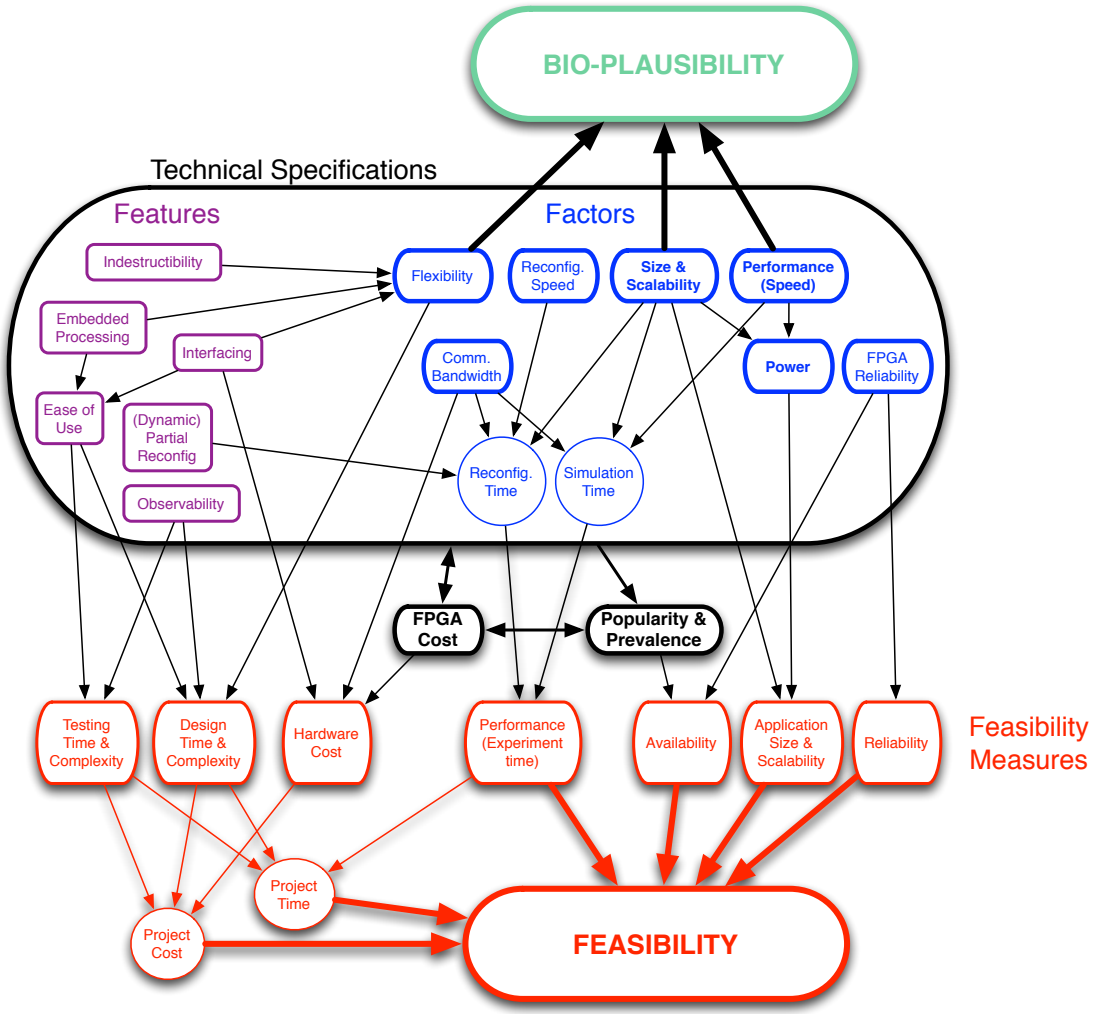


Figure 3.2: Detailed trade-offs and interactions between different technical features and factors involved in the selection of the FPGA platform and how they affect the feasibility and bio-plausibility of the whole system.

Figure 3.1 shows the general trade-offs and interactions between technical specifications of the FPGA, its cost, and its popularity and prevalence. Figure 3.2 depicts the details of the above analysis showing technical specifications (features and factors in purple and blue) and feasibility measures (in red) and how they might be related to feasibility and bio-plausibility of the whole system.

3.2 FPGA Selection

Selection of the FPGA device and platform for this study is heavily constrained both by project time and budget and availability and popularity factors. The hardware budget in this study was limited to £1000. The time for exploration, investigation, design, development, verification and testing and experiments should also fit in the timeframe of this research. The selected device needs to be a good representative of the latest available technology that provides highest scalability and performance and also a popular and prevalent choice that promises lower costs in future. The performance of the system must be acceptable for research experiments if not fast enough for real-life applications. Size of the application should be enough for simple experiments and must be preferably scalable. Final system must be reliable enough

Table 3.1: Major FPGA Manufacturers, their market share, focus, main FPGA device families and their reconfiguration features (at the time of this study, 2007).

Manufacturer	Market share	Focus	Device Familes	Reconfiguration
Xilinx	$> 50\%$	SRAM based, general logic	Virtex and Spartan	Dynamic Partial
Altera	$> 30\%$	SRAM based, general logic	Stratix, Cyclone	Only full
Lattice	$\approx 7\%$	SRAM based, Hi-speed	LatticeXP	TransFR
Actel	$\approx 7\%$	Flash, Anti-fuse, mixed signal, low-power	ProASIC, Axcelerator, Fusion	Only full
Atmel	$< 5\%$	Low density logic and DSP	AT40KAL	Dynamic Partial

to give conclusive results for research. FPGA size, performance, and flexibility are objectives that can allow higher levels of bio-plausibility in the system.

Table 3.1 shows that the FPGA market is mainly dominated by two major companies: Xilinx and Altera. They share a majority of the market and compete closely by introducing the latest technologies. The general architectures are similar to some extent and fundamental techniques are copied quickly by the other party. Therefore, it must not be very difficult to port a design from one system to the other system.

A key feature, affecting the overall performance of the system, is dynamic partial reconfiguration. Altera FPGAs did not support partial reconfiguration at the time. This feature was introduced recently (2010) in Altera's latest family of 32nm FPGAs (such as Stratix V) [8]. Without partial reconfiguration the reconfiguration time and thus evaluation time of each solution during evolutionary process will be proportional to the size of the FPGA. Although virtual FPGA method can be used to introduce partial reconfiguration, as discussed in section 2.5.1, it requires 4.5x more hardware resources to implement the same logic [372]. Neither virtual FPGA technique nor complete reconfiguration result in a scalable solution, and in practice using partial reconfiguration is inevitable to attain a compact and fast design. Other manufacturers of FPGAs with partial reconfiguration capability are Lattice and Atmel [255]. The dynamic partial reconfiguration technique offered by LatticeXP FPGAs from Lattice involves reprogramming the on-chip non-volatile memory while FPGA is working and then halting the FPGA that effectively freezes IO pins during the quick reconfiguration of the configuration SRAM [215]. Although this type of dynamic reconfiguration allows updating the hardware in situ, it poses an extra buffering overhead on the system and requires stopping the whole FPGA for reconfiguration that means the circuit that is controlling the reconfiguration process needs to be off the chip. AT40KAL devices by Atmel also support dynamic partial reconfiguration but they have much lower densities compared to Xilinx and Altera FPGAs. Both Virtex and Spartan families of FPGAs by Xilinx support dynamic partial recon-

figuration. At the time of this study Xilinx was offering the largest FPGA capable of dynamic partial reconfiguration (Virtex-5) and had significantly larger market share than Atmel and Lattice. Spartan 3 was the latest group of devices from Spartan family that constitute the cheaper and lower density devices from Xilinx.

Virtex-5 was the latest family of FPGA devices from Xilinx. It featured 65 nano-meter fabrication technology, ExpressFabric™, 6-input LUTs, up to 330K logic cells, RocketIO™serial transceivers, built-in PCI Express™endpoint and Ethernet MAC (Media Access Control) blocks. Xilinx provides two different soft processor cores (MicroBlaze™and PicoBlaze™) with C programming language support and a Linux-based operating system (in case of MicroBlaze). Most of these features are also available in Altera FPGAs (with slightly different forms and names). Table 3.2 shows a comparison of SRAM-based FPGA devices from different manufacturers. At the time of this study, due to the lack of partial reconfiguration feature in Altera FPGAs, significantly lower densities of Atmel, Altera and Lattice FPGA's, and Xilinx Spartan Series, Xilinx Virtex-5 FPGA was simply the best choice for this study. Virtex-5 family of Xilinx FPGA devices not only represent the latest technology of the largest and most popular manufacturer of FPGAs but also provides the highest range of capacity, performance, connectivity scalability, and other features with Dynamic Partial Reconfiguration (DPR) support.

3.3 Prototyping Board Selection

To swiftly pass the hardware platform preparation phase of the project and engage in developing the system itself, a pre-assembled FPGA board should be used. Based on their new devices, FPGA manufacturers produce pre-assembled prototyping boards stacked with a diverse and flexible set of I/O interfaces and other features to meet the prototyping and evaluation needs of designers in different domains. Third party companies also produce FPGA boards for different specific applications such as ASIC prototyping, research, and development.

Among the very wide range of Virtex-5 FPGA boards, Xilinx ML505 Development Platform [405] had significantly higher specifications among very few Virtex-5 FPGA boards that meet the project budget. Figure 3.3 and 3.4 show the ML505 FPGA board and its block diagram. We can briefly measure this hardware platform in terms of the factors stated in section 3.1 as follows:

1. Cost and availability: The complete ML505 platform is available to researchers for less than £800. The software tools are also available to the research community and can also be downloaded and used for an evaluation period.

2. Popularity and prevalence: The ML505 board is built around a Virtex-5 FPGA, which is the best representative of a family of the latest, fastest, and largest FPGAs available at the time of this study. The subsidised academic price of the ML505 can contribute to the popularity of this specific FPGA board and Virtex-5 family in research community. Virtex-5 FPGAs are already popular in ASIC prototyping and high-speed communication devices and servers due to their high performance and capacity.

3. Performance (Speed): The FPGA on the ML505 platform is of -1 speed grade (lowest speed in

Table 3.2: Comparison of the latest FPGA devices from different vendors (at the time of this study). Size is represented in approximate number of Logic Elements (LEs) and I/O pins. Performance is represented by total propagation delay from LUT inputs to FF outputs (t_{ITO}) in nano secs. Full, Dynamic Partial and Lattices' Proprietary reconfiguration methods are represented by Full, DPR and TransFR [407, 410, 413, 9, 7, 11, 214].

Vendor	Device	Size (#LEs, #IO Pins)	Prop. Delay (t_{ITO} in ns)	Embedded Processing	Reconfig.	Other Features
Altera	Cyclone	3K to 20K, 104 to 301	6.56 to 8.51	Soft IPCores (Nios II)	Full	Block RAMs, Distributed RAM, Transceivers
Altera	Stratix II	16K to 180K, 366 to 1170	1.84 to 2.48	Soft IPCores (Nios II)	Full	DSP, Transceivers, Block RAM/FIFOs, Distributed RAM, Multipliers,...
Atmel	AT40K(AL)	≈500 to 3K, 128 to 384	≈10	-	DPR	Block RAMs, Distributed RAM, Multipliers
Lattice	LatticeXP	3K to 20K, 62 to 340	0.81 to 1.17	-	TransFR	Block RAMs, Distributed RAM
Xilinx	Spartan-3	1.7K to 75K, 124 to 633	1.90 to 2.29	Soft IPCores (MicroBlaze, PicoBlaze)	DPR	Block RAMs Distributed RAM, Multipliers
Xilinx	Virtex-5	30K to 330K, 172 to 1200	0.67 to 0.90	Soft IPCores (MicroBlaze, PicoBlaze) and Hard IPCores (PowerPC)	DPR	Ethernet MAC, PCI Express Endpoint, DSPs, Block RAM/FIFOs, Distributed RAM, Transceivers,...

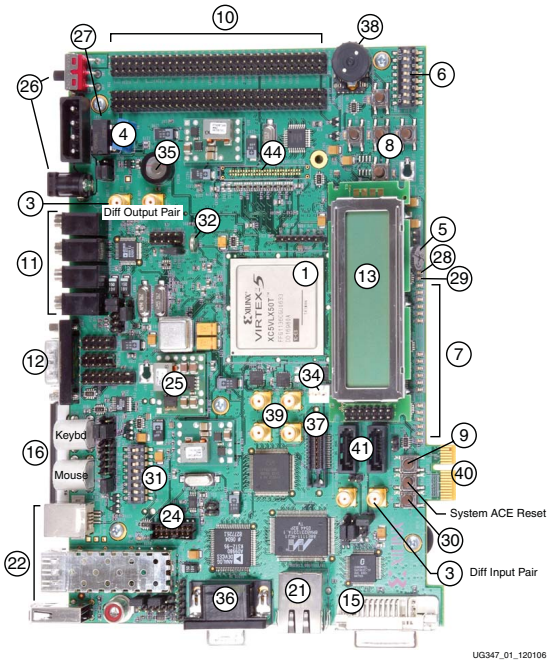


Figure 3.3: Xilinx ML505 Virtex-5 Development Platform.

Virtex-5 family) due to cost efficiency. The working speed of an FPGA depends on the component and routing line propagation delays. Virtex-5 delays are significantly better than the previous families of Xilinx FPGAs (*e.g.* Virtex-4) and other available FPGAs of this size in the market. The same chip is also commercially available in better speed grades (-2 and -3).

4. Size and scalability: The ML505 hardware platform is built around a Virtex-5 XC5VLX50TFFG1136 device with 46080 logic cells (7200 slices), 480 kbits of distributed RAM, and 60 Block RAM/FIFO (36kbits each). While this platform provides enough resources to build and evolve a small neural network (based on literature), the system is also scalable in the sense that larger devices (*e.g.* LX330T with up to 331776 logic cells) are commercially available. With high number of I/O pins and high-speed serial/parallel data transfer blocks on Virtex-5 devices it is also possible to connect few FPGAs to build larger systems.

5. Power Consumption: As the number of FPGA devices in this project is limited to one, the power consumption is not a concern. Heat dissipation of the FPGA chip on the ML505 can be carried out by an optional add-on heat-sink that is sold separately as it is not always necessary. On-chip Virtex-5 heat sensors can be also monitored using JTAG port on ML505 to make sure that heat dissipation would not be an issue during testing and experiments.

6. (Dynamic) Partial Reconfiguration: The Virtex-5 FPGA on ML505 supports both partial and dynamic reconfiguration. Two Internal Configuration Access Ports (ICAP) in the FPGA enables the device to reconfigure itself dynamically at the maximum nominal speed of 50MHz (32 bits). The new PlanAhead^{TM}design tool is promised to simplify the design process and adds to its stability. These were two major concerns in dynamic and partial reconfiguration of previous devices.

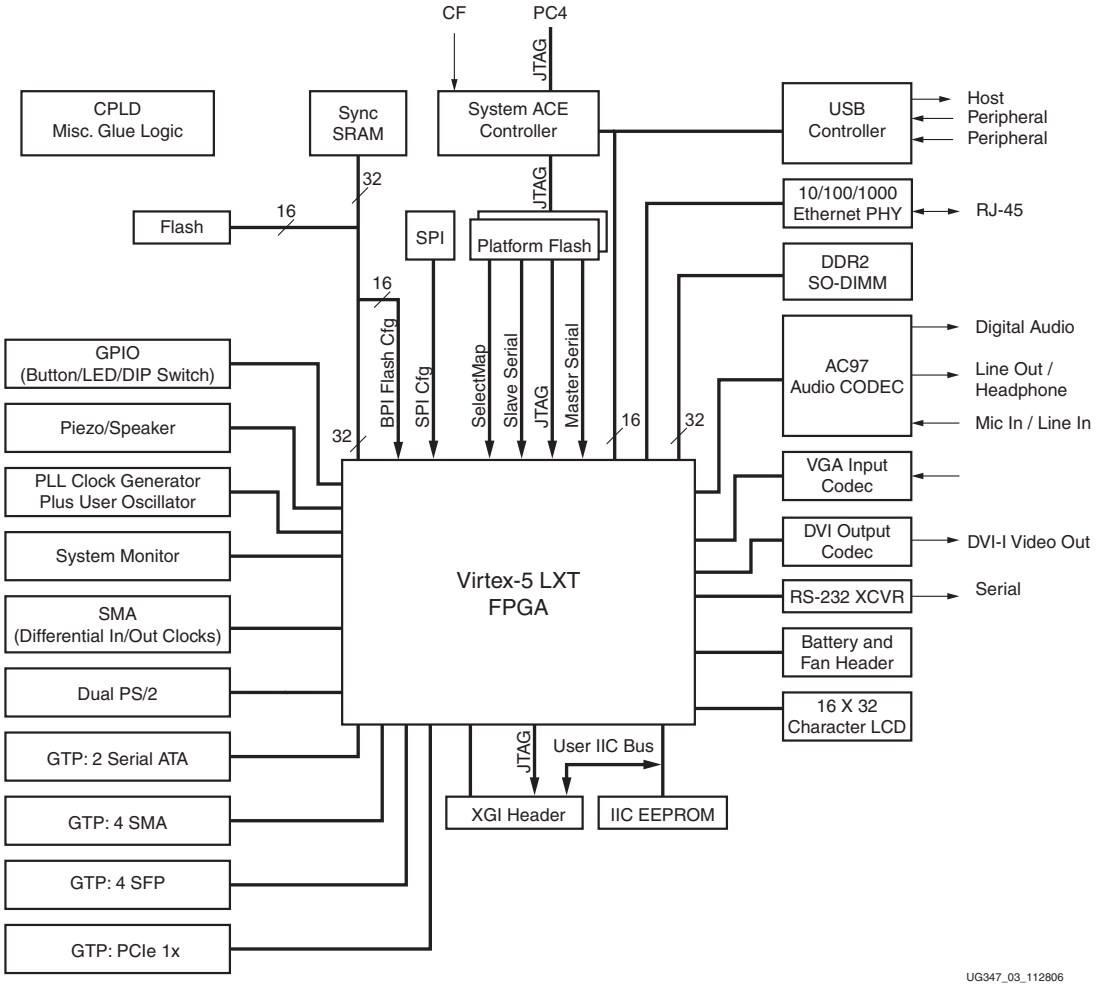


Figure 3.4: The ML505 platform block diagram.

7. Reconfiguration speed: The Virtex-5 device on ML505 supports a transfer speed up to 50Mhz with bus widths of 1,8,16, and 32 bits. The actual maximum configuration speed depends on the PC interface used for the configuration. Using JTAG interface, the Xilinx Platform Cable USB interface supports up to 24MHz 1-bit serial transfer rate. It is also possible to use the Internal Configuration Access Port (ICAP) of the FPGA in conjunction with another interface (PCI Express™, USB2.0, etc.) as a dynamic partial reconfiguration system for obtaining the maximum configuration speed.

8. Data communication bandwidth: The ML505 platform supports many high-speed serial, parallel and analogue interfaces including PCI Express™1x edge connector, USB2.0, Ethernet 10/100/1000, video I/O, audio I/O, and SATA (Serial Advanced Technology Attachment). Moreover, the on-board memory devices (256 MB DDR2 SODIMM, 1MB ZBT SRAM, 32MB Linear Flash, System ACE™CompactFlash, and a Xilinx Platform Flash) can be used for buffering or local storage of data or configuration bit streams, which can significantly reduce the data transfer overhead in some scenarios.

9. Interfacing: The ML505 platform is sporting a wide range of interfacing features (high-speed digital/analogue serial/parallel I/O, Memories, audio I/O, video I/O, Mouse, Keyboard, LCD display, etc.) that help to deploy the system in different application domains.

10. Indestructibility or validity checking: Unfortunately, new generations of FPGAs (including Virtex-5) can be damaged by incorrect configuration and Xilinx neither provided a validity check utility nor released the complete configuration bit-stream format of the device. The latest indestructible FPGA from Xilinx (XC6264), which has much lower size and performance specifications, is discontinued. Also other FPGAs with open bit-stream formats are from a few generations before Virtex-5 and provide much lower specifications.

11. Embedded Processing: Xilinx has provided two RISC soft processors cores (MicroBlaze and PicoBalze) in different configurations and sizes for implementation on FPGAs that can be used for performing I/O tasks or running the evolutionary or developmental processes on chip. They take only a small amount of FPGA resources but provide the system with lots of flexibility and programmability. Xilinx also provides C libraries for partial reconfiguration of Virtex-5 by MicroBlaze using their HWICAP IP-core.

12. Observability: It is possible to use Xilinx ChipScope Pro™utility to monitor and debug the internal behaviour of the FPGA using a PC connected to the platform by adding a special IP-core to the design and connecting it to the signals that need monitoring. However, to be able to monitor a signal in the FPGA fabric it needs to be defined at the synthesis time and a ChipScope Pro ILA (Integrated Logic Analyser) core to be included in the design.

13. Reliability: Although Virtex-5 original family of FPGA devices were soon followed by more reliable radiation-hardened (rad-hard) versions, in this study, the reliability of the original versions are quite sufficient as the FPGA will not be exposed to any harsh environments.

14. Ease of use: Xilinx provided a complete suite of GUI design tools for Virtex-5 FPGAs to simplify the design process. However, the effective use of these tools still needs a lot of experience and hard work. The previous familiarity of the author with the Xilinx devices and tools can save a lot of time in design and implementation. Although PR (Partial Reconfiguration) support in Xilinx tools was still new and immature at the time of this decision, Xilinx was more committed to developing PR tools than any other manufacturer. ML505 platform can be connected to a PC using a USB-to-JTAG adapter for initial reconfiguration of the FPGA, programming the MicroBlaze and monitoring the FPGA at run-time.

3.4 Summary of Selection Factors

Based on feasibility measures defined in section 2.2 and factors that may contribute to bio-plausibility of the system, different factors involved in the selection of the FPGA and hardware platform were identified and their interactions and trade-offs, based on previous experience and available literature, were discussed. It was identified that for the challenges and trade-offs investigated and discovered in the next chapters to be applicable and general enough, it is important to select a good representative of the latest technology in FPGA, which is also popular and available. Main factors that impact the bio-plausibility of the system appeared to be the size (capacity), scalability, performance (speed), and flexibility of the FPGA. Main factors that may affect the feasibility of the system apart from the cost of the FPGA, were simulation and reconfiguration times (impacted by size, performance, reconfiguration speed, communication bandwidth and dynamic partial reconfiguration of the FPGA). An important factor was identified to be the popularity and prevalence of an FPGA device that can drop the unit cost regardless of all its expensive features and specifications. Flexibility and ease of use (that are impacted by interfacing and embedded processing features, and indestructibility of the FPGA), and observability were other factors that appear to affect the feasibility measures. Reliability and power consumption seemed to be of less concern at this scale and in the scope of this study.

Different options of FPGA devices from different manufacturers were briefly reviewed. The significant market domination of two manufacturers (namely Xilinx and Altera) and unavailability of dynamic partial reconfiguration feature in Altera FPGAs (at the time of this study) and higher performances and larger capacities of Xilinx FPGAs made the actual selection for this study simple. A Virtex-5 FPGA was selected for this study as a good representative of a popular family of FPGAs with the latest technology, largest size and highest performance available. Although the partial reconfiguration workflow from Xilinx was still immature at the time of this study and a beta version of the tools were needed, Xilinx was the most committed manufacturer to dynamic partial reconfiguration methods. Table 3.3 presents a summary of the reasons for selection of a Virtex-5 FPGA for this study.

The limited budget of this project and subsidised academic price of ML505 prototyping board also simplified the selection of the prototyping board as there were no other FPGA board with such a high specification in this price range. The ML505 FPGA development board is built around an entry level Virtex-5 FPGA with many different interfacing options, which increase the flexibility of the hardware platform in different application domains. The ML505 development platform was purchased and Xilinx

Table 3.3: Summary of the comparison of hardware platforms for this study and their trade-offs showing Virtex-5 as the best choice.

Manufacturer	FPGA device	Size and scalability	Performance	Embedded processing	Reconfiguration	Other features
Xilinx	Virtex-5	●●●	●●●○	●	●●	●●●●
Xilinx	Spartan-3	●●○	●●○○	●	●●	●●○○
Atmel	AT40K	●●○	○○○○	○	●●	●●○○
Altera	Stratix II	●●●	●●○○	●	○○	●●●○
Altera	Cyclone	○○○	●○○○	●	○○	●○○○
Lattice	LatticeXP	●○○	●●●●	○	●○	○○○○

design tools, the required software licenses, and beta version access to PR tools were obtained.

3.5 Summary

Figure 3.5 represents a summary of the investigations carried out in this chapter in a graph. In this chapter, the factors involved in the selection of the hardware platform were analysed based on the general definitions in section 2.2, experience, and literature. A group of important factors and their major interactions and trade-offs were identified. Based on the result of the analysis, state of the market, and constraints of this project, an FPGA device and a prototyping board for this very study were selected. The features of the selected hardware platform were briefly reviewed in section 3.3. In section 3.4 the hardware platform selection factors and trade-offs were summarised. Specifying the hardware platform and the FPGA device, provides a concrete basis for investigation of challenges in the design and implementation of evo-devo neural microcircuits on FPGAs in the following chapters. The next chapter is dedicated to investigation of the challenges in design and implementation of a bio-plausible neuron model feasible on this specific FPGA.

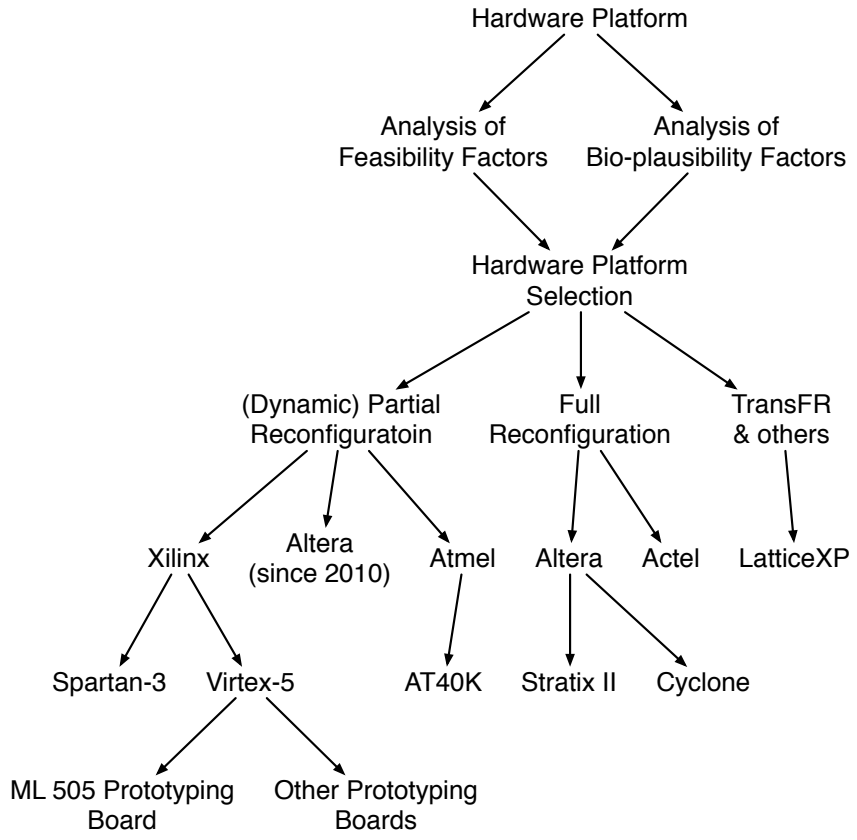


Figure 3.5: A graph of the investigations carried out in chapter 3 regarding the hardware platform.

Chapter 4

Neuron Model

Among different processes involved in an evo-devo neural microcircuit, neural processes can be regarded as the most critical, compared to evolutionary and developmental processes. This is mainly because neural activity and learning processes are executed in a much shorter time scale compared to evolutionary and developmental processes. Changes in the system due to evolutionary process occur only in every generation at reproduction time of each individual. However, for the evolutionary process to work it needs to develop and evaluate every individual microcircuit. Developmental changes take place at a higher speed than evolutionary ones, as many developmental steps might be needed to develop an individual (before or during the simulation of neural processes) to obtain the individual fitness value or any other measurement of its fitness. Similarly, neural changes such as action potentials and other synaptic dynamics are taking place at a higher rate than developmental changes. Even those developmental changes such as activity driven synapse formation, elimination, and neurite growth are regulated by neural activity of the network over a significant number of input vectors from a rather large dataset. Therefore, a bio-plausible evo-devo model of such neural system needs to dedicate a large portion of its available computational resources to such frequent changes in the neural states. This explains why the neuron model is so critical in the feasibility of the system as it has a significant impact on two important feasibility measures of application size and performance. From bio-plausibility standpoint, apart from bio-plausibility of the neuron model itself, the reliance of developmental and evolutionary processes on the neuron model, its dynamics, and its flexibility, makes it very critical in the bio-plausibility of the whole system.

This chapter discusses the challenges in design, implementation, and testing of a feasible neuron model that is not only bio-plausible by itself, but also allows bio-plausible developmental and evolutionary processes to be built upon it. First, the general factors, requirements, and design objectives of such bio-plausible and feasible neuron model are discussed, different approaches to the neuron model design are briefly explored, and design factors, constraints, and general trade-offs are identified. For further investigation of the challenges, design choices, constraints, and trade-offs in practice, and also as a basis for further investigation in following chapters, a feasible and relatively bio-plausible neuron model is designed, implemented, and tested on the constraints of this project. The process of the design, implementation, verification, and testing of the new neuron model is explained and practical challenges

of the process are summarised.

4.1 General Design Factors

Investigation of challenges in achieving bio-plausibility and feasibility are the main objectives of this study. These two factors must be translated into tangible factors in the design of the neuron model. Therefore, in this section each one of these two main factors are elaborated to identify these tangible factors and constraints. Then it will be possible to discuss the trade-offs between these factors and constraints and related design challenges.

4.1.1 Bio-plausibility Related Design Factors

Bio-plausibility of a neuron model have many different aspects: From soma model and its dynamics to dendrite and synapse models, axonal delays, development of neurites and synaptogenesis. It is important to examine each one of these aspects in the light of the current knowledge in biology and neuroscience. To limit the complexity of the model and the study, it is inevitable to ignore some biological complexities and details. Those biological complexities and bio-plausible features that are not likely to be useful in the context of evo-devo neural microcircuits can be abstracted or ignored. This can be done based on the biological evidence or experimental results in neuroscience and bio-inspired computing.

Bio-plausibility of the neuron model can be divided into bio-plausibility of the network architecture, and the neural coding, bio-plausible features of soma model, neurite model, and synapse model. Some biological background and different aspects of bio-plausibility in each of these areas are discussed in the following.

Factors Affecting the Network Architecture

Statistical analysis of the neural networks in mammals and nervous system of *C. elegans* shows that they have characteristics of so called small-world networks [39, 51, 19], meaning that while they are sparsely connected, the average number of hops between any two random nodes is very small (of order of $\mathcal{O}(\log n)$, where n is number of nodes). Preliminary studies on some cortical networks suggest that they also show behaviours of scale-free networks associated with their multi-cluster organisation [39, 51, 19].

Cortical network connectivity is very dynamic. Connections between neurons form and disappear all the time. Fault-tolerance and robustness of the cortical circuits are associated both with the network characteristics and dynamic nature of their connectivity. Evidence suggests that structural plasticity [65] and wiring delays [66] play major roles in the brain. The placement and wiring of the neurons appear to be optimised for the high interconnectivity in the brain [64] and optimising the trade-off between the physical cost of interconnections and its complexity [39, 19]. It is argued that the dynamic wiring of the cortical circuits might be also involved in the long-term memory and learning [65]. Apoptosis (controlled process of cell death) also plays a major role in the development of nervous systems and its remarkable network characteristics [404]. Although many simulations only use one excitatory and one inhibitory neuron type for simulation, biological cortical circuits are heterogeneous networks of different types of neurons with different behaviours and characteristics [168, 173]. Evolutionary and developmental

processes seem to be responsible for the emergence and maintenance of such a network architecture by controlling and regulating cell duplication, migration, differentiation, growth, and apoptosis [404].

From a bio-inspired computing standpoint, there is no systematic method for specifying the size and architecture of recurrent neural networks in general or for a given problem or class of problems [175]. Few studies provided some evidence that hierarchical, modular, and structured topologies can significantly increase the performance of the network in the context of reservoir computing [332, 307, 308]. This shows the necessity of an evolutionary process to control the architecture of the network in a bio-plausible design. While the exact network connectivity in the cortex appears to be controlled by stochastic processes and structural plasticity, the modular, structured and hierarchical nature of the cortical circuits points to the necessity of a developmental process [39, 19, 228, 351].

It is clear that in a useful bio-plausible model, the network architecture needs to be malleable, evolvable, and adaptable, and at the same time controlled by a developmental evolutionary process that is constrained by physical limitations of the hardware platform. Evolutionary and developmental processes need to be able to control the duplication, migration, differentiation and death of the neuron cells. These processes also need to be able to add or remove connections (synapses) between neurons.

Factors Affecting the Neural Coding

Neural coding of the brain is still a subject of study and debate. Many scientists and modellers believe that the spike rates are sufficient and there is no need for precise timing for information processing. While it is true that sigmoidal neurons, which only model spike rates, can be effectively used in recurrent neural networks, it can be shown that different neural coding schemes based on the precise timing of the spikes can appear as rate codes [239]. However, evidence suggests that in many parts of the brain neurons use precise timing of the spike to convey information and fast and reliable transmission of spikes is important in the cortical circuits [237, 239, 42, 43, 91]. In [91] authors argue that using spikes and their timing is not only bio-plausible, but also more efficient. Moreover, bio-plausible models of Hebbian learning such as STDP (Spike-Time Dependent Plasticity) [216, 344, 43] and even at least one interpretation of supervised learning [172] are sensitive to the precise timing of the spikes.

Although there is no need to decide about the exact coding scheme(s) of the brain, it is clear that naive rate codes with a long counting window cannot support many fast and real-time applications [239]. At the same time, neurons in some parts of the brain seem to be insensitive to the exact timing of the spikes [43].

Different rate codes based on spike density, average number of spikes over a time window, and average over a population, and also time-specific codes, based on time-to-first-spike, phase, correlation, and synchrony are suggested for modelling the cortical circuits [239]. It appears that all of these coding schemes can be plausible in specific situations and brain may use any or a mixture of them when applicable. However, many bio-inspired systems are limited to one or a limited set of these coding methods on the basis of their specific application or for sake of simplicity and feasibility. These models are usually justified based on the assumption of bounded network activity (*e.g.* event-based simulation of spiking neural networks [331, 229]).

A useful bio-plausible model in our context needs to be flexible enough to use whatever coding scheme that is suitable for the application. It might prove useful for the system to be able to use different coding methods in different parts of the network. For example, in a hierarchical network, higher levels might work on a slower time scale using a spike density code, while the lower levels that deal with stimulus and response might use the precise timing of the spikes. It is therefore important that evolutionary and developmental processes be free to explore the solutions for the appropriate coding or mixture of coding methods depending on the given problem while constrained by the physical limits of the hardware platform such as spike resolution, wiring delays, processing speed. etc. Such flexibility requires that we do not pose any additional a priori limitations on the spike coding at the design time of the neuron model.

Factors Affecting the Soma model

When it comes to bio-plausibility of a neuron model, many studies focus on the soma, its dynamics and its variants. As discussed in chapter 2, the most bio-plausible models of the neuron are multi-compartmental models that track the dynamics of the membrane potential, ion channels, and ion densities with respect to time and space. Obviously, that level of detail would be impractical for an evo-devo neuron model on FPGA. It is evident that abstracting all those behaviours even as delay lines can give a Liquid State Machine universal computing power [240]. This is probably the justification behind very simplistic and degenerate LIF (Leaky Integrate-and-Fire) models used in [374, 105, 311]. Even traditional LIF neuron models are incapable of showing many important behaviours of regular spiking neurons and many other types of neurons in the cortex [169]. To be able to create a bio-plausible heterogeneous network of spiking neurons, a parametric model such as Hodgkin-Huxley [158], Izhikevich [168], or SRM is needed. As mentioned before, a parametric flexibility that can be exploited by evolutionary and developmental processes can lead to a more bio-plausible solutions than homogeneous networks of a fixed neuron type. A continuous parametric space leading to continuous changes in the phenotype can significantly contribute to the evolvability and adaptability of the system as well [165, 343, 387].

Bio-plausible neuron models also take into account the slow high-threshold dynamics of Na and K conductances using a 2-dimensional system of equations that can give neuron model capabilities and behaviours such as bursting, phasic spiking, rebound responses, threshold variability, bi-stability of attractors, and autonomous chaotic dynamics [169].

Factors Affecting the Neurite Model

Neurites are the projections of dendrites and axons out of the soma. These processes that are extended in space between different soma cells contribute to many different properties of the neural systems. Axons and dendrites have a relatively very low signal transmission speed compared to electronic signals [73, 239]. This introduces signal delays depending on the length and thickness of these neurites [73, 239]. Also there are nonlinear dynamics involved in the integration process of distal dendrite branches on many types of neurons [149]. These nonlinearities depend on many different factors including the dendrite morphologies and spatiotemporal patterns in the input spikes. They provide higher computational power to the neurons and affect synaptic plasticity and neurons ability to detect synchrony in the input signals

[91, 149]. Multi-compartmental neuron models tend to model all these dynamics in time and space while many other simpler models abstract them into axonal delays or simplified nonlinearities in the dendrites [149].

The dynamic connectivity and malleability of the cortical circuits is based on the developmental processes in the neurites. Axons and dendrites are able to grow, recoil, or die, guided by developmental processes, neuronal identity, network activities, and environmental cues, to form new connections with other neurites or eliminate redundant connections [404]. These developmental processes behind the growth and retraction of the neurites and their connections must be flexible enough to be smoothly tuned by the evolutionary process. A large portion of the learning process is based on formation and elimination of these connections rather than only changes in the synaptic weights [65, 340, 200, 264]. Although the exact algorithm and mechanism of this type of learning is not clear, through phenomena such as ocular dominance, it is evident that the principle of "fire together, wire together" is essential in formation and maintenance of the cortical circuits [404, 187]. This points to local Hebbian learning processes that use local available information from both post and pre-synaptic neurons for synaptic formation, elimination and weight changes [91].

Factors Affecting the Synapse Model

Synapses are very important part of cortical circuits. Not only they are formed and removed by the development processes and their strength is adjusted by Spike-Timing-Dependent Plasticity (STDP) and long-term potentiation/depression (LTP/LTD), but also they have shorter-term temporal dynamics such as short-term synaptic enhancements and Short-Term Depression (STD) at different time scales [248]. It is also shown that even different synapses on the same axonal tree can show unique dynamical behaviours [248]. Although the exact algorithms for all these dynamic changes are not discovered yet, many algorithms and mechanism are proposed to model these dynamics and their interactions with each other and with developmental processes of the neurites [248]. Many of these models are based on local information that can be accessed by the synapse from pre and post-synaptic neuron cells. However, extracellular chemicals such as Dopamine signals are also involved in the synaptic plasticity and learning process[172]. Chemical factors controlling these dynamics are regulated by the network activity, reward signals, and developmental and evolutionary processes. Therefore, it is clear that all the factors involved in adaptation and learning in the synapses are flexible and must be smoothly controlled by evolution. Baldwin effect and interactions between learning, development, and evolutions can also facilitate the evolvability of neural microcircuits [351, 76, 157].

4.1.2 Feasibility Related Design Factors

Different aspects of feasibility of the neuron model can be also discussed based on some of the feasibility measures defined in section 2.2 and constraints on these measures. Here, each of these measures and their relation to the neuron model design factors are discussed.

1. Hardware cost: the cost of the hardware and mainly FPGA chip is proportional to the logic and routing resources and the performance of the FPGA device and its interfacing. These are related

to the speed efficiency and compactness of the neuron model in terms of the amount of logic and routing resources it needs to implement a neural microcircuit and the speed of execution. In other words, neural model compactness and efficiency can reduce the hardware cost.

2. Design and testing time and complexity: The time needed for the design, verification and testing of the model in hardware is related to its complexity.

3. Performance: The time that is needed for experiments is tightly coupled to the simulation, learning, development and evolution speed of the neuron model. Each solution must be evaluated in the shortest possible time to maximise the evolution speed. Moreover, parallelism allows dedication of more hardware resources to increase the performance.

4. Reliability: Reliability of the whole system depends on the reliability of the neuron model. While an unstable and unpredictable model can lead to unreliability of the whole system, a robust and fault-tolerant model can provide the basis for a reliable, robust and fault-tolerant system. Reliability and accuracy of the neuron model must be comparable with the biological neurons as well. Redundancy, distributed processing, and parallelism are also related to the fault-tolerance and robustness of the system.

5. Application size and scalability: It is important that the neuron model can be scaled up to higher number of neurons and connections. Moreover, compactness of the model leads to larger applications on the same chip. Again parallelism allows us to scale up the system without sacrificing the performance.

6. The complexity of the design and testing of the neuron model must be managed by a modular design and observability of the signals in the model for debugging and testing purposes.

4.2 General Design Options

The tangible bio-plausibility and feasibility related factors involved in the design, implementation, and testing of the neuron model are summarised in Table 4.1. Based on these general factors it is possible now to investigate different general design options and approaches. Each neural engineering system needs to deal with three aspect of computation namely, processing, data storage and communication. As reviewed in section 2.4.7, different design choices for these aspects will lead to totally different designs with different properties. For examples some designs [111, 112] use Address Event Representation (AER), Network-On-Chip (NOC) and inter-chip packet-switched networks for intra-cellular communication of spikes (axons). Many others use central or distributed, dedicated or shared random access memories for storage of the spikes, soma and synapse state variables and parameters. Processing elements can be also centralised or distributed, dedicated to each neuron and synapse or shared between all or a group of neurons and synapses. Inter-cellular communication and processing (in dendrites and soma) can be also performed using stochastic or deterministic computing, parallel or serial processing, parallel or serial arithmetic and any combination of them.

Table 4.1: A summary of the tangible bio-plausibility and feasibility related factors involved in the design of the neuron model.

Bio-plausibility Related Factors
Flexibility and evolvability of the neurite model that allows growth, retraction, synaptogenesis, and synapse elimination
Continuous parametric flexibility and evolvability of the soma model
Continuous parametric flexibility and evolvability of the synapse model and learning rules
Heterogeneity of the network in terms of synapse and soma types
Locality of the developmental and learning processes
Possibility of global learning processes
Time-accuracy of the spike signals
Flexibility of the neural coding
Temporal dynamics of the synapse and neurites (delays, short-term plasticity, nonlinearities)
Fine-grain parallelism, distributed processing, redundancy, and modularity

Feasibility Related Factors
Performance (simulation speed) and using parallelism
Compactness (hardware resources)
Efficiency (speed to hardware resources ratio)
Scalability and application size
Reliability and accuracy (comparable with biological neuron)
Fault-tolerance
Robustness
Manageable complexity, modular design and signal observability

Here we explore the feasibility of taking a bio-plausible approach regarding these design options, meaning that, inspired by the biological structure of the brains, we examine if a structurally more accurate design of the system is feasible and useful towards bio-plausibility of the model. In biological neural microcircuits almost all of the processes, data storage and communications are distributed, parallel, local, stochastic and asynchronous. Long-range communication paths usually consist of many local processing, storage, and communication elements that concertedly work together. In biological brains different aspects of communication, data storage, and processing are blended at a very fine atomic scale and cannot be isolated as we are used to in engineering. Towards that goal, we can analyse the basic structure of neurons and cortical circuits to identify these local storage, communication and processing units and investigate the feasibility of a design on that basis.

Starting with the soma cell as a unit separated by its membrane from the surrounding tissue and other cells, the data storage is performed by the differential density of the ions, neurotransmitters, proteins, and other molecules across this membrane. Moreover, the local density of these chemicals along the dendritic tree creates many local storage units. All these storage units of data interact with each other and with the membrane that is covered with different types of voltage-sensitive ion channels and receptors working as processing elements. The dendritic tree can be regarded as a 2-way communication channel of local processing elements that are relatively isolated from the environment and can grow in different directions and form synaptic connections with axons of the other neurons. Similarly, axons can be modelled as one-way communication channels consisting of many simple processing elements that mainly transfer spikes to all the axonal branches with delays proportional to the distances. Each synapse has access to the local protein, neurotransmitter and ion densities, and membrane potential, and the learning process is affected by these factors. The growth or retraction of these dendrite and axon branches and formation and elimination of the synapses between them are also controlled by local developmental processes affected by the proteins diffused by processes inside the neurons or at a further distance.

This analysis models the dendrites as bidirectional data channels full of dendritic processing elements and synapses, and axons as unidirectional spike transmission lines comprised of elements that only introduce delays. Soma units are also a body of many locally interacting processing elements that interact with the dendrite and axon bases. Such a model suggests an abstracted general architecture of figure 4.1 as an aggregation of processing elements, synapses, and delay elements. Although highly abstracted, such a model still features many properties of the biological neurons such as distributed processing, redundancy, fault-tolerance, locality, fine-grain parallelism, and modularity.

Another feature of the biological neurons is the stochastic nature of their processing elements. For instance, ion channels stochastically switch between open and close states at a high speed and the collective effect of many such channels switching in response to voltage or neurotransmitters appears as a continuous change in the membrane voltage [73]. It then makes sense to use many simple stochastic digital processing elements that can collectively imitate the behaviour of a biological neuron. As there are different types of ion channels on the neuron membrane with different properties and sensitivities,

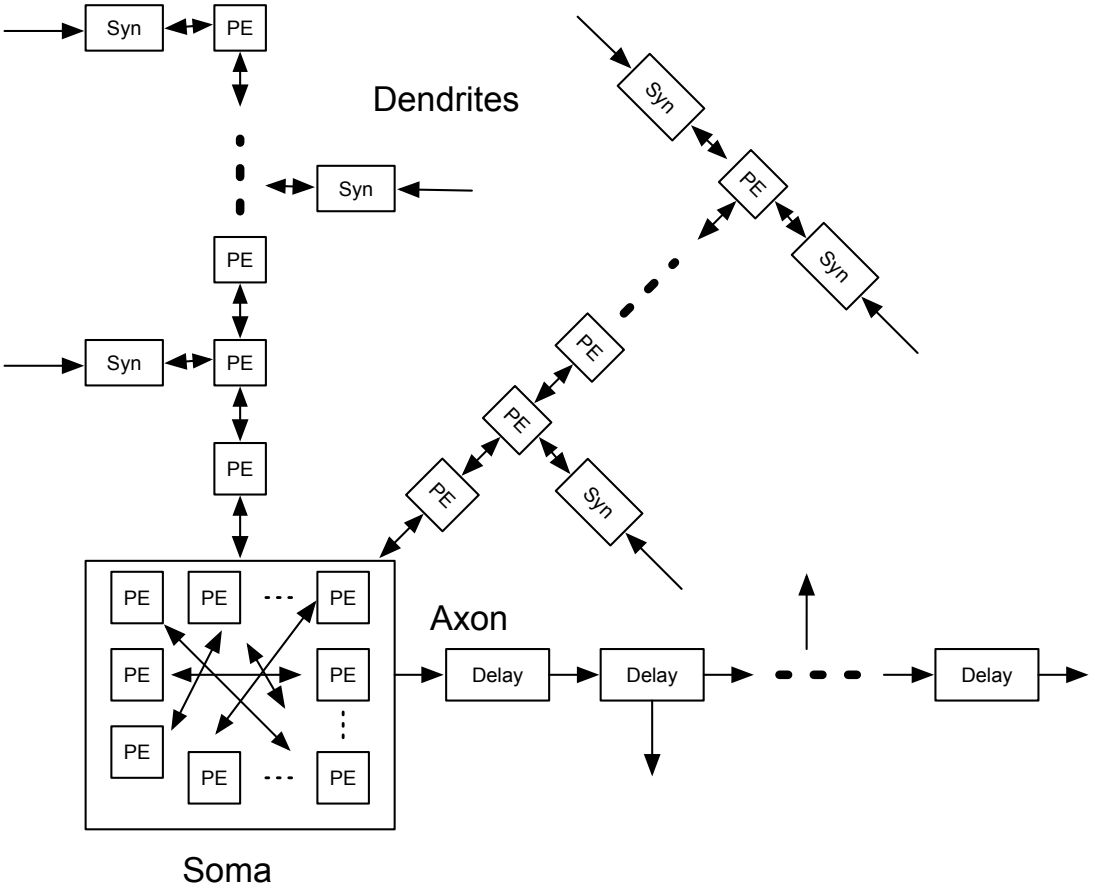


Figure 4.1: Abstracted general architecture of a neuron as an aggregation of processing elements (PE), synapses (Syn), and delay elements (Delay).

it would be also possible to have different types of processing elements that are sensitive to different signals. However, in many cases the added value of such bio-plausible design may not justify the cost of that many processing elements. For example, in the axon, each segment consisting of many modules that mainly work as a delaying transfer line for almost identical action potentials. Each of these segments can be efficiently modelled with a single digital delay element that transfers digital pulses instead of many stochastic processing elements collectively propagating an analog ripple.

Based on a similar bio-plausible approach, a single wire can be used for extracellular spike transmission of every axon instead of using central shared memories or buses of traditionally-engineered designs. Such a one-to-one mapping between implementation and network architecture, which dedicates a separate routing resource to each axon, with sufficient time resolution, can support any neural coding that evolution needs for tackling a given problem. This is in contrast with the event-driven simulators and corresponding Address Event Representations (AER) used in buses, memories, and packet-switched networks that may face too many event collisions caused by synchrony in the signals or simply high network activities. Such event-driven methods need to assume a bound on the network activity or compromise the reliability and accuracy of the signal transmission. It is important to notice that such inaccuracies can destruct the synchrony and information content of temporal-coded signals exactly when they are needed

the most. Therefore in the first step such event-driven methods are avoided in order to keep the neural coding free of any biologically implausible limits that can impact the neural network performance and its evolvability.

Based on the above analysis, the first investigated design approach is to use simple stochastic digital processing elements to generate the behaviour similar to those of soma, dendrite, and synapses and use digital delay elements and dedicated routing resources for axons all organised in the general architecture of figure 4.1. This approach implies that we use a separate set of processing elements for each synapse, each dendrite segment, and the soma. The total hardware resources used by these processing elements and the communication lines between them is a major constraint in the design that needs special attention. Routing resources in FPGA are usually the bottleneck of such connectionist designs. Therefore, it would make sense to explore designs that require minimum amount of these routing resources. Although, using the same bio-plausible principle of locality in the design of the neuron model can help to reduce the number of long-range routing resources, the number of local connections also play a major role. As discussed in the next section, using stochastic bitstreams can reduce the number of these connections to a bare minimum.

4.3 Stochastic Models

As reviewed in section 2.4.8, stochastic computing can be used effectively in neural computing. A plethora of studies and methods suggest slightly different representations and arithmetic elements [116, 50] including single and double-wire, bipolar, and unipolar, and unbounded representations. The feasibility factor of compactness of the neuron model can justify using single-wire representations. Moreover, as the range of membrane potential values in biological neurons is bounded, using one of the bounded representations makes sense. Therefore, we can focus our investigation on the bipolar and unipolar representations (see section 2.4.8). Although any system of equations based on one of these representations can be transformed into the other one through change of variables, the hardware resources needed for arithmetic operations in these representations are quite different. For example, a multiplication that can be performed in the unipolar representation using a single AND gate, needs at least three gates to be done for bipolar representation. Therefore, depending on the type of operations, a designer might like to switch between these representations or even use a mixture of them. In the first design exploration step, we focus on a distributed stochastic neuron model design that uses bitstreams for inter-cellular communication. Then, we investigate the effect of reducing the number of processing elements on the feasibility and bio-plausibility factors.

4.3.1 Distributed Stochastic Models

In a distributed neuron model, the processing is distributed over all the processing elements in the neuron. Designing each processing element needs to follow the behaviour of the biological counterparts at a micro level. For example, processing elements in the dendrite need to act like ion channels on the cell membrane. This is obviously not a feasible approach at the current time as the current technology cannot provide the number of processing elements needed for such one-to-one mapping of the structural units.

We are bound to scale down the number of processing elements.

When the number of processing elements is reduced, it is not clear what behaviour to expect from each element. We have to reverse engineer the behaviour of each element from the known behaviour of the biological neuron in a way that when they work together they show the same collective behaviour. Bio-plausible parametric models of the neurons such as Hodgkin-Huxley [158] and Izhikevich [168] are presented as a system of differential equations. The challenge is to design processing elements to capture these differential equations.

Although originated from another context, studies in the distributed protocols [135, 274, 136] provide us with an effective tool for translating these equations into stochastic Finite State Machines (FSM) that communicate with each other and generate the desired collective behaviour.

Following the guidelines from [135, 274, 136], and based on the general architecture of the neuron model in figure 4.1 we can explore the design challenges of a neuron model. Bio-plausible neuron models use at least a two-dimensional system of equations that govern the behaviour of the rapid and slow dynamics in the neuron. The rapid dynamics are governed by a quadratic differential equation while the slow dynamics are captured by a first degree differential equation (see equations 2.10). At this stage we can focus on a simpler one-dimensional differential equation (based on a quadratic neuron model - equation 2.9) to investigate the design challenges. Considering a quadratic differential equation of the general form:

$$\dot{u} = a(u - v_r)(u - v_t) \tag{4.1}$$

where u denotes the membrane potential, a is a constant capturing the membrane decay constant, v_r and v_t are constant resting and threshold voltages of the neuron respectively. In [135, 274, 136] Gupta and Nagda presented a procedural method that starts with a set of differential equations and arrives at a set of condition-action rules for a distributed system of interconnected nodes that behave according to those differential equations. Following the guidelines in [135, 274, 136] yields processing elements with a binary state variable ($S = \{0, 1\}$) and following actions:

1. An element in state 1 performs the following with probability $P_A = p \cdot a$ (p is a scaling factor): it randomly samples another processing element from the system and if it is also in state 1 it senda a token to a randomly selected processing element.

2. An element in state 1 will go to state 0 with probability $P_B = p \cdot a(v_r + v_t - v_r v_t)$.

3. An element in state 0 goes to state 1 with probability $P_C = p \cdot a \cdot v_r \cdot v_t$.

4. An element in state 0 that receives a token goes to state 1.

5. An element in state 1 that receives a token stays in state 1 and resends the token to another randomly selected process.

The above distributed protocol assumes a network that allows communication of each element with any other randomly selected processing element. To emulate such communication network in the gen-

eral architecture of figure 4.1 we can randomly shuffle and move around the state and token bits using communication lines between processing elements. Moreover, we can add a device (here, called a scrambler) that is able to change the order of the stochastic bits. As all the processing element have identical structure and functionality, only moving the state bits around is sufficient for random communication of elements and tokens and there is no need for moving the tokens around. Figure 4.2 shows a general design using a state ring for moving the state bits around. In this design each processing element can send/receive state bits to/from upper and lower neighbouring elements. Each unit also has a token flip-flop that stores the token generated by the processing element. Each of such processing elements can be constructed using a few flip-flops and Lookup Tables (LUTs) on the FPGA. The scrambler sits in the middle of the state ring, receives random numbers from a Random Number Generator (RNG) and shuffles the state bits. This scrambler can be also constructed efficiently with a shift register present in the FPGA. About half of the Virtex-5 LUTs can be used as random addressable 32 bit shift registers. The logic circuit inside each processing elements captures the logic of the above actions on the values of the token and two state flip-flops. This logic circuit needs stochastic bitstreams with probabilities of P_A, P_B, P_C and another stochastic bitstream P_h with probability equal to half. This last stochastic bitstream is needed when logic circuit has two different equally probable options. For example, when both state bits that are received from neighbouring element are 0, and token bit is 1, action 4 can be performed on any of the state bits. In this case P_h is used to decide which one to change. Note that each PE unit actually contains two processing elements (two state bits) that share the logic and token bit. The logic equations for the logic unit can be derived from the above actions as:

$$S_u = TS_uS_d + P_C\bar{T}\bar{S}_d + T\bar{S}_d\bar{S}_u(\bar{P}_hP_C + P_hP_B) + P_B(T\bar{S}_dS_u + S_d\bar{S}_u + \bar{T}S_d) \quad (4.2)$$

$$S_d = TS_uS_d + P_C\bar{T}\bar{S}_u + T\bar{S}_d\bar{S}_u(\ P_hP_B + P_hP_C) + P_B(TS_d\bar{S}_u + \bar{S}_dS_u + \bar{T}S_u) \quad (4.3)$$

$$T = S_uS_d(T + P_A) \quad (4.4)$$

where S_u and S_d on the right hand side of the equations are inputs from up and down neighbouring processing element.

Although this model does not exactly work as the distributed protocol, it is a good estimation. The only differences are that the random samplings may not be always as random as distributed protocol, and in very rare cases (much less than 2% of the time if fitted with Izhikevich parameters) a token will be lost when all of the S_u, S_d, T, and P_A in a PE are 1. While this simple model lacks the mechanism for detection of an action potential, spike generation, and resetting the membrane potential, and does not include the slow dynamics of the biological neurons, it is useful for investigation of the challenges in using this model in a design.

Since accurate in-vivo measurement of the intrinsic neuronal noise is not possible [355], for comparison with biological neuron, we resort to using estimates from [246]. The total peak-to-peak voltage amplitude (σ_V) of the intrinsic neuronal noise of a pyramidal soma at resting potential (including thermal noise, K^+ and Na^+ ion channel noises, and synaptic noise) is estimated to be about 1mV [246]. Based on this estimation, and a signal range of -100 to 30mV, the Signal-to-Noise Ratio (SNR) of the biological

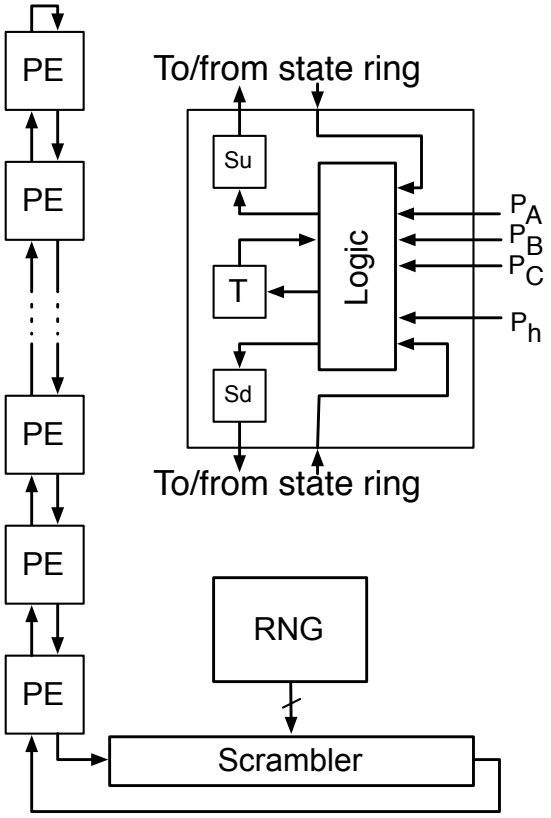


Figure 4.2: Left: general design of a distributed stochastic model consisting of a random number generator (RNG), a scrambler, and a number of processing elements (PE). Right: internal structure of a processing element (PE). Su, Sd, and T are the State up, State down, and token flip flops respectively.

pyramid neuron can be calculated:

$$SNR_{dB(Pyr)} = 20\log_{10}(\frac{A_{signal}}{A_{noise}}) \quad (4.5)$$

$$= 20\log_{10}(\frac{130mV}{1mV}) \quad (4.6)$$

$$\approx 42dB. \quad (4.7)$$

To investigate the effect of the number of PEs and length of the scrambler on the SNR of the system, the above stochastic distributed model of the quadratic neuron was simulated at the resting potential for 5000 time steps. Preliminary simulations with a scrambler length of 32 bits showed that over 2000 PEs (a total of 4032 state bits and 2000 token bits) are needed to achieve a SNR equal to the above estimate of 42dB. Moreover, to detect and generate an action potential, the bitstream must be examined with a large binary counter for a long time. This will also add to hardware resources and latency of the model. Obviously, such a high number of PEs is outside of the feasibility margins for compactness, performance, and efficiency of the neuron model on an FPGA device. However, pursuing this method might prove useful for small scale systems with very few neurons due to the bio-plausible stochastic nature of the model. Next, non-distributed stochastic models are examined.

4.3.2 Centralised Stochastic Models

In a centralised stochastic neuron model based on the general architecture of figure 4.1, synapses and dendrite structure are only responsible for input current integration and all the non-linear dynamics of the neuron is centralised in the soma unit. In such a model, each synapse unit can add some pulses to a bitstream according to its synaptic weight. The soma unit then generates the updated membrane potential signal in response to the arriving stochastic pulses and sends a copy conveying the updated potential to all the synapse units. Therefore, soma unit should contain all the parametric non-linear stochastic operators that can be evolved to change the behaviour of the neuron.

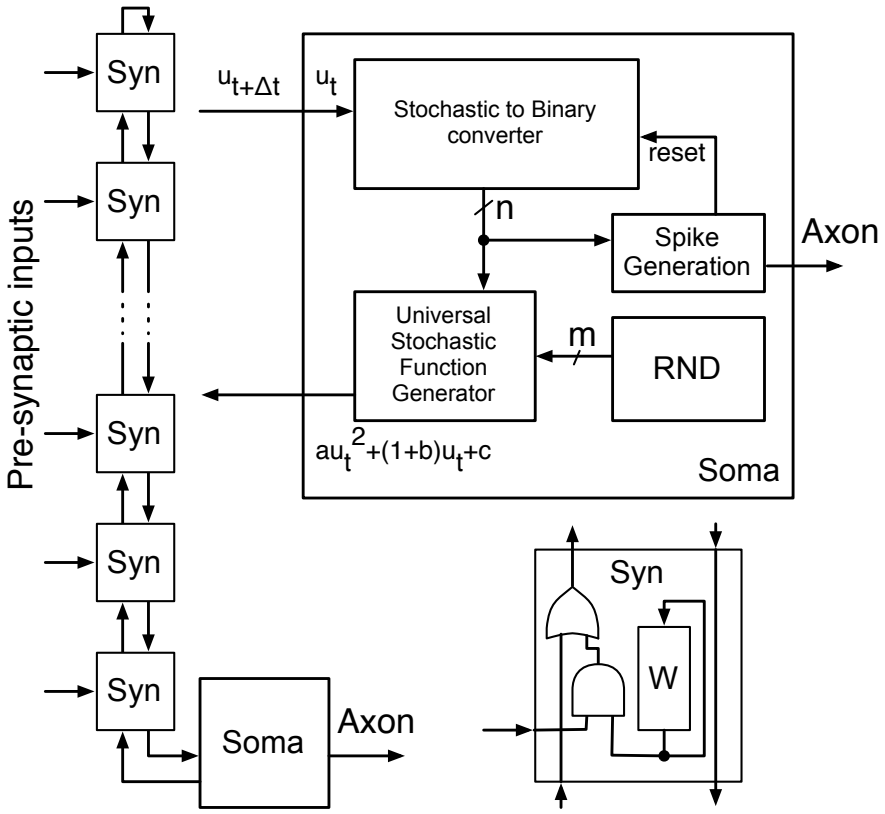


Figure 4.3: Left: General design of a centralised stochastic model. Right: Internal structure of the soma and synapse units. RND represents a source of random bitstreams with specific probabilities.

To investigate the challenges of centralised stochastic models, here again, a one-dimensional quadratic neuron is designed using stochastic computing techniques. Figure 4.3 shows the general design of a centralised stochastic model with a dendritic ring that conveys the stochastic bitstreams (with probability $au_t^2 + (1 + b)u_t + c$) to synapse units. Every time that a synapse unit (bottom right of figure 4.3) receives a presynaptic input spike it adds the synaptic weight (a short bitstream with probability of w stored in a shift register or generated by a digital to stochastic converter) to this upward stochastic bitstream and passes it to the next synapse unit. The soma unit (top right in figure 4.3) receives the total membrane potential including postsynaptic currents ($u_{t+\Delta} = au_t^2 + (1 + b)u_t + c + I$) and calculates the new upward bitstream. The soma unit is designed based on a stochastic-to-digital converter [50], a universal stochastic function generator, a spike generation module and a source of random bitstreams

with constant probabilities (RND in figure 4.3).

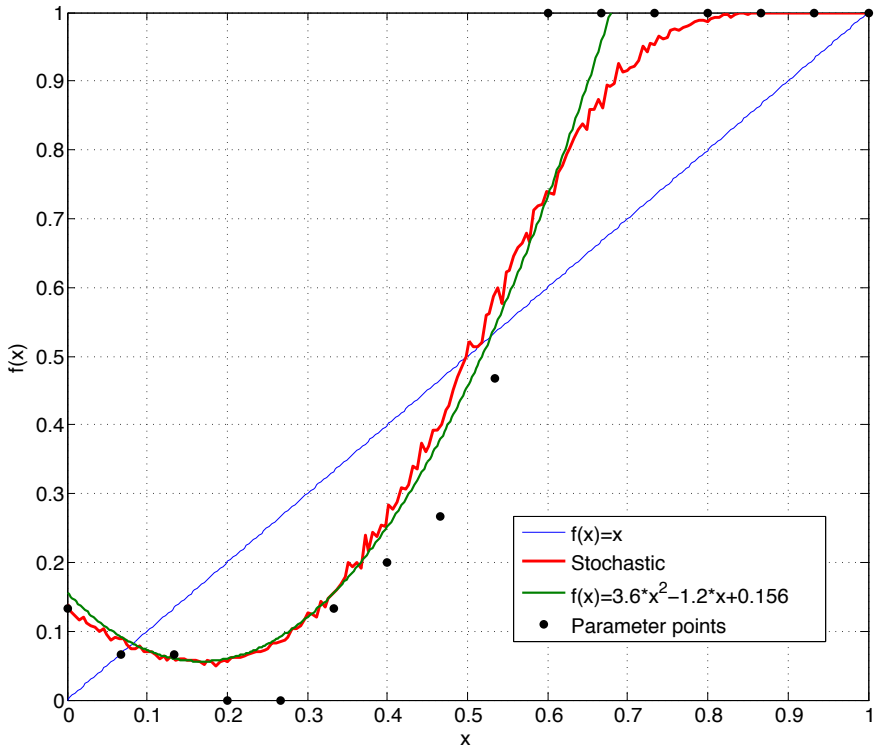


Figure 4.4: Simulation results of a 16x4-bit stochastic function generator with parameters tuned to approximate the function $\mathbf{f(x) = 3.6x^2 - 1.2x + 0.156}$ (shown in green). The red curve shows the probability of the function generator output bitstream against the input value (x). Intersection points of the blue line ($\mathbf{f(x) = x}$) and the update function determine the resting ($\mathbf{x = 0.1}$) and threshold ($\mathbf{x = 0.5}$) potentials respectively.

The stochastic to digital converter is mainly an up/down n-bit saturating binary counter that counts the ones up and the zeros down in the bitstream and estimates the probability of 1s in the bitstream as a n-bit binary number. This is the binary representation of the current membrane potential. After an action potential that is detected by spike generation module the counter will be reset to a pre-specified value (reset potential) and a spike pulse will be sent to the output. The universal stochastic function generator is essentially a lookup table of 2^n values of 2^m-bit binary number. It uses the same principle of digital to stochastic converters presented in [50] and used in [391, 14]. However, in the function generator each n-bit value of the membrane potential will address a register in the lookup table that contain the corresponding value of the function. The other address lines of the lookup table are connected to m random bitstreams with relative probabilities proportional to the weights of the binary digits in the register so that the total probability of these m bitstreams add up to one. It is therefore possible to tune the parameters in the lookup table to generate any arbitrary function of the input. Figure 4.4 shows the simulation results of an example stochastic function generator with a 16x4-bit lookup table with parameters tuned to approximate a quadratic function that can be used in a quadratic neuron model. The red curve shows the probability of the function generator output bitstream versus the input binary value. The green curve is the reference quadratic function. Intersection points of the blue line with

these curves about $x = 0.1$ and $x = 0.5$ determine the resting and threshold potentials respectively. The probability of the bitstream is calculated by counting 10,000 bits. The simulation demonstrates that very close approximations of such functions can be achieved by tuning a set of parameters. Parameters loosely work as control points that pull the function curve up and down at different values of x.

Although this is a very simple and general model it allows many different extensions and improvements. Each synapse unit has access to both the presynaptic spike and the current partial and total membrane potential (as upward and downward stochastic bitstreams) to be used for adding synaptic plasticity mechanism for learning, short-term plasticity, etc. It is also possible to add a second variable and equation for the slow dynamics of the soma.

This simple model can be implemented using Virtex-5 LUTs, shift registers, and flip-flops. The n-bit converter can be implemented using n flip-flops, a shift register (1 6LUT), and $n/2$ 6LUTs virtex-5 primitives. The function generator can be implemented with 2^{m+n-6} 6LUTs configured together as a 2^{m+n}-bit RAM. The random bitstreams with constant probabilities can be generated globally and used in all modules. The total hardware needed for the soma unit can be estimated to need $2^{m+n-6} + n/2$ 6LUTs and $n + 1$ flip-flops.

To calculate the noise in the system we start with the quantisation error of the binary representation in the stochastic to digital converter. The SNR (Signal-to-Noise Ratio) of a n-bit binary representation is equal to:

$$SNR_{dB} = 20 \log_{10} (2^n) \approx 6.02 \cdot n \;\; dB \tag{4.8}$$

Therefore, for achieving the 42dB SNR of the biological pyramidal neuron, such a centralised stochastic neuron model needs at least $n = 7$ bits. Even with a minimum of $m = 2$, necessary to achieve the needed accuracy for the function generator, a soma unit at least needs 12.5 6LUT primitives and eight flip-flops. That is roughly equal to four Virtex-5 slices. It would be also possible to compactly implement a synapse units and a very simple flexible learning mechanism in a minimum of half a Virtex-5 SLICEM.

Another source of noise in such a system is the inherent statistical variation in counting stochastic bitstreams [50]. The coefficient of variation of a Bernoulli bitstream with probability p is:

$$CV = \sigma/\mu = \sqrt{\frac{1-p}{np}} \tag{4.9}$$

where n is the number of counted bits [50]. Then, SNR can be calculated as:

$$SNR_{dB} = 20 \log_{10}(1/CV) = 20 \log_{10}(\mu/\sigma) = 10 \log_{10}(\frac{np}{1-p}) \;\; dB \tag{4.10}$$

This value depends on the probability of the bitstream as well. Counting $n = 128$ bits that is equal to the capacity of the 7-bit counter of the stochastic to digital converter gives a maximum of about 21dB for $p = 0.5$ and 11.5dB for $p = 0.1$. This is a tighter bottleneck on the accuracy of the system than the noise due to the digital quantisation. This means to achieve 42dB SNR of the biological neuron, even only at $p = 0.5$, the soma needs to count over 16000 (2^{14}) bits. This was also confirmed through simulations

that showed for an acceptable noise level in signal estimation, well over 8000 stochastic bits must be counted.

This affects both the speed and hardware efficiency of the model since counting 2^{14} bits needs at least 2^{14} clock cycles and a 14-bit counter. To mitigate the impact of increasing accuracy on the hardware efficiency, it is possible to increase only the number of bits in the counter and use the higher order bits for the same small function generator. However, even accepting a higher noise level cannot significantly improve the speed as noise level is of the order of $\mathcal{O}(1/\sqrt{n})$ where n is length of the counted bitstream.

Stochastic models have the advantage of hardware efficiency at the cost of lower speed, thus possibility of realising a large-scale network on FPGA. They are slower than their deterministic counterparts. However, very simple and low latency circuits allow them to achieve higher clock frequencies that may slightly mitigate this problem. Stochastic models are more immune to noise and when they achieve the required accuracy, they are more robust than their deterministic counterparts. Their reliance on single wire signals can minimise the FPGA routing resources needed for intracellular communications. Stochastic neuron models are also biologically more plausible than deterministic models as they can better model the internal noise and uncertainty of the biological neurons. Preliminary experiments and simulations showed that it is also possible to evolve different stochastic functions using LUTs in the FPGA giving a rather continuous range for parametric flexibility needed for evolvability of the neuron types and synaptic plasticity. Distributed models appear to have a better convergence speed than centralised stochastic models as they exploit many more PEs. With the right design, the inherent redundancy, modularity, and parallelism in distributed stochastic models can also lead to fault-tolerance. They can also model the distribution of the ion channels in space better than centralised models. This may prove useful in evolving local nonlinearities in the dendrite trees. The difference between centralised and distributed stochastic models shows a trade-off between hardware compactness and performance.

Although stochastic models can lead to very bio-plausible, simple, compact, robust, flexible, and evolvable neuron models, the fundamental trade-off between their reliability and efficiency may not lead to very efficient bio-plausible designs on FPGAs compared to deterministic models. Therefore, deterministic models are also investigated in the following section.

4.4 Deterministic Models

In order to reduce the noise, it is possible to use deterministic bitstreams for intercellular communication on a single wire. A range of different approaches from deterministic bitstream methods to using traditional serial binary bitstreams are available.

4.4.1 Using Uniformly-weighted Bitstreams

A number of methods that use uniformly weighted but deterministic bitstreams are proposed in [49, 286]. These methods are similar to stochastic computing methods in the sense that all the bits in the bitstream have the same weight and some operations such as multiplication can be simply performed using very simple logic circuits. However, by generating the bitstreams deterministically they reduce the noise to the level of the quantisation error. The noise level is reported to be of $\mathcal{O}(1/n)$ of the bitstream length

[49], while stochastic noise is of order $\mathcal{O}(1/\sqrt{n})$. This can lead to much better performance as only a bitstream of length 128 bits could be sufficient to achieve a SNR of 42dB. However, the summation and other operations using such deterministic schemes need complex logic circuits, which adds to the hardware resources and design complexity. These models are mainly proposed for the rate model neural networks. In rate model neural networks, unlike spiking neuron models, multiplication of the synaptic weights by input signals is a major part of the computation load. In spiking neuron models, however, the major computation load is summation in the dendrite and synapses, and calculation of a nonlinear function in the soma. As the number of synapses is much higher than number of neurons, it makes sense to simplify the circuitry for the synapse and dendrites with the price of more complex hardware in the soma.

4.4.2 Using Binary Bitstreams

A traditional approach is to use a binary representation for the bitstream. Starting the bitstream from LSB (Least Significant Bit) allows relatively simple circuits for serial arithmetic operations such as summation, accumulation, and other operations. A serial adder can be implemented efficiently with a single Virtex-5 6LUT and a flip-flop. A shift register, which is abundant in the FPGA can be used to store synaptic weights efficiently. Although flexible nonlinear operations are not as simple and efficient as stochastic computing, it is still simpler than parallel arithmetic circuits and its complexity is less dependent on the representation length (n). The only part of the hardware that depend on the length of the representation are shift registers that store the variables (such as membrane potential, and synaptic weights) and constants (such as soma parameters). The length of the shift registers are of order $\mathcal{O}(n)$.

A bitstream of length n can represents 2^n different values. Based on the SNR calculation of a binary representation (equation 4.8), the quantisation noise of a binary bitstream of length n bits is equal to $6.02 \cdot n$ dB. That means to achieve a SNR of 42dB, a binary bitstream of length 7 bits is sufficient. This leads to a higher performance than any other bitstream-based model. The speed in bitstream-based models is of order $\mathcal{O}(1/n)$, where n is the length of the bitstream, and models with binary representation need the shortest length to achieve the same quantisation error. Moreover, the inherent noise in measurement of the stochastic representation does not exist in deterministic models.

4.4.3 Distributed Binary Systems

It is conceivable to distribute the nonlinear computation of the soma dynamics over different PEs in the dendritic tree. Due to the large number of synapses compared to number of neurons in cortical circuits, and the large difference between complexity of the computations in the soma and dendrites, it does not make sense to distribute the soma processes over PEs in the dendritic tree. However, it may prove efficient in practice to keep the non-linear interaction between synapses (and other possible PEs) on the dendritic tree distributed rather than centralising them in the soma PEs.

4.5 Summary of the Design Options

Here we summarise different approaches to neuron model design, their challenges, important factors, and trade-offs between these major factors and constraints. We assumed a minimum required level of bio-

plausibility and feasibility to narrow down the exploration to models based on the general architecture of figure 4.1, then investigated the properties of stochastic and deterministic, distributed and centralised neuron model designs.

Efficiency

Computation using bitstreams can lead to rather efficient spiking neuron models both in terms of performance and compactness (intracellular communication and other hardware resources) compared to traditional models based on parallel communication and arithmetic. Distribution of processing, storage, and communication functions over FPGA area is not only bio-plausible, but also prevents bottlenecks of central modules such as shared memories for network activity, or variables and constants, shared buses for intracellular communications, and shared PEs of a group of neurons or synapses. Such distribution is also compatible with FPGA design practices and can effectively leverage the potentials of the FPGA.

Bio-plausibility

Direct mapping of network connectivity to neuron intercellular communication channels (spike transmission on axons) not only leads to a more bio-plausible model of distributed communication, but also allows a free choice of neural codings suitable for a given problem to emerge through evolution. Bus and memory-based models are bound to make biologically implausible assumptions about the neural coding to provide the bandwidth needed for such massively-connected networks or sacrifice time-accuracy of the spikes. Distribution of the storage and processes over the synapses and dendritic PEs can effectively capture the distribution, locality, redundancy, parallelism, fault-tolerance and modularity of the natural neurons.

Using bitstreams allows flexible single-wire communication between PEs with minimum routing resources. Both stochastic and deterministic models can be designed in a rather flexible way to model the dynamic connectivity of biological neural microcircuits, and flexibility of the soma and synapses. Designs based on bitstreams can also capture the heterogeneity, distribution, locality, parallelism, and modularity of natural processes in the brain. Stochastic and particularly distributed stochastic designs can lead to very bio-plausible models of the neuron that capture the stochastic nature of the biological processes. Distributed stochastic models also show a high level of redundancy and fault-tolerance.

Performance-Reliability Trade-off

There is an inherent trade-off between accuracy and performance of the stochastic models. While each membrane update cycle needs at least n clock cycles (for measurement of n bits of the stochastic bitstream), the noise level is of order of $\mathcal{O}(1/\sqrt{n})$. To achieve the required bio-plausible accuracy (a minimum SNR of 42dB) the stochastic models need about $n = 2^{14}$ clock cycles for each update cycle. Noise level of models based on uniformly-weighted deterministic bitstreams is of order $\mathcal{O}(1/n)$ while deterministic models based on serial binary arithmetic have a noise level of order $\mathcal{O}(1/2^n)$.

Compactness-Reliability Trade-off

There is also a trade-off between the required accuracy and the hardware resources in these models. Both distributed and centralised stochastic models have a trade-off between accuracy and the length of

the measured stochastic bitstream. The hardware resources needed for such measurement is of order $\mathcal{O}(\log_2 n)$ where n is the length of the bitstream needed to achieve the required accuracy with a quantisation noise of order $\mathcal{O}(1/n)$ and stochastic noise of of $\mathcal{O}(1/\sqrt{n})$. Distributed stochastic models have an extra trade-off between the number of PEs and their accuracy that significantly impacts the compactness of the neuron model. Simulation results showed that to achieve a SNR of 42dB over 2000 PEs with more than 6000 state bits are needed.

The hardware resources needed for linear and non-linear operations in deterministic models do not depend on the representation length and are of order $\mathcal{O}(1)$. However, deterministic centralised models need hardware resources of order $\mathcal{O}(n)$ for storing the variables and constants while the quantisation noise of the uniformly-weighted and binary bitstreams are of orders $\mathcal{O}(1/n)$ and $\mathcal{O}(1/2^n)$ respectively. Although hardware resources needed for deterministic distributed model are not precisely assessed, it can be expected to be significantly more than deterministic centralised models.

Performance-Compactness Trade-off and Scalability

The above trade-offs can be also viewed as a general trade-off between performance and compactness of these neuron models. Centralised stochastic models are very compact but much slower than deterministic ones, due to the inherent measurement noise of the stochastic bitstreams. Deterministic binary models are much faster than stochastic models but need more hardware resources for processing. Both deterministic and stochastic models need the same amount of routing resources for intracellular communications. While a stochastic model can be used to implement a bio-plausible small-scale system of a few neurons on a FPGA, their performances are far from the hyper-realtime performance needed for an evo-devo system in FPGA. On the other hand, a deterministic serial arithmetic neuron model can deliver the required performance with a fair amount of hardware resources. Parallel arithmetic binary models can be n times faster than serial models (where n is the number of bits used for representation of variables and constants) but require hardware resources of order $\mathcal{O}(n)$, while serial arithmetic models only require that much resources for storage. Therefore, it must be feasible to use a deterministic serial binary model to implement a network of about 100 neurons in a small Virtex-5 FPGA.

Fault-tolerance and Robustness

Stochastic models can be very immune to noise. With a good design, distributed stochastic models can be also extremely fault-tolerant due to the redundancy in their structure. Deterministic models based on uniformly-weighted bitstreams are less sensitive to noise than models based on serial binary arithmetic.

Simplicity

Stochastic systems have a rather simple design but are harder to test and debug due to their unpredictable nature. Serial binary models are based on the traditional design principles, rather simple to design, test and debug. Deterministic models based on uniformly-weighted bitstreams are not simple to design but probably simpler to test and debug than stochastic models. Centralised models are always easier to design and test than distributed models.

Table 4.2: An evaluation summary of different design approaches and their trade-offs for the neuron model in the context of this case study assuming a fixed minimum required accuracy. The general trade-off between bio-plausibility and efficiency and dependancy of the fault-tolerance and robustness to bio-plausibility is clear in this general view of the trade-offs.

Design approach			Bio-plausibility	Efficiency	Performance	Compactness	Fault-tolerance	Robustness	Simplicity
Stochastic	Distributed	Distributed stochastic models	● ● ●	○ ○ ○	● ○ ○	● ○ ○	● ● ●	● ● ●	● ○ ○
	Centralised	Centralised stochastic models	● ● ○	● ○ ○	● ○ ○	● ● ○	● ● ○	● ● ●	● ● ○
Deterministic	Centralised	Uniformly-weighted deterministic bitstream	● ○ ○	● ● ○	● ● ○	● ● ○	● ○ ○	● ● ○	● ○ ○
	Centralised	Centralised models using binary bitstreams	● ○ ○	● ● ●	● ● ●	● ● ●	● ○ ○	● ○ ○	● ● ●
	Distributed	Distributed models using binary bitstreams	● ● ○	● ○ ○	● ● ○	● ○ ○	● ● ○	● ○ ○	● ○ ○

4.6 Case Study: Digital Neuron Model

In this section, based on the aforementioned analysis and insights, practical challenges in design and implementation of a simple but rather general bio-plausible neuron model that is feasible in Virtex-5 FPGA are investigated. This model will be used as a platform for practical evaluation of the constraints, trade-offs and challenges of the promising approaches, and also as a base for further investigations in the future chapters. Although the model is kept simple enough to be feasible in the time constraints of the present project, it is shown how this example design can be modified or extended to be more bio-plausible, efficient, flexible, scalable, and reliable. Here the design of the neuron model, called Digital Neuron Model, is explained along with more explanation of the detailed design options and justification of design decisions.

Table 4.2 shows an evaluation summary of different design approaches for a neuron model given a fixed accuracy in the context of this case study. Since the performance and efficiency are very important for an evo-devo neuron model, a deterministic model with binary bitstreams with central processing for the soma and distributed processing for dendrite is adopted. To reduce the number of connections and hardware area while maximising the speed and parallelism, a serial arithmetic bit-level parallel design is selected. Following the general bio-plausible architecture of section 4.2, each digital neuron consists of a set of synapse units and a soma unit connected in a ring architecture (called dendritic tree) shown in figure 4.5(a). While this brings some distribution and parallelism (between synapse processes), modularity, and redundancy to the model, it reduces the number of PEs in the system to the number of neurons and synapses. However, it is quite possible to extend the model to accommodate extra processing elements along the dendritic tree to introduce other features and nonlinearities. This architecture creates a 2-way communication channel and allows the development of different dendrite structures as demonstrated in the example of figure 4.5(b). The signal pairs that connect the units form a loop that conveys data packets (a start bit and 16 data bits). The soma unit sends an upstream packet containing the current membrane potential on its upstream output (USO). Synapse units pass upstream packets unchanged but process downstream packets. The spike input of each synapse is connected to the axon of the pre-synaptic neuron. If a synapse unit receives a pre-synaptic action potential, it adds (subtracts) its synaptic weight to the first arriving downstream packet. Therefore, the soma unit receives the sum of membrane potential and post-synaptic currents in its downstream input (DSI). After processing this packet, the soma unit sends another packet with the updated membrane potential. Serial arithmetic is used in all the units to create pipelined parallel processing inside each neuron, meaning that neighbouring units process different bits of the same packet at the same time adding to the efficiency of the system. Using this architecture has a number of collective benefits. First, a 2-way communication channel makes it possible to have a local synaptic plasticity mechanism in each synapse leading to a higher level of parallelism. Most of the bio-plausible unsupervised learning mechanisms like STDP [344] and its variants involve a local learning process in each synapse. Secondly, it minimises the number of local and global connections, which leads to a significant relaxation of constraints imposed upon the network architecture, as limited routing resources is the major constraint in optimal utilisation of FPGA functional resources. Each unit

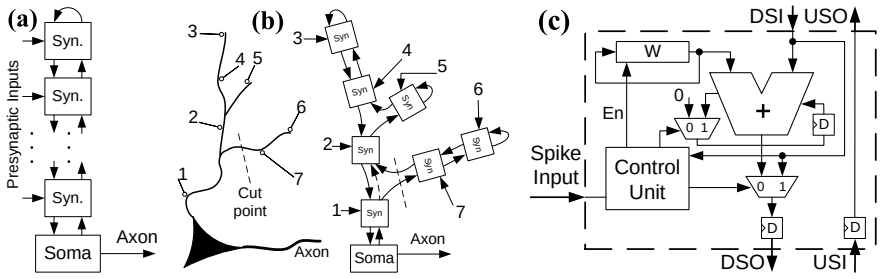


Figure 4.5: a) General architecture of the digital neuron (Syn=synapse) b) Example of the dendrite structure and its adaptability (c) Synapse unit architecture.

needs only a global clock signal to work. Another global reward or punishment signal can be added for a supervised learning mechanism. Although other architectures may bring about less pipeline latency, they need more local and global connections. For instance, a binary tree structure similar to [331] needs about double the number of local connections including the upstream links (excluding the global control signals). Third, it allows developing any dendrite structure similar to biological dendrites. The user is free to trim (add) dendrite sub-trees at any point simply by cutting (connecting) a (pair of) connection(s) and bypassing (inserting) the root unit of the sub-tree as shown by the dashed lines in figure 4.5(b). This can be implemented in FPGA using multiplexers or other routing resources as explained in section 5.4 . This flexibility is vital for a developmental model that needs on-line growth and modification. Fourth, it maintains the regularity of the model by reducing the diversity of the module types (synapse and soma units) and connection types (dendrites, axons) to a biologically plausible bare minimum. This simplifies the place and route or dynamic reconfiguration process if a regular infrastructure of cells and connections (similar to [377]) is used. Finally, it is possible to add other variables to the data packet (*e.g.* the membrane recovery variable in the Izhikevich model [171]) if required.

4.6.1 The Synapse Unit

The synapse unit, shown in figure 4.5(c), comprises a 1-bit adder, a shift register holding the synaptic weight, two pipeline flip-flops, and a control unit. The upstream input (USI) is simply directed to the upstream output (USO) through a pipeline flip-flop. The control unit disables the adder and weight register when no spike has arrived by redirecting the downstream input (DSI) to the downstream output (DSO) through another pipeline flip-flop. When the control unit detects a spike, it waits for the next packet and resets the carry flip-flop of the adder when it receives the start bit. Then it enables the shift register and the adder until the whole packet is processed. Using a shift register with a feedback loop allows compact storage and retrieval of the synaptic weight. Moreover, a learning block can be simply inserted into the feedback loop of the weight register in order to realise an unsupervised local learning mechanism like STDP [344]. This learning block can access the current membrane potential and the axonal input as well. It is also possible to modify the synapse to create a digital postsynaptic current input by loading the input voltage into the weight shift register serially or in parallel.

4.6.2 The Soma Unit (PLAQIF model)

Here, a new Piecewise-Linear Approximation of the Quadratic Integrate and Fire (PLAQIF) is designed as the soma model. This new model is published and presented in peer reviewed international conferences [338, 339], reviewed and cited in [303, 181, 32, 300, 18, 167, 295, 164, 163, 179] and inspired others to design similar models [2].

For this case study, a sufficiently bio-plausible but simple and compact model is needed. Very bio-plausible models such as Izhikevich are beyond the time and hardware budget of this project. However, as will be shown later, the PLAQIF model can be simply turned into an approximation of the Izhikevich model with more hardware resources and design time. Most of the hardware models are based on the LIF [171] or simplified LIF neuron models [311, 374]. However, a Quadratic Integrate and Fire neuron model (QIF) is biologically more plausible compared to the popular LIF model as it has dynamic threshold and resting potentials, can generate bio-plausible spikes with latencies and has two bistable states of tonic spiking and silence [171]. Using a PLAQIF model has a number of benefits. While it is relatively inexpensive (in terms of hardware resources) to convert a serial arithmetic implementation of a LIF neuron model into a PLAQIF model (as shown later), PLAQIF model can generate a bio-plausible action potential. This is particularly important as we may use the membrane voltage in the learning process. Moreover, the behaviour of the model can be specified with a number of parameters (*i.e.* time constants and reset potential). These parameters can be placed in shift-registers and look-up-tables (LUT) to be modified at run-time (*e.g.* by partial dynamic reconfiguration) or can be hard-wired for hardware minimisation. Finally, it is easy to extend this model to a piecewise-linear approximation of Izhikevich model (with a wide range of bio-plausible behaviours *e.g.* bursting, chattering, and resonating [171]) by adding a second variable and a linear equation., if hardware budget permits. The hardware circuit for this second equation can be accommodated in the soma unit or even in a special type of synapse or dendritic unit.

The dynamics of the QIF model can be described by a differential equation and reset condition of the form [171]:

$$\dot{u} = a(u - u_r)(u - u_t) + I \ , \quad \text{if } u \geq u_{peak} \text{ then } u \leftarrow u_{reset} \tag{4.11}$$

where u is membrane voltage, a specifies the time-constant, I is the postsynaptic input current, and u_r and u_t are nominal resting and threshold voltages (when $I = 0$) respectively. Note that in contrast with LIF models, the actual resting and threshold voltages are dynamic and they change with input current I [171]. Applying first-order Euler method results in an equation of general form:

$$u_{k+1} = u_k + a(u_k - u_r)(u_k - u_t) + I_k \ , \quad \text{if } u_{k+1} \geq u_{peak}, \text{ then } u_{k+1} \leftarrow u_{reset} \tag{4.12}$$

where k is the step number. The design of the PLAQIF model starts with a serial arithmetic implementation of a LIF model with equation $u_{k+1} = u_k + I - au_K$ (for $a < 1$), and some modifications. The last term can be computed using two taps:

$$u_{k+1} = u_k + I_k + \underbrace{\left\lfloor \frac{u_k}{P_1} \right\rfloor}_{\text{Tap 1}} + \underbrace{\left\lfloor \frac{u_k}{P_2} \right\rfloor}_{\text{Tap 2}} \tag{4.13}$$

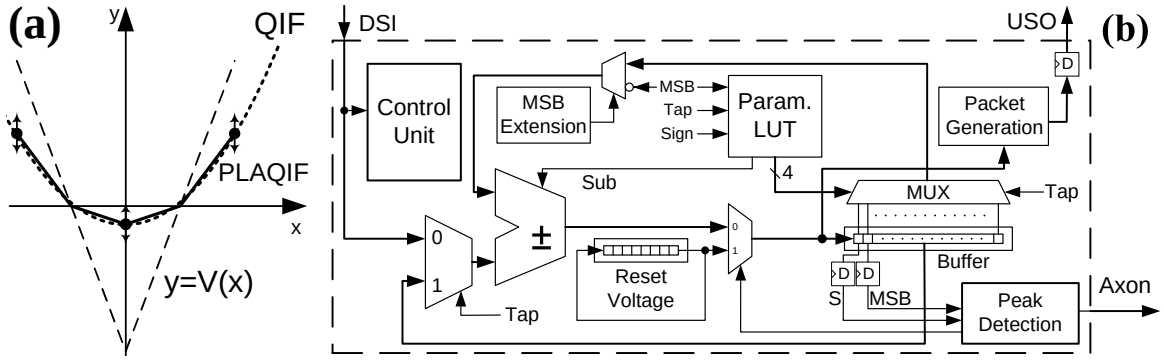


Figure 4.6: (a) The PLAQIF model approximates the QIF model (the dotted curve) with a piecewise linear function by modulating the V-shape function $V(x)$. The control points (arrows) can be moved by tuning the parameters. (b) Soma unit

where $P_i = (-1)^{s_i} \cdot 2^{p_i}$ with p_i and s_i being the parameters of ith tap. Each tap is computed by adding (or subtracting depending on s_i) the shifted version (arithmetic shift right by p_i bits) of the binary representation of u_k. By replacing the sign bit (`S`) and the most significant bit (`MSB`) of u_k with the complement of `MSB` we can produce the piecewise linear function $V(u_k) = |u_k| - 2^{14}$ (assuming a 16-bit representation). This function is shown in figure 4.6(a) as the V-shape function. By tapping (modulating) $V(u_k)$ with different parameters ($p_{i,0} \dots p_{i,3}$ and $s_{i,0} \dots p_{i,3}$) for different combinations of `S` and `MSB` (positive or negative, small or large values of u_k) we get:

$$u_{k+1} = u_k + I_k + \left\lfloor \frac{V(u_k)}{P_1(u_k)} \right\rfloor + \left\lfloor \frac{V(u_k)}{P_2(u_k)} \right\rfloor \tag{4.14}$$

$$\text{where } P_i(x) = (-1)^{s_{i,j}} \cdot 2^{p_{i,j}} \ , \quad j = \left\lceil \frac{x}{2^{14}} \right\rceil + 1 \tag{4.15}$$

which is a piecewise-linear approximation of a quadratic function shown in figure 4.6(a) as PLAQIF. It is possible to approximate equation 4.12 with equation 4.14 by changing the parameters $p_{i,j}$ and $s_{i,j}$ as shown in figure 4.6(a).

The soma unit, shown in figure 4.6(b), comprises a 1-bit adder, a 32-bit buffer shift register (holding the partial sums from the last cycle), a 16-bit shift register (holding reset voltage u_{reset}), a lookup-table (LUT, a 8x5 bits RAM, which holds the parameters $p_{i,j}$ and $s_{i,j}$), a control unit (CU, which detects the arriving packet and generates all the control signals *e.g.* `Tap`, `ShiftEn`, etc.), and a few multiplexers. The soma unit initiates a data packet thorough USO and waits for a packet on DSI input. At this point, the buffer holds the value u_k in its left half and `S` and `MSB` flip-flops hold the sign and most significant bit of u_k. The LUT selects the correct shifted version (according to `S` and `MSB`) of u_k through the multiplexer and has its first bit ready on the input of the adder. The first tap starts with receiving a packet. An arriving packet, which contains the value $u_k + I_k$, goes to the other input of the adder. The LUT also selects the add or subtract operation in each tap (s_i). As the operation goes on, the MSB extension block switches the multiplexer to $\overline{\texttt{MSB}}$ at the right time to generate the value $\left\lfloor \frac{V(u_k)}{P_1(u_k)} \right\rfloor$ on the input of the adder. Therefore, the new value of $u_k + I_k + \left\lfloor \frac{V(u_k)}{P_1(u_k)} \right\rfloor$ shifts into the buffer through a multiplexer. The second tap starts immediately and the value in the left half of the buffer goes to the adder input. The other input

of the adder is again $\left\lfloor \frac{V(u_k)}{P_2(u_k)} \right\rfloor$ now generated by selecting the correct shifted version of the u_k from the right half of the buffer. The adder generates the updated value of u (u_{k+1} in equation 4.14) at its output, which is shifted into the buffer and is also used to generate a new packet in the upstream output of the soma unit. This value is also used to update the `S` and `MSB` flip-flops according to the new value of u_{k+1}. This process continues until the peak detection block detects a transition of `S` without any change in `MSB`, which indicates an overflow, and immediately corrects the sign bit in the departing packet, generates a pulse in the axon, and initiates the absolute refractory period. The absolute refractory period, which lasts for a complete cycle, is like any other cycle except that in the second tap the output of the adder is ignored and contents of the reset voltage shift register is used instead as the new membrane potential u_{k+1}. The membrane update period ($T = 2N + 18$ clock cycles), and thus neuron time constants, depend on the number of synapses (N). This can be compensated by evolving parameters or corrected in future designs using a padding shift register, if needed.

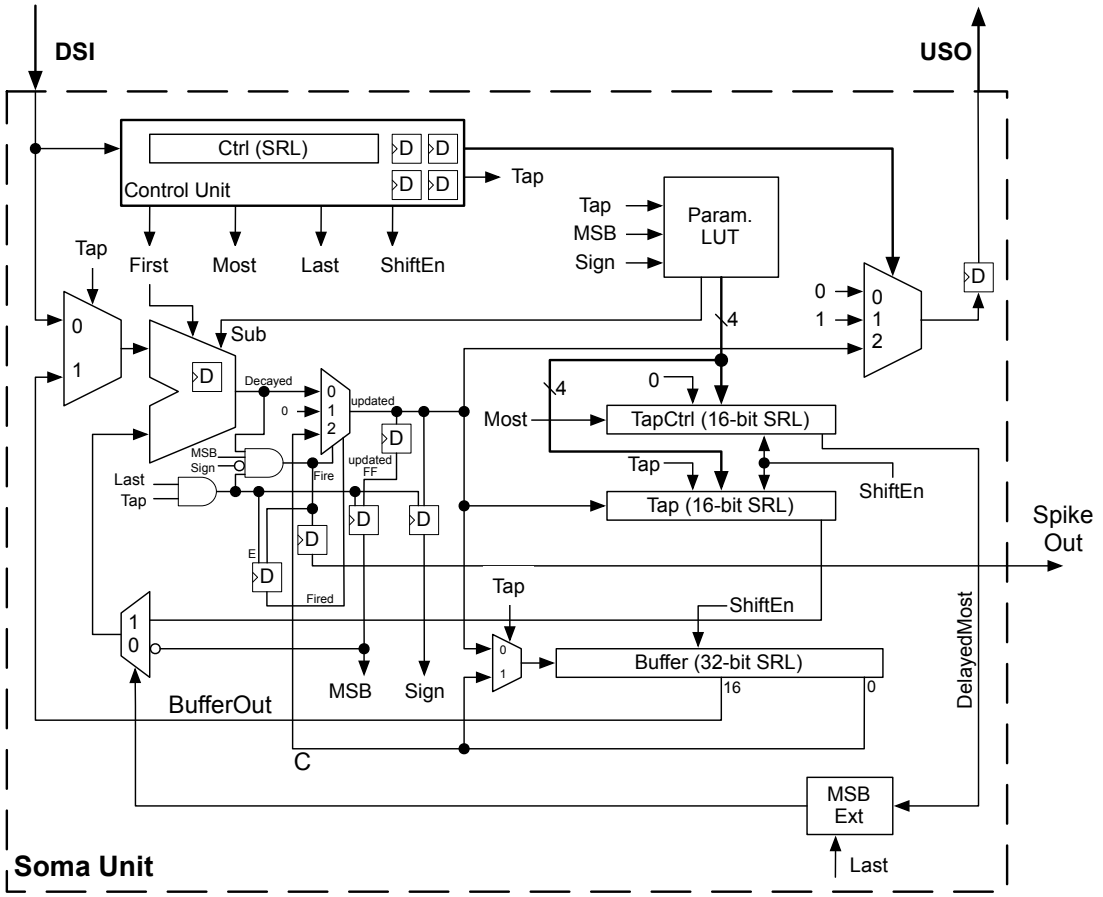


Figure 4.7: Detailed block diagram of the soma unit design.

4.6.3 Implementation and Testing

Figure 4.7 and 4.8 show the detailed design of the soma and synapse units using the 16 and 32-bit shift registers (SHR) available in the Virtex-5 SLICEMs. For minimising the hardware resources, current membrane potential and reset voltage are both stored in a single 32-bit SHR and multiplexers were

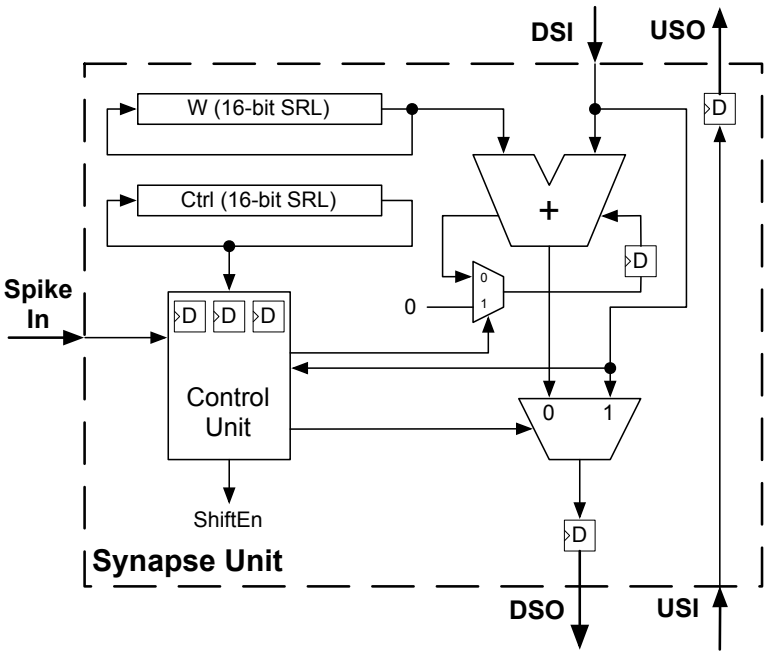


Figure 4.8: Detailed block diagram of the synapse unit design.

used to maintain the values in the shift register through both taps. For shifted version of the membrane potential and syncing the last bit of the shifted version, two other SHRs (Tap and TapCtrl) are used. Another SHR is used in the control unit for generation of the delay in the state machine. Control unit generates all the control signals: `Tap`, determines the first and second tap of the processing; `First` is asserted when the first data bit is being processed; `Most` is asserted when the most significant bit (before sign bit) is being processed; `Last` is asserted when the sign bit is being processed; `ShiftEn` enables the shift functionality of the shift registers (SHRs).

The behaviour and flexibility of the neuron model was verified by VHDL simulation of a single neuron. The membrane potential packets that are sent out of the soma unit can be decoded as a signed binary number and its dynamics monitored against the input spikes. Random spikes were fed into 16 synapses with different weights using different bio-plausible parameter settings and its membrane potential was monitored and compared to the expected dynamics of equation (4.14). This was performed both in VHDL simulation and on FPGA chip and the results were confirmed to be the same.

A random small-world network of 161 16-bit neurons with 20 inputs, 20 outputs, and 10 fixed-weight synapses per neuron was simulated and synthesised for a XC5VLX50T chip using VHDL and Xilinx ISE, resulting 85% utilisation and a maximum clock frequency of 160MHz (4210 times real-time real-neuron simulation speed with a 1 ms resolution). It is possible to improve some of these figures by low-level design optimisation and a cellular floor planning similar to [377] (as described in section 5.4).

4.6.4 Experiments

To verify the functionality and the parametric flexibility of the new PLAQIF neuron model and to compare its behaviour and capabilities with those of biological neurons and hardware neuron models, four

experiments were carried out. To explore a wide range of behaviours, arbitrary different parameter settings were selected.

In the first experiment, we checked if the neuron model is capable of showing both bistable and monostable behaviours of biological neurons. For bistable behaviour the u_{reset} was set to 17000 and for monostable behaviour $u_{reset} = -16384$. The other parameters were set as follows:

$$P_1(x) = \begin{cases} 2^7 & 0 \leq x \\ -2^7 & x < 0 \end{cases}$$

$$P_2(x) = \begin{cases} 2^5 & 0 \leq x \\ -2^5 & x < 0 \end{cases}$$

In the second experiment, to check the effect of changing u_{reset} on the F-I curve (spiking frequency against input current) of the neuron, the F-I curve was recorded using different values of the parameter u_{reset}, keeping all other parameters fixed as follows:

$$P_1(x) = P_2(x) = \begin{cases} 2^4 & 2^{14} \leq x \\ 2^3 & 0 \leq x < 2^{14} \\ -2^4 & -2^{14} \leq x < 0 \\ -2^3 & x < -2^{14} \end{cases}$$

In the third experiment, the F-I curve was recorded changing the middle control point (in figure 4.6(a)) keeping all other parameters fixed ($u_{reset} = -16384$ and for two other control points: $P_1(x) = 2^7$ and $P_2(x) = 2^3$ when $|x| \geq 2^{14}$).

In the fourth experiment, only u_{reset} was fixed at -16384 and the F-I curves for a few different symmetric settings of $p_{i,j}$ and $s_{i,j}$ (where $P_i(x) = -P_i(-x)$, $i = 1, 2$) were recorded. For comparison, a QIF model of the form:

$$\dot{u} = 0.1u^2 + 1.25 \times 10^4 I - 0.27175 \qquad (4.16)$$
$$\text{if } u \geq 30 \text{ then } u \leftarrow -1$$

was also simulated with the same resolution.

All experiments were carried out using VHDL simulation of a single neuron with 16 synapses with their weights set to $2^0, 2^1, \cdots, 2^{14}, 2^{15}$. Each frequency measurement was made by first setting all the synaptic inputs to zero for 2 membrane update cycles and then fixing the binary representation of the input current on the synaptic inputs, waiting for the first spike in the axon and counting the number of update cycles until the second spike (N). The frequency was then calculated assuming that each update cycle is equal to 1 ms of neuron simulation time ($f = \frac{1000}{N}$).

4.6.5 Results

Figure 4.9 shows traces of the input current, membrane voltage and the axon output of the digital neuron in two different settings of the first experiment. The PLAQIF model is clearly capable of working in both bistable and monostable modes and generating spikes with latencies. This is in contrast with the popular LIF model that works only in the monostable mode and cannot generate spikes with latencies.

Figure 4.10 shows the results of the second experiment that demonstrates the effect of changing the parameter u_{reset} on the F-I curve of the neuron. The F-I curve clearly shows that the digital neuron is class 1 excitable [171] for $u_{reset} < 0$. The PLAQIF model F-I curve appears as a class 2 excitability [171] for $0 \leq u_{reset} \leq 16384$. This is also an advantage over LIF model. Moreover, changing u_{reset} affects the general slope and curvature of the neuron F-I curve. For positive values of u_{reset}, the minimum spiking frequency and current change as u_{reset} changes.

Figure 4.11 shows the results of the third experiment. A class 1 excitability is clearer in this figure. It also shows how the slope and curvature of the F-I curve can be fine-tuned by changing the middle control point (in figure 4.6(a)) parameters. The bold lines show the F-I curve when the middle control point is higher than zero (W shaped function instead of V-shaped or quadratic function for $\dot{u}(u)$). These exotic nonlinearities in $\dot{u}(u)$ that do not match contemporary biological neurons can be exploited during evolution.

Results of the last experiment, shown in figure 4.12, demonstrates diversity of neuron characteristics using different parameter settings without changing u_{reset}. The F-I curve of the QIF model of equation 4.16 is shown in bold, which is close to the F-I curve of the PLAQIF model with the parameter settings:

$$P_1(x) = \begin{cases} 2^7 & 0 \leq x \\ -2^7 & x < 0 \end{cases}$$

$$P_2(x) = \begin{cases} 2^3 & 0 \leq x \\ -2^3 & x < 0 \end{cases}$$

It is an acceptable approximation, however the PLAQIF curve does not exactly match the QIF curve due to the piecewise-linear approximation.

The step-wise shape of the curves (particularly in higher frequencies) is due to the 1-millisecond simulation resolution and calculating the frequency based on the number of 1-millisecond update cycles between two successive spikes.

4.7 Practical Considerations

The practical challenges, options, and issues in the design and implementation of the neuron model are further discussed and summarised here. Possible modification and extensions that were not implemented in the time frame of this project are also discussed and elaborated in this section.

Bio-plausibility

In terms of bio-plausibility of the neuron model used as the basis for this design, QIF is one equation short of the Izhikevich model that is one of the very bio-plausible models for simulation of cortical neurons. It is possible to simply add more circuit to the soma unit to implement this functionality as the second variable is governed by a linear differential equation.

The dendritic tree structure is very flexible as it allows adding and growing branches or removing them by simply adding synapse units and pipeline flip-flops to the tree. This model does not propose any specific spike transmission scheme and therefore leaves the flexibility and time accuracy of the spike transmission structure to the later stages of the design (in the next chapter).

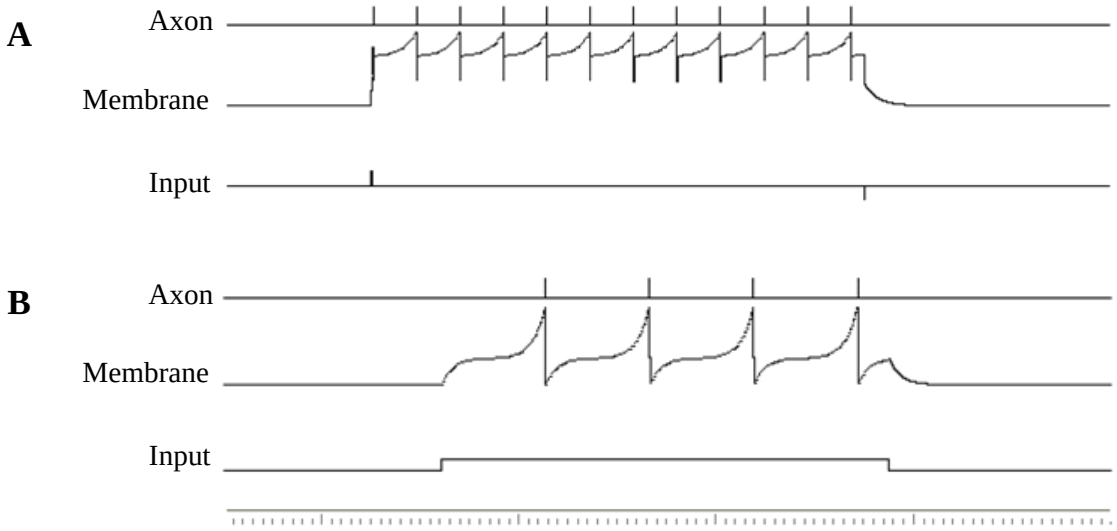


Figure 4.9: Traces of the input current, membrane potential, and the axon output of the digital neuron in the first experiment: A) Bistable behaviour ($u_{reset} = 17000$). B) Monostable behaviour ($u_{reset} = -16384$).

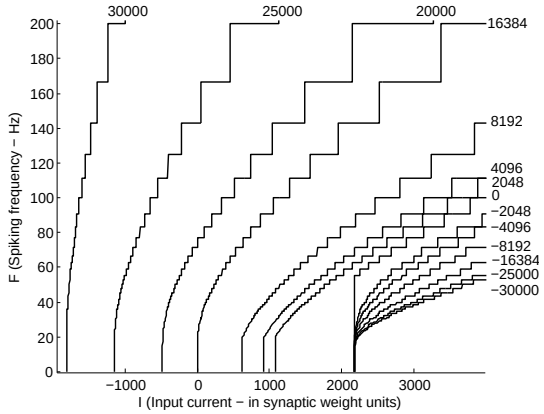


Figure 4.10: F-I curve of the digital neuron using different values of u_{reset} showing class 1 and 2 excitability.

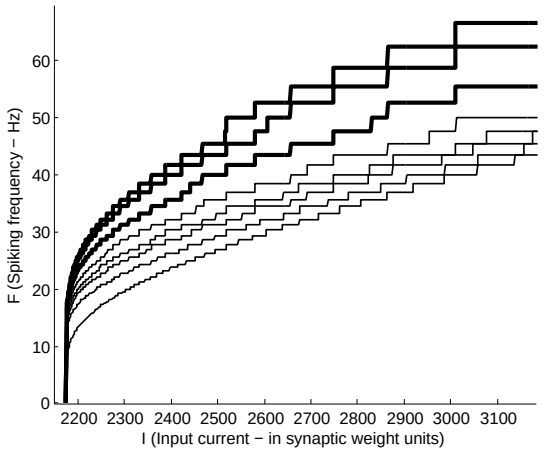


Figure 4.11: The effect of moving the middle control point (of figure 4.6(a)) on the F-I curve of the digital neuron. The bold lines are results of the middle point higher than zero.

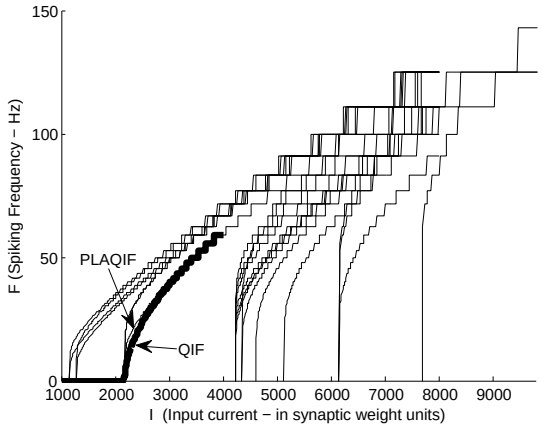


Figure 4.12: F-I curve of the digital neuron using different parameter settings for $p_{i,j}$ and $s_{i,j}$ keeping $u_{reset} = -16384$ along with the F-I curve of the QIF model of equation 4.16 superimposed in bold.

The synapse unit is designed to be very simple in order to minimise the hardware resources as the number of synapses in the cortical microcircuits is usually a few order of magnitude larger than number of neurons. However, the model is flexible enough to accommodate unsupervised and supervised learning processes. A synapse plasticity circuit can be simply inserted in the feedback loop of the shift register that stores the synaptic weight. This circuit can be designed to use serial binary arithmetic to use the timing of the pre-synaptic spike and the value of the membrane potential (or other variables if available on the dendritic loop) to modify the synaptic weight. As both this process and the synaptic value are local they can be integrated into the developmental processes that control the growth of the neurites and formation/elimination of the synapses. A global reward or punishment signal can be also used to modify some of the synaptic weights in a supervised learning model.

Moreover, it is also possible to use a few more shift registers or longer ones to create more complex and bio-plausible synapse models. Bio-plausible synapse models have an exponential response to pre-synaptic spikes. Synapse model used here only adds synaptic value to the membrane potential once in one update cycle. It is easy to store a few synaptic values in a longer shift register and modify the control unit to keep the synapse unit active for a few update cycles. Assuming a 16-bit representation, a synapse response function of 4ms (4 different values) needs only one extra flip-flop and an extra 32-bit shift register already available in the FPGA slice. With an 8-bit representation, response functions up to 8ms (8 values) can be implemented with the same hardware. This way it is not only possible to simulate sophisticated synapse models but also change the response function of each synapse by changing the values in the shift register(s). The parametric flexibility of the soma model that allows each neuron to have a different nonlinear update function allows developing heterogeneous microcircuits.

To incorporate the nonlinear interactions between distal synapses it would be possible to use a single synapse unit to serve as many synapses but depending on the number of virtual synapses receiving a spike the synapse unit may use a different set of shift registers. Also synapse facilitation and depression dynamics can be added to the synapse model similarly by modifying the control unit and adding more shift registers to store facilitated and depressed synaptic weights.

Compactness

All the above modifications and improvements are feasible as the original synapse model can be implemented with two 16-bit shift registers, one LUT, and few flip-flops that all take less than one SLICEM on Virtex-5 FPGA. Half of the Virtex-5 LUTs can be used as 32-bit shift registers. Each SLICEM in Virtex-5 has 4 of these LUTs. In the original implementation 2 LUTs (one shift register and one logic LUT) are used for control unit and one LUT for synaptic weight shift register. With a 16-bit representation it would be possible to use the same shift registers to double the representation length, use the synapse PE to serve two virtual synapses, add a 2 value synapse response function, or add synapse facilitation and depression features to the synapse model. Using two other shift registers in the SLICEM or reducing the representation length can provide enough hardware resources to add many of these feature to the synapse unit without going beyond that one-slice hardware footprint. However, there is a trade-off between the bio-plausible features that can be packed into the soma and synapse units and the compactness of the neuron model.

Performance

The example design can update the membrane potential variable every $T = 2N + n + 2$ clock cycles where N is the number of synapse units ($2N$ is the length of the dendritic tree loop), and n represents the length of the binary representation for variables and constants in the model. For $n = 16$ and a dendrite of $N = 10$ synapses (dendritic tree loop of 20 stages) this model updates the membrane every 38 clock cycles. With a clock frequency of 160 MHz this resulted 4.2 million updates per second, which assuming each update cycle to be equal to 1ms simulation of the biological neuron, it translates to 4210 times the speed of biological neuron.

Compared to [331] that also uses serial arithmetic, this model is 43% faster, but needs more hardware resources. However, it is difficult to compare these two designs due to differences in design objectives (flexibility, performance, and adaptability versus hardware minimisation in [331]) and technologies. Clearly, the replication and redundancy of the synapse control units increased the hardware resources but also contributed to the fault-tolerance of the neuron model, adaptability of the dendrite structure, and the real possibility of introducing meaningful development into the process.

This performance can be improved by trading-off other factors. For example by reducing the presentation length (n) to 8 bits the model performance can be increased by 26%. Using a synapse unit for serving more than one pre-synaptic input not only allows us to simply introduce nonlinear interactions between distal synapses to the neurite model, but also allows us to reduce the number of synapse units in the dendritic tree. This improves both the compactness of the model and the performance by shortening the dendritic loop. Using one synapse unit to serve up to 4 inputs can reduce the number of synapses in the current implementation from 10 down to 3 (2.5) that in turn improves the update frequency by 65% with a 16-bit representation and by 150% with an 8-bit representation.

Reliability

One of the main sources of the noise in the system is the quantisation error of the representation. A 42dB SNR can be achieved by a 7-bit representation. However, in this model a 16-bit representation is used to

be in line with the other hardware based bio-plausible models and allow space for evolution to change the effective range of variations and compensate for the limitations of the PLAQIF soma model. Therefore, there is a trade-off between the reliability of the model and its efficiency. Experimental results may prove it possible to reduce the representation length down to 8 to 12 bits that can significantly improve the performance and compactness of the model and allow for much extra features to be packed int the same hardware area.

Scalability

At this stage that no routing resources were used for the implementation of the neural microcircuit on the FPGA, 161 neurons and 1610 synapses could be fitted in 85% of a XC5VLX50T. It appears that even after dedication of some resources to routing, it would be possible to develop microcircuits of about 100 neurons and 1000 synapses. High end Virtex-5 devices (XC5VLX330T) provide over 6 times more hardware resources, which gives 660 neurons and 6600 synapses on a chip.

Although this model is designed for single chip implementations it does not limit the techniques used for the spike transmission. The neurons and their dendrite structures are limited to a single chip but axons can extend to other chips and make synaptic connections to neurones on the other chips. Although this model recommends a direct mapping of axons to single wires for bio-plausibility and flexibility of the neural coding, it would be possible to use any inter-chip communication mechanism that suits the large scale system. In a direct mapping (one axon - one wire) the number of axonal connections between chips is limited by the number of FPGA user IO pins. However, multiplexing and high-speed data link cores available on Virtex-5 FPGAs can be exploited to increase the inter-chip bandwidth.

Fault-tolerance

The redundancy present in the design of this model allows a high level of fault-tolerance. Each neuron can work separately from other neurons relying only on a global clock signal. Each soma has its own control unit and assumes a closed dendritic loop. Each synapse unit has a separate control units and relies only on the integrity of the data packets on the dendritic loop. If a faulty synapse does not corrupt the passing packets the rest of the neuron and its synapses can be still functional. A faulty synapse or group of synapses that render a neuron malfunctioning can be detected by a developmental process (sensitive to very high and very low frequency of spiking). The developmental process can retract the dendrite tree resulting removal of the synapse units until the neuron is either back to normal regime or out of circuit. Modularity, distribution and local parallel processing and storage allows the developmental processes to utilise very bio-plausible fault-tolerance schemes at a higher level.

Complexity

Design and testing complexity of the system is directly related to the bio-plausible features of the soma and synapse model, and hardware optimisations to improve the performance and compactness of the neuron model. For example, using a single 32-bit shift register to hold two 16-bit variable or constant adds to the complexity of the design, testing, and debugging. However, the modularity of the neuron model enable the designer to focus on a single unit and follow a bottom-up approach in the design,

implementation, optimisation, testing and debugging of the whole system.

4.8 Summary

Figure 4.13 shows a graphical representation of the investigations carried out in chapter 4. First in this chapter, the importance of the neuron model, its design and impact on both bio-plausibility and feasibility of the whole evo-devo neural system were discussed. In section 4.1, different aspects of the bio-plausibility of the neuron model and different feasibility measures in this context were considered as general but tangible design factors such as flexibility, locality, heterogeneity, bio-plausible features, redundancy, modularity, distribution, parallelism, time accuracy, temporal dynamics, performance, compactness, efficiency, scalability, reliability, fault-tolerance, robustness, and complexity.

Based on these tangible design factors, some general design options regarding distribution of processing, communication, and storage functions over the FPGA area, type of intercellular and intracellular communications and their flexibility, and a bio-plausible approach to general design of the model were discussed. As a result, the investigation was focused on a group of designs based on a general flexible architecture for the neuron that comprised of a connected network of processing elements (PEs) that provide the infrastructure for bio-plausible simulation of neuron dynamics with the minimum use of routing resources of the FPGA devices.

After narrowing down the general design of the model, a few possible design approaches were examined. First, as the most promising bio-plausible approach, distributed stochastic models were investigated. Centralised stochastic models, deterministic models based on uniformly-weighted bitstreams, binary bitstreams, and distributed deterministic models were also explored in turn. Challenges, major factors, constraints, and general trade-offs in all these design approaches were summarised in section 4.5.

By Applying the analysis of the design factors and options to the specific needs of this study, a new neuron model called Digital Neuron Model was designed as a case study, and also as a basis for further investigations in the following chapters. The detailed design, implementation, testing and debugging of the Digital Neuron Model were explained and practical considerations in the detailed design were summarised based on the same tangible bio-plausibility factors and feasibility measures of section 4.1. Based on this neuron model, in the next chapter, design and implementation challenges of a reconfigurable structure for development of neural microcircuits are investigated.

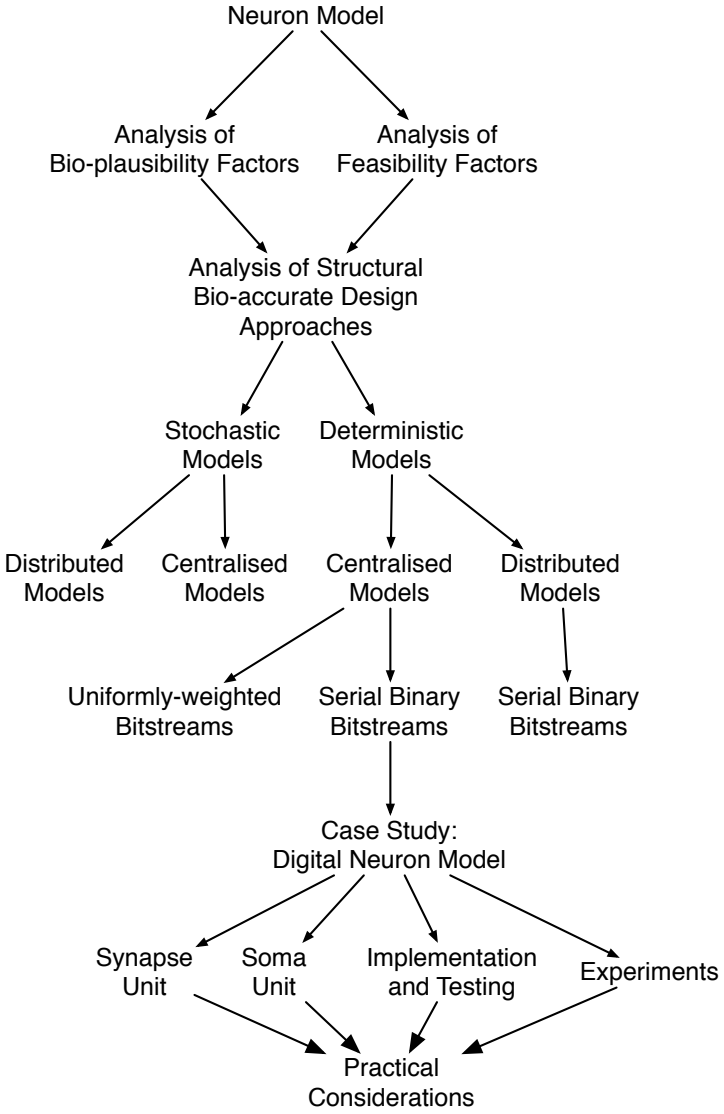


Figure 4.13: A graph of the investigations carried out in chapter 4 regarding the neuron model.

Chapter 5

Cortex Model

Biological neural microcircuits, grow in a neural tissue called cerebral cortex. It consists of the neurons and a large amount of supporting tissue that feeds the neurons and allow them to grow dendrites and axons and form synapses. Likewise, to grow neural microcircuits on FPGAs, neurons must be situated in a substrate that provides the physical resources for their functioning. In biological development, after the initial growth, chemical signals and proteins regulate and direct the function, growth, and connectivity of the cortex cells that already exist. These biological cells in the cortex are continuously modified instead of being regenerated every second. Similarly, neural microcircuits in FPGAs need a similar fixed tissue, which contains all the cells that can be modified by developmental processes. This modifiable initial structure on the FPGA that provides the basic functionality of the neurons and neurites is called a cortex model. The investigation of challenges in the design of this cortex model is the subject of this chapter while the next chapter focuses on the developmental and evolutionary process that continuously modify the cortex.

The design and implementation of such a structure, and the limitations that it may pose upon the phenotype have great impact on the bio-plausibility and feasibility of the whole system. The design and implementation of this reconfigurable structure can limit the flexibility, scalability, efficiency, fault-tolerance, reliability, and modularity of the neuron model implemented on it among other bio-plausibility and feasibility factors. It not only provides a platform for the simulation of the neural network but also specifies the boundaries and conditions for the developmental processes. Developmental processes will control the reconfiguration of this structure to form the neural microcircuits. Its limitations or flexibilities that it can provide for the developmental system can suppress or promote the evolvability and adaptability of the system. It can also have an impact on the learning process.

In this chapter, the challenges in the design and implementation of a reconfigurable cortex structure suitable for an evo-devo neural system based on the Digital Neuron model of chapter 4 are investigated. The investigation of the evolutionary and developmental processes that control and reconfigure the cortex is left to chapter 6. Here, different design factors and their trade-offs are highlighted and examined, and an example cortex model is designed and implemented as a case study.

5.1 General Design Factors

Similar to the previous chapter, here again, we start from the general definitions of bio-plausibility and feasibility, and their general measures from sections 2.1 and 2.2 and translate them into a set of general but tangible design factors and constraints in the context of the cortex model design. First in this section, we focus on the biological cortex and its properties to derive bio-plausibility related design factors and constraints that apply to the cortex model. These factors and constraints describe the expected features of a bio-plausible cortex model.

Next in this section, similarly, we focus on different feasibility measures to infer the feasibility related design factors and constraints. It will be then possible, in the next sections, to discuss the general trade-offs between these factors and constraints, and elaborate on the challenges. Given those challenges, it would be then practical to focus the exploration on promising areas of the design space and investigate the challenges of a few design options in depth.

5.1.1 Bio-plausibility Related Design Factors

The biological cortex is mainly composed of neurons and glial cells [213]. Biological glial cells provide support and nutrition for neurons and act as "glue" between them. Recently, they were suspected to be also involved in the synapse formation as well as axon and dendrite development [213, 293]. The majority of the cortical neurons are concentrated in a few layers of the surface of the neocortex known as the grey matter. The other major volume of the neocortex is white matter, which mainly consist of long-range axons (that connect different areas of the cortex) and supporting glial cells.

The majority of the important features of the cortex is already discussed in section 4.1.1 on the neuron model and here we only need to review them from a different perspective. Useful features for a bio-plausible cortex model will be examined and classified into five groups of factors that can affect five aspects of: neural network architecture, neural coding, the soma model, the neurite model, and the synapse model.

Factors Affecting the Neural Network Architecture

Statistical analysis of the networks of the neocortex and nervous systems indicates that they show characteristics of both small-world and scale-free networks [39, 51, 19]. The network connectivity of the cortex is also very dynamic [403]. Developmental processes not only control the position, density, and differentiation of the neurons in cortex but they also regulate the growth, formation, extraction, and elimination of the dendrites, axons, and synapses. Developmental processes can isolate a faulty cell or neurite branch and employ other cells and neurites instead. Neurons can grow dendrites and axons in different directions guided by chemical clues that are regulated by developmental processes. Synapses can form between axons and dendrites and soma cells. Redundant synapses and neurites can retract and disappear. As a matter of fact, elimination of the redundant connections and apoptosis (programmed cell death) play a major role in the development of the nervous system [404]. The placement of the neurons and their connectivity pattern appears to be optimised for the trade-off between high interconnectivity and interconnection cost. Also hierarchy and modularity are evident in the brain structure.

A useful bio-plausible cortex model needs to be dynamic and malleable. It must allow connections

between neurons to form and disappear. It must also allow long range axonal connections while encouraging locality and modularity with a bias towards short-range connections. The reconfigurable structure must provide evo-devo processes with all the available routing resource allowing those processes to optimise for the trade-off between complexity and resource utilisation to satisfy the requirements of the application problem. This requires the cortex model to provide development with information about availability of routing resources. Evo-devo processes must be able to regulate the ratio of resources dedicated to neurons and their connectivity.

A cortex model must also provide means for communication of the neural microcircuit with the outside world. A sufficient number of input and output signals must be available to feed stimuli to the network and collect the outputs.

Factors Affecting the Neural Coding

The characteristics of the neural coding on the cortex and nervous systems is already discussed in section 4.1.1. The reconfigurable structure needs to provide the maximum flexibility for the neural coding by minimising the assumption on the network activity, and signals temporal correlations. A high level of jitter in the spike transmission can corrupt the temporal information in the signals. The axonal delays also play a major role in the spiking neural networks. It is therefore important to maximise the time-accuracy of the spike transmission while providing reliable means for axonal delays.

In biological brains, long range signals such as hormones can regulate the general regime of the network activity through neuromodulation [73]. A bio-plausible cortex model need to support such long range or global signals that can be used by developmental or learning processes [172].

Factors Affecting the Soma Model

The network heterogeneity, parametric flexibility of its neurons, and possibility of changing these parameters during simulation are important bio-plausible features already discussed in section 4.1.1. The reconfigurable structure must not only allow evo-devo processes to adjust neuron parameters, but it must also provide those processes with neuron activity and other useful information on a time-scale larger than simulation, which can be used to regulate the development and maintenance of the network. The processing resources dedicated to the biological soma is more or less concentrated in a position in 3-dimensional space of the cortex. Biological neuron placement is controlled by the developmental processes through cell migration and differentiation [404].

Factors Affecting the Neurite Model

A biological neuron can grow an axon and many dendrites out of the soma cell in different directions. These neurites are malleable projections that form the means for communications between neurons. The flexibility of the neurites in terms of dynamic connectivity, growth and retraction, and developing non-linear interactions between distal dendrites are already discussed in section 4.1.1. This growth and retraction is controlled by the developmental processes. A bio-plausible cortex model must not only allow developmental process to regulate the growth and retraction of the neurites, but also it must be able to provide developmental process with local data about the activity of the axons and dendrites that

may affect their growth. Axons and dendrites in biological brains can fork into branches that grow in different directions [404]. A bio-plausible cortex model needs to allow developmental processes to create such branches in axons and dendrites. In a bio-plausible cortex model axons must have reliable delays proportional to their length.

Neurites must be placed, preferably, in a 3-dimensional space similar to the biological cortex that is the same (or mapped to the) 3-dimensional space used by the developmental processes, allowing formation of layers and regions in the cortex. The biological cortex is formed as a neural cortical tube [404] allowing for wrap-around connections at least in one dimension. A bio-plausible cortex model must support such local connectivity. The amount of resources (the space and supporting glial cells) dedicated to a neurite is proportional to its total length. Although neurites do not share space, they can pass each other without any interference. However, developmental processes must be able to control creation of synapses between axons and dendrites that are very close to each other.

Factors Affecting the Synapse Model

In a bio-plausible cortex model, developmental processes must be able to form synapses between axons and dendrites close to them. The elimination of those synapses must be also controlled by the developmental processes. The developmental and learning processes must be able to change the synaptic weight and other parameters of synapses. The cortex model must provide the developmental processes with enough information about the activity and state of the synapse for activity dependent neurodevelopment to occur. Activity dependent development means developmental and neural processes can influence each other and they must be executed concurrently. This may require dynamic partial reconfiguration or similar approaches that allow changing the connectivity and parameters of the neural microcircuit during the simulation. This also applies to reconfiguration of somas and neurites.

Having discussed the general design factors and constraints that can affect the bio-plausibility of the cortex-model, we can now focus on the design factors and constraints that affect the feasibility of the cortex model.

5.1.2 Feasibility Related Design Factors

The general factors related to the feasibility of the cortex model design can be analysed based on the feasibility measures defined in section 2.2. Except for the availability measure, which mainly applies to the hardware platform and is already covered in chapter 3, factors that can affect other feasibility measures are discussed here in the context of the cortex model design and implementation.

Factors Affecting the Hardware Cost

The compactness of the design and the silicon area that is needed for implementing a neural microcircuit using a cortex model affect the hardware cost. Part of this is already dictated by the neuron model and the amount of hardware resources needed for soma and synapse units. The rest are the hardware resources dedicated to the reconfiguration and routing of the cortex and supporting the soma and synapse units with connectivity and global signals such as reset and clock.

Factors Affecting the Performance

Although the performance of the cortex is partly determined by the performance of the neuron model, the rest of the hardware used for routing, connectivity and support of the synapse and soma units should allow those units to perform at the highest possible speed. This requires minimising the dendrite loop delay needed for a dendrite, which directly impacts the performance of the neuron model. Another aspect of the cortex model performance is the configuration speed. The cortex model must allow reconfiguration of the soma and synapse units and their connectivity in the minimum time. For concurrent execution of neural simulation and developmental processes (as in activity dependent development), reconfiguration needs to be repeated during the execution of neural simulation. Therefore, configuration speed will have a great impact on the overall performance of the system.

Factors Affecting the Scalability

Design of the cortex model must be scalable. This means it must be possible to implement a larger version of the same cortex design in a larger FPGA device or a group of interconnected FPGA devices. From a cortex model design point of view, this means a local routing mechanism must be used or any global mechanism must not be dependent on the size of the cortex. Moreover, preferably, the performance, efficiency, complexity, and reliability of the system must not be impacted by the size of the cortex.

Factors Affecting the Design and Testing Time and Complexity

It must be possible to use a modular approach for design, implementation, and testing of the cortex model to keep the design and testing time and complexity in a reasonable scale. It would be desirable to have a design that can be scaled in terms of complexity by adding extra features and spending more time for its design and testing. For testing and debugging purposes, the cortex model must allow probing and monitoring the internal signals and network activities.

Factors Affecting the Reliability

The cortex model needs to be reliable and robust to faults and errors. For example using asynchronous signals may lead to data loss or errors, affect the functionality of the reconfiguration process or neural simulation and may impact the reliability of the whole system. Any type of distribution, parallelism, and redundancy that can bring fault-tolerance and robustness can add to the reliability of the whole system.

5.2 General Design Options

The tangible feasibility and bio-plausibility related factors and constraints analysed in the previous section are summarises in table 5.1. Based on these general design factors and constraints, now, it is possible to investigate different general deign options and focus on the promising areas in the design space for further investigation. Looking at table 5.1 it appears that the cortex model needs to implement four main functions:

1. Intracellular communication (dendritic loops in the Digital Neuron model)

2. Intercellular communication (spike transmission and I/O)

Table 5.1: A summary of the tangible design factors and constraints in the design and implementation of the cortex model that can affect the bio-plausibility and feasibility of the system.

Bio-plausibility Related Design Factors	Feasibility Related Design Factors
Flexibility and evolvability of the soma cells in the cortex model that allow evo-devo processes to regulate the placement and density of the neurons and dynamically control the differentiation of the neurons in a 3D substrate similar to cortical tube	Compactness (minimisation of reconfiguration and routing hardware overhead)
Flexibility and evolvability that allow dynamic growth, retraction, modification, and branching of the neurites in a 3D substrate, and formation, elimination, long-term plasticity of the synapses during simulation	Simulation speed (minimisation of latencies in dendritic loops and spike transmission)
Possibility of creation of small-world and fee-scale networks	Reconfiguration speed (minimisation of reconfiguration latencies and overheads)
Possibility of cell isolation and apoptosis	Scalability to larger chips and multi-chips (performance, efficiency, reliability and complexity must not be impacted significantly by the size of the cortex)
Cortex model must be locally controlled and organised by concurrent evo-devo processes and also provide local feedback information about network activity (soma, synapse, and neurites) and routing resource availability	Manageable (modular and structured) and scalable complexity (by adding or removing extra features)
Communication with outside world (stimuli and response)	Accessibility of the internal signals for testing and debugging
Reliable and low-jitter delays for axons proportional to their length	Robustness, fault-tolerance (by redundancy, parallelism, and distribution)
Possibility of long-range and global signals for network activity regulation and learning	

3. Reconfiguration of the parameters and network connectivity (by concurrent developmental processes)

4. Providing local feedback information to developmental processes

Each one of these functions, and possible general options regarding each one are discussed here in detail.

5.2.1 Intracellular Communication

The intracellular communication, between synapse units of a neuron and its soma unit through a dendritic loop is mainly dictated by the Digital Neuron model that is used as the basis for cortex model design investigation. For achieving bio-plausibility and fault-tolerance and using the potentials of the FPGA architecture, synapse units must be distributed over the area of the FPGA rather than concentrating them in one region.

The intracellular communication infrastructure of the cortex model can be seen as an interconnected network [294] of PEs (Processing Elements, here, soma and synapse units) sending packets of neuron state variables (only membrane potentials in this case). From this point of view, a shared media network or a switched media network can be considered. Sharing a medium between different PEs (*e.g.* a bus) with different packet timings requires a very complicated arbitration mechanism and with the high traffic of the packets between soma and synapses it will lead to very high packet latencies. This significantly degrades the performance of the Digital Neuron model as it is highly sensitive to the packet latency in the dendritic loop. The other option is to use a switched media network. In a switched media network, a network fabric of links and switch components is used to convey messages between PEs. Such a network must implement three main functions of routing, arbitration, and switching.

Evo-devo processes can be responsible for routing the packets along with allocating the PEs (soma and synapse units). Another option is to let the developmental process specify the dendrite structure in a virtual 3D space that is then mapped to the physical network and routed by a separate process in the cortex model. This effectively decouples the evo-devo processes from resource management, the physical shape of the dendrite, and proximity of the synapses, which not only reduces the bio-plausibility of the system, but also adds to the hardware resources needed in cortex model for routing. Therefore, here, we focus on a bio-plausible approach of leaving routing to evo-devo processes. Chapter 6 will discuss the evo-devo processes that perform the routing and reconfiguration of the cortex. Intracellular signals need to be reconfigurable for a flexible and dynamic dendrite model. Therefore switching hardware resources are needed that allow growth and branching of the dendrites. This way, synapse units and dendritic routing resources can be shared between neurons and developmental processes can assign them dynamically to different neurons instead of preallocating a fix number of synapses and a fixed dendritic tree for each neuron.

Different switching techniques for Networks On Chips (NOCs) are discussed in [294, 188]: circuit switching, packet switching, and cut-through switching. In circuit switching, a circuit is established by a probing packet following the route before any payload packets are sent. All the links on the path will

be reserved for that circuit until all the packets are passed. As the routing is carried out and repeated by the developmental processes during simulation, this effectively creates a set of configured circuits that have deflected each others, depending on which one was first stablished. With dendrite packets being send continuously, once a circuit is established it needs be kept until the dendrite is reconfigured. This mechanism leads to a subset of circuit switching called configured switching [188].

For fairly short dendritic loops that are continuously used, configured switching can provide the best performance and efficiency as there are no latency or hardware cost overheads involved. However, it requires the circuits to be freed (by developmental processes) when they are not needed anymore. This is in agreement with a bio-plausible model in which developmental processes are responsible for deallocating redundant synapses and retracting dendrite subtrees.

For longer dendritic loops, the bandwidth of the links in the circuit will be under-utilised. Packet switching allows the bandwidth of the links to be shared and allocated more efficiently when packets are not sent continuously. This is apparently not the case in the Digital Neuron model dendrites. Moreover, packet switching stores a packet at each node and then forwards it to the next node based on the routing information in the packet, which dramatically increases the hardware cost and the latency of each node. The third option is cut-through switching. This technique and its variants (virtual cut-through and wormhole switching) reduce the hardware cost and latency of the packet switching by only storing part of the message that is required for switching the next hop. However, in this technique, different packets can block each other that again can cause long delays and even deadlocks that is not tolerable for the Digital Neuron model.

Therefore, it is reasonable to focus on configured switching as a promising bio-plausible and efficient solution for intracellular communication network of dendrites. Additionally, by leaving the routing of a configured switched network to developmental processes, the arbitration process can be also absorbed into the routing functionality, which reduces the hardware cost of each node.

Reducing the distance between synapse units and using shorter wires for dendritic loop signals reduces both the hardware cost and also improves the simulation performance. Therefore, it is better to use local FPGA routing resources for these signals rather than global or long-range wires. This points to a locally connected network topology. The bio-plausibility requires this topology to be 3-dimensional and wrapped around in one dimension. Different topologies for the cortex model substrate are investigated later in this chapter.

5.2.2 Intercellular Communication

Another function of the cortex is to implement the spike transmission between presynaptic soma units and synapse units. This, again, can be regarded as a communication network. In contrast to the intracellular network with a continuous flow of membrane update packet, spikes are sent intermittently and less often. Assuming one membrane potential update for simulation of one millisecond of neuron activity, a maximum spiking frequency of 200Hz shows that spike packets are at least 5 times less frequent than dendritic update packets. Unlike intracellular communication that is unicast (from one PE to the other), intercellular communication needs to be multicast, as one presynaptic neuron can send spikes to may

postsynaptic neurons. Additionally, spike packets does not need to convey any extra information other than their source identity and their timing (which is implied in the existence of each spike packet).

A shared media network such as a shared bus is not an option as the number of nodes and number of spikes that must travel through the network at the same time are quite high and the long and variable latencies and hardware cost of arbitration in shared media networks are not justified here [188, 294].

Using a switched media network requires routing, arbitration and switching functions. Similar to intracellular communication, to follow a bio-plausible approach, evo-devo processes must be responsible for the routing of spike packets. Alternatively, as in [275, 301, 72, 300, 378], the physical and logical network connectivity can be decoupled, which effectively deprives the system from the role that evo-devo processes can play in the resource management and optimisation of the neural microcircuit. Allowing evo-devo system to control the physical routing of the neurites enables it to exploit all the physical and even unwanted properties of the cortex substrate to optimise the functionality of the neural microcircuit. It also removes the burden of routing and arbitration from network nodes that significantly reduces the node latency and hardware cost. Moreover, [188] reported that offline scheduling of switches can yield up to 63% performance increase over online scheduling.

With the routing being performed by the evo-devo processes (discussed in chapter 6), three different main switching techniques are available: packet switching, cut-through switching, and circuit switching. These methods have different throughput, bandwidth, hardware cost and latency trends. Unlike intracellular network packets, spikes packets have more time for delivery. Performance of the system is also less sensitive to the latency of the spike packets. The spikes in biological brains can have a delay of up to 20ms depending on their length and other factors. Assuming a simulation resolution of one membrane update for equivalent of 1ms time of biological neuron activity, and an average of 50 clock cycles for each update, spike packets in the cortex model need to be delivered in 50-1000 clock cycles depending on their length. However, the spike packets latencies need to be reliable and accurate enough. For the spikes to have the same resolution of 1ms, their timing accuracy must be ± 25 clock cycles.

Packet switching has a high latency compared to the other two as it requires buffering the whole packet to decode the routing information before forwarding it to the next node. While packet-switching works perfectly in some large-scale applications such as [300], where real-time simulation is intended and very short packets are used, it is not a feasible solution for hyper-realtime simulations such as this. Even with 256 neurons, a spike packet needs to be at least 8 bits long and it can pass through a maximum of 125 hops in 1000 clock cycles. With different packets blocking each other and multi-casting of the packets to different destinations adding to the network traffic, this latency will be fluctuating and far from guaranteed.

Considering cut-through switching, as the whole spike packet is needed for decoding the routing information, each packet will consist of only one flit (flow unit [294]). Therefore, in this case cut-through switching will be equal to the packet switching in practice. Packet switching and cut-through switching methods also require buffers, and logic for flow control and arbitration that further add to the latency and hardware cost of each node. The hardware cost and unreliable latency of packet switching and

cut-through switching techniques render them infeasible for this intercellular communication network.

Considering circuit switching, apart from the initial latency for establishing a circuit, it has the best latency among switching techniques. With routing and arbitration already carried out by evo-devo processes, circuit switching turns into configured switching [188]. Configured switching has the minimum hardware cost per node as it does not need any buffers or logic for routing and arbitration. While configured switching provides a possible solution, it can wastes the bandwidth of the links by preallocating them to circuits that are very rarely used (for sending a spike packet). Additionally, by dedicating a circuit to a single axon, there will be no need for explicitly sending the pre-synaptic neuron ID, and the spike packet size will be reduced to only one bit. Taking this into account, degrades the channel utilisation of the configured switching even further by a factor of $\log_2(n)$, where n is the number of neurons.

Fortunately, it is possible to use time-multiplexing to utilise the bandwidth of the links more efficiently. In time-multiplexed switching each switch in the network follows its own predefined schedule on a time-division basis. Based on the length of the repeating schedule (number of contexts, n) this creates n virtual channels on each physical link between two nodes. However, this techniques requires that the switching schedule on every cycle to be locally stored (or somehow be available) in each node. The switching memory hardware cost for each port is of order $\mathcal{O}(n \log_2(m))$ where m is the number of ports in each node.

Table 5.2: Summary of different design patterns for implementing communication in FPGAs, and their characteristics and trade-offs (adopted from [188]).

Characteristics	Configured Switching	Time-Multiplexed Switching	Packet Switching	Circuit Switching
Communication predictability	High		Low	
Latency	Lowest	Low	Highest	Moderate
Switching logic HW cost	Low	Low	High	High
Switching memory HW cost	Lowest	Low	Highest	Modest
Comm. throughput-physical link bandwidth ratio	Highest	High	Low	Lowest
Channel utilisation (Application dependent)	Depends on app.		Low	
Latency overhead (Message length dependent)	Lowest	Low	Highest	Depends

Kapre *et. al.* have investigated the hardware costs, latency and trade-offs of time-multiplexed switching versus packet switching networks for FPGAs in [188]. Table 5.2 summarises their evalua-

tion of four different design patterns for implementing communication in FPGAs: configured switching, time-multiplexed switching, packet switching, and circuit switching. Clearly, with the lowest hardware cost and latency, and highest throughput, configured switching is the best option when application needs to use a circuit all the time. This is the case for intracellular communication in short dendritic loops. However, if application is using the network sporadically, configured switching is not the best options. Then, if communication predictability is important, time-multiplexed switching will be the best option as circuit and packet switching can not provide that predictability. This is clearly the case for intercellular communication in a hyper-realtime neural microcircuit application. However, in realtime neural applications, such as [300] and [147], packet switching makes much more sense particularly when packets are very small and number of PEs is very large [188]. Circuit switching is only an option when application needs to send very long messages sporadically and the latency overhead compared to the length of the message is negligible.

Utilising time-multiplexed switching for the intercellular communication network not only provides a solution to use the routing resources in FPGA efficiently, but also, as will be explained in section 5.2.2, it extends a 2D interconnection network, that is feasible on an FPGA, to a 3D virtual intercellular network, that is bio-plausible. Using such time-multiplexed communication network can also increase the scalability of the system to multiple FPGAs [235]. If all the physical links are local, as a bio-plausible approach suggests, it will be possible to run a time-multiplexed network at much higher clock frequency than rest of the system and increase its bandwidth even further.

Topology

Before moving to discussion of the reconfiguration and feedback functions of the cortex model, topology of the intercellular and intracellular networks must be discussed, so that the next sections can focus on particular promising topologies. From bio-plausibility point of view, although biological neurons and their projections are embedded in a 3-dimensional substrate, the fractal dimension of the connectivity of the neurons in *C. elegans* and the human brain are measured at around 4 [19]. This is indicative of much higher dimensional topology in these nervous systems. However, looking at the local interactions underlying these long range connectivity, they are still all local interactions with neighbouring elements in a 3-dimensional space. This 3D space is wrapped around in one dimension and has connections to the outside world at one of its edges, since a brain is modelled as a layered neural tube connected at the root to the rest of the body. This 3D space must provide enough resources and connectivity that supports networks with small-world and free-scale characteristics.

In terms of feasibility, the topology must provide low and reliable communication latency and enough throughput with the minimum hardware cost. To achieve this, it is important to appreciate the PEs vs. interconnection trade-off and find the balance between the amount of hardware resources dedicated to computation versus communication. A modular and structured topology is preferred for its reduced and manageable design and testing complexity. Reliability, robustness and fault tolerance are other feasibility factors related to the topologies of these networks.

Figure 5.1 shows different common topologies studied in the context of NOCs (Network-On-Chip)

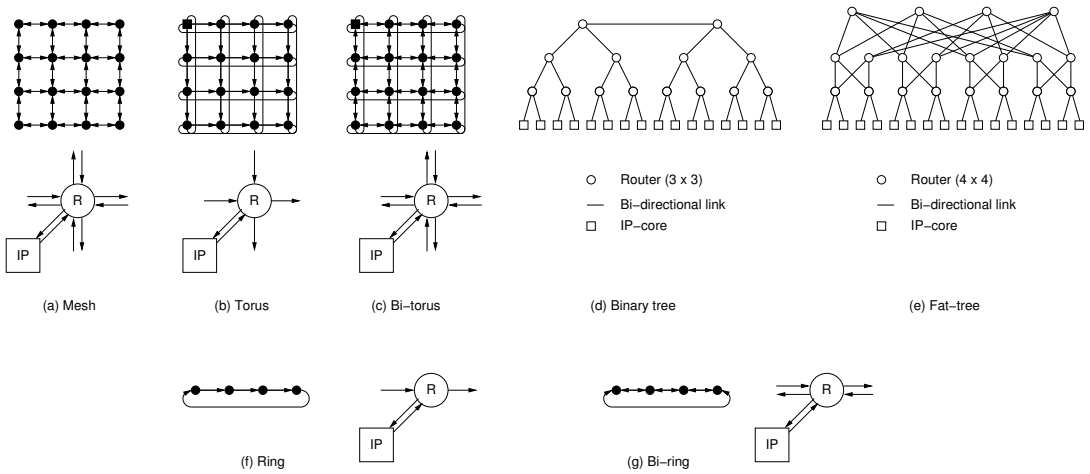


Figure 5.1: Common NOC topologies along with their router(R) and PE (IP) connections and ports (From [328]).

[294]. Inadequacy of bus (star) topology is already discussed in sections 5.2.1 and 5.2.2. A very straightforward topology (not shown in the figure) is a fully-connected graph that usually uses a single switch for connecting all the nodes centrally. Since the hardware cost of the switching is of order $\mathcal{O}(n^2)$, where n is number of ports (equal to number of PEs here), total switching hardware cost in a fully-connected network is only justifiable for small number of PEs. Moreover such a topology does not represent the locality required as a bio-plausibility factor.

Table 5.3 shows characteristics and hardware-performance trade-offs in different common NOC topologies . Bisection bandwidth is a measure of total performance of the network in terms of throughput. Maximum and average hop counts show the upper bound and typical number of hops that a packet needs to travel, which is directly related to the total latency. These two are the main performance factors of a topology. Switching hardware cost comes from the total number of switches in the network times hardware cost of each switch (number of ports squared). Number of links represent the hardware cost of physical links including the links between PEs and their corresponding switches.

Between these common topologies used in the NOC context, bus and fully-connected topologies can be rejected straightaway for very low bisection bandwidth and very high hardware costs respectively. Among the rest of the topologies, hyper-cube and fat-tree topologies have very good performances for a hardware cost that grows rather rapidly with the number of PEs. They also need many long-range connections, when embedded in a 2D substrate of an FPGA. Long-range wires in FPGAs are scarce and costly, and cause long delays that lead to a lower clock frequency impacting the overall performance of the network. Ring, 2D mesh and 2D torus are the only topologies that can be simply implemented on a 2D silicon chip using only short, local, and high-performance links. Therefore, 2D Mesh and torus topologies are two of the very popular topologies in NOCs. Ring can be seen as a 1-dimensional version of the torus. Higher dimensional versions of the mesh and torus are also conceivable. However, they have the same problem of long-range links when mapped to a 2D FPGA. It is also possible to have a hybrid of ring and mesh or torus by dividing each link into segments and adding more nodes in between.

Table 5.3: Characteristics and hardware-performance trade-offs in different major NOC topologies where n is number of PEs [294]. Bisection bandwidth represent the total bandwidth of the network in unit of link.

Topology	Performance		Hardware cost		Locality (2D mapping)
	Bisection Bandwidth	**Max** (ave.) hop count	Switching HW cost	Number of links	
Bus (star)	$\mathcal{O}(1)$	$\mathbf{1}$ (1)	$\mathcal{O}(n)$	n	No
Ring	$\mathcal{O}(2)$	$\mathbf{n/2}$ $(n/4)$	$\mathcal{O}(9n)$	$2n$	**Yes**
2D Mesh	$\mathcal{O}(\sqrt{n})$	$\mathbf{2\sqrt{n}-2}$ $(\sqrt{n}-1)$	$\mathcal{O}(25n-36\sqrt{n})$	$3n-2\sqrt{n}$	**Yes**
2D Torus	$\mathcal{O}(2\sqrt{n})$	$\mathbf{\sqrt{n}/2}$ $(\sqrt{n}/4)$	$\mathcal{O}(25n)$	$3n$	**Yes**
Hyper-cube	$\mathcal{O}(n/2)$	$\mathbf{\sqrt{n}-\log_2(n)}$ $(\log_2(n)/2)$	$\mathcal{O}(n(\log_2(n)+1)^2)$	$n\log_2(n)/2+n$	No
Fat-tree	$\mathcal{O}(n/2)$	$\mathcal{O}(>Mesh, <Ring)$	$\mathcal{O}(2kn\log_{k/2}(n))$	$\mathcal{O}(n\log_{k/2}(n))$	No
Fully-conn.	$\mathcal{O}(n^2)$	$\mathbf{1}$ (1)	$\mathcal{O}(n^2)$	n^2+n	No

This leads to a heterogeneous networks with two type of switches (5 and 3-port) and slightly increases the PE-interconnection ratio, which can be used to adjust the ratio for best overall hardware cost and performance. It is also possible to do the reverse and increase the number of local links to 6 or 8 as in [275], which practically decreases the PE-interconnection ratio. Schoeberl *et. al.* have investigated different topologies for time-multiplexed NOCs on FPGAs in [328] and reported that for networks above 16 nodes, only torus and fat trees have enough link capacity to enable a schedule period that is in the same range as the IO capacity of the IP cores. With respect to the local connectivity pattern of the FPGA CLBs, a 2D grid torus with 4-neighbourhood connectivity appears as a simple and efficient option that can be extended to 6 or 8 neighbours since each Virtex-5 CLB has 1-hop (low-latency) connectivity wires to all 8 neighbouring CLBs. Selection of the best neighbourhood connectivity and cell design is a separate subject that needs mush further investigation with comprehensive simulations or analytical study (see [92] for example).

Although a 2D torus appears to be the best feasible topology for intercellular and intracellular communication networks of the cortex model, it does not map perfectly with the 3D substrate needed for bio-plausible neural microcircuits. Fortunately, time-multiplexing a 2D topology can create a virtual third dimension in time axis that allows a better mapping to a bio-plausible 3D substrate. This has been already proposed in previous section for intercellular communication network. However, due to the asynchrony of soma units and timing of their packets in dendritic loops, it is not possible to use time-multiplexing for intracellular communication network and extend the growing substrate of dendrites to three dimensions.

Figure 5.2 from [328] depicts the general circuitry for a 2D mesh or torus time-multiplexed switched network. Each switch is shown as a multiplexer receiving inputs from north (N), south (S), west (W), east (E), and the local PE (L). The scheduled switching data for selecting inputs for each multiplexer come from a Schedule Table (ST) that is addressed sequentially by a time-slot counter that can be local to each node or global. This counter generates the slot numbers from zero up to the length of the schedule period. The main hardware cost overhead in this method is the memory needed for the schedule tables. A m-port switch needs a total of $mn \log_2(m-1)$ bits of RAM, where n is the length of the schedule. If a global time-slot counter is used, $\log_2(n)$ global signals are also needed to be connected to all switches. Otherwise each switch or group of switches need a local counter of complexity $\mathcal{O}(\log_2(n))$.

5.2.3 Reconfiguration

To evaluate the fitness of each individual, evo-devo processes must be able to modify the parameters and connectivity of the neurons and synapses both for setting up the neural microcircuits and for successive modifications during development. This process is called reconfiguration of the cortex, although it may not necessarily entail using the reconfiguration feature of the FPGA.

Regarding the bio-plausibility of the cortex, reconfiguration must allow localised modifications of the parameters and connectivity of the neurons, neurites and synapses. The cortex model must also allow the density and location of the soma and synapse units to be controlled by the evo-devo processes. From a feasibility point of view, the cortex reconfiguration must introduce the minimum overhead on the hardware cost of the cortex model and performance of the simulation and reconfiguration processes. Since the cortex needs to be reconfigured at least once, and many times in case of activity-dependent development, for evaluation of each individual during evolution, the reconfiguration overhead directly affects the performance of the whole system. This can be due to long reconfiguration times or because simulation may need to be stopped during cortex reconfiguration.

Virtex-5 FPGA supports different ways for reconfiguring the device. JTAG and SelectMAP are serial and parallel modes of externally reconfiguring the device. Internally, there are two identical Internal Configuration Access Ports (ICAP). These are very similar to the SelectMAP port available externally to the FPGA user but they are available to be used by the internal circuit of the FPGA to partially reconfigure itself at the highest possible speed.

The type of memory elements that are used for storing the configuration of the cortex parameters and connectivity has an impact on both reconfiguration time, need for pausing the simulation, and hardware cost of reconfigurability of the cortex. Different types of memory elements available on the Virtex-5 family of FPGAs and their use as a reconfigurable element are discussed here [408, 411].

Flip-flops and Latches

Storage elements are the simplest type of memory primitives in Virtex-5 FPGAs available to the user that can be configured as edge-triggered flip-flops (FF primitive) or level-sensitive latches (LATCH primitive). Their states can be modified by pushing new data in the storage element. They can be also reset to an initial state (specified at FPGA configuration time) with a signal global to the slice. The density of these storage elements is quite low (only 4 elements per slice) compared to the other possible memories

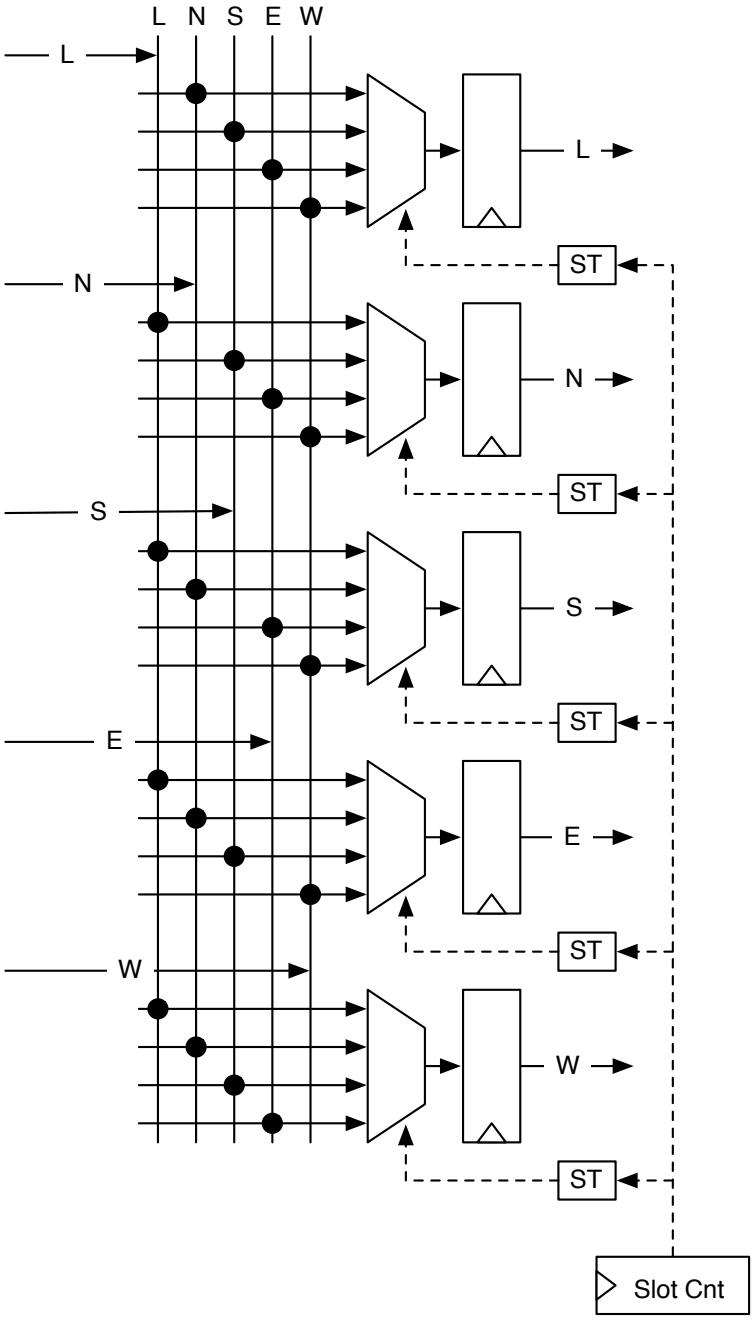


Figure 5.2: The circuit to be added to each PE for time-multiplexed switching of a 2D mesh network (From [328]). L, N, S, E, W respectively present links to and from Local PE, North, South, East, and West nodes. ST represent a Scheduling Table.

in the slice. They are usually used for storing the state of the sequential logic circuits. They can be used as a reconfigurable element by directly feeding the data in them at run-time or by reconfiguring the FPGA. It is also possible to specify the set or reset initial value of these elements by reconfiguring the FPGA and then asserting the set or reset signals of the slice (as clock and these signals are shared over one slice). Although these primitives can be used to store parameters (and connectivity if connected to a multiplexer) but the low density of these memory elements in the slice makes them a scarce resource that is better to be used for sequential logic and control rather than reconfigurable memory.

Lookup Tables (LUTs) and ROMs

Each slice in Virtex-5 contains four LookUp Tables (LUT). Each LUT has six inputs and two outputs and can be configured as two 5-input LUTs (with the same set of inputs), a 6-input LUT with one output, or equivalently a 64-bit ROM. Content of these LUT primitives can be only modified through FPGA reconfiguration. These are the most abundant reconfigurable resources in FPGA CLBs, which can be used to store parameters and connectivity (if configured as a multiplexer). They are only reconfigurable through the global FPGA reconfiguration process, and there is no way for a local process (such as developmental processes) to reconfigure these primitives.

Distributed RAMs

LUTs in Virtex-5 SLICEMs (left side slice of every other CLB, 1 in every 4) can be also reconfigured as a 64-bit single port RAM. It is also possible to join two of these LUT primitives in the same slice to create a 64-bit dual port RAM as SLICEM RAM primitives have separate read and write address inputs. Other mixed combinations of single and dual port RAMs with more outputs or capacity is possible by combining four RAM primitives in a SLICEM. The contents of these distributed RAM primitives can be modified directly by feeding synchronous data into them (asserting Write Enable input and feeding the address and clock) at run-time, or through FPGA reconfiguration. After LUTs, these are the second most abundant resource available for reconfiguration in CLBs. The fact that they can be reconfigured by writing data directly to them makes it possible for a local developmental process to reconfigure these elements.

Shift Registers

LUTs in Virtex-5 SLICEMs can be also configured as 32-bit shift registers (SRL32). They provide a shift-in input (DI1), and a multiplexed output that can be selected by address inputs to provide the state of any of the 32 bit values in the register at any time. A shift output (Q31) is also available inside the slice that can be configured to be cascaded to the input of other shift registers to make longer shift registers. However, this signal in not available outside of the slices for three shift registers out of four in one slice. The content of these primitives can be directly modified by shifting data in at run-time or by FPGA reconfiguration. These are as abundant as distributed RAMs. In fact they are essentially the same primitive that is configured to behave slightly differently. However, they offer half of the capacity when used as a shift register. Nevertheless, the fact that they can be configured serially by a local process without addressing makes them a very efficient option for both storing the parameters and connectivity.

Block RAMs

Block RAMs are modules of 18Kbits of dual-port RAM that can be configured as 32K, 16K, 8K, 4K, 2K and 1024 words RAM and FIFO modules with 1,2,4,9,18, and 36-bit width respectively. Different Virtex-5 devices have different number of Block RAMs, which is proportional to the size of the device. A 50K logic element FPGA such as XC5VLX50T, which is used in this study, has 60 36KBit Block RAMs providing a total of 2160KBit RAM. This is 4.5 times the total distributed RAM available in all the SLICEMs. However, these Block RAMs are condensed in a few columns in the FPGA and are not distributed evenly over the whole silicon area. Apart from writing directly to Block RAMs at run-time, it is possible to specify their initial content through FPGA reconfiguration.

Programmable Interface Points (PIPs)

One of the most abundant reconfigurable resources in the FPGAs is part of the reconfiguration memory that controls the Programmable Interface Points (PIPs) in the switch boxes used for the interconnection of the logic resources. These switch boxes are only reconfigurable through the FPGA reconfiguration process and are not accessible directly to the user. However, if they can be somehow used for storing the connectivity (and parameters when used as a register), the hardware cost can be reduces significantly. Since these are not supposed to be available to the user for direct partial reconfiguration, Xilinx does not suggest a method for modifying connectivity directly at such a low level. The only way they can be partially reconfigured at run-time is using a difference-based partial reconfiguration workflow. This work flow requires that the connectivity of the part of the circuit to be modified using a Xilinx tool such as FPGA editor and then saved. Then another Xilinx tool (Bitgen) can be used to create a difference-based reconfiguration bitstream from the two versions of the circuit. Bergeron *et. al.* in [29] proposed a method specifically for low-level reconfiguration of PIPs in Virtex-II FPGAs[29]. Direct generation or manipulation of the bitstreams would not be possible without knowing the complete bitstream format. Although the general format of the Virtex-5 configuration bitstream is publicly available [411] but the detail of the data format in each frame is proprietary and not released. However, it would be possible to reverse engineer and use that information as shown in [83, 28, 279]. However, using PIPs in a DPR workflow must be thoroughly tested as glitches may affect the functionality of the circuit or contentions damage the device permanently.

Dynamic Partial Reconfiguration vs. Virtual FPGA

In Virtex-5, reconfigurable elements and user storage elements can be read or written through reconfiguration process. Both full and partial reconfiguration (and read-back) is possible [411]. Virtex-5 also allows user to dynamically reconfigure the device modifying the reconfiguration of part of the FPGA when the rest of the device is working normally (DPR or Dynamic Partial Reconfiguration). Event the reconfigured region may continue running as in many cases (such as LUT content modifications) there is no glitches in the transition. However, the smallest readable or writable unit of information through reconfiguration process is one frame. A frame in Virtex-5 is part of the reconfiguration information that spans across a column of 20 CLBs. This is particularly restrictive in partial reconfiguration as write operations to the frames that span over some storage elements (such as FFs, RAMs, or SRLs) will corrupt

the content of these primitives. This is because the content of these elements can change in the period between reading a frame and writing it back with modification as the reconfiguration process does not support an "atomic" read-modify-write operation.

Two general types of dynamic reconfiguration of the cortex is conceivable. One is to use the normal dynamic partial reconfiguration of the FPGA [184]. This may allow the user to somehow access all the reconfigurable elements such as PIPs and ROMs (LUTs) that are not writable through the circuit itself. There are some limitations, requirements, and possibilities:

1. Access to details of the bitstream and frame format of the FPGA
2. Performing the reconfiguration centrally from outside of the FPGA or through one of the ICAPs
3. To avoid corruption of the neighbouring reconfigurable memory elements in the same column during reconfiguration, simulation must be paused and a frame must be first read, modified, and then written back, which adds another overhead to reconfiguration time.
4. Xilinx offers some C libraries for reading and writing the content of LUTs and FFs through ICAP that can be run on the MicroBlaze soft processor core connected to a XPSHWICAP IP core (both provided by Xilinx).
5. With some reverse engineering to discover the reconfiguration frame format, it would be possible to use PIPs as well.
6. This method offers relatively lower reconfiguration speeds as FPGA needs to sequentially receive all the reconfiguration frames padded with header and trailer data. Also partial reconfiguration of only a single bit require reconfiguration of a whole frame of 1312 bits. Moreover, content of a single LUT is segmented over four different frames.

The second method is to use the virtual FPGA approach and take provisions for the run-time reconfiguration of parameter and connectivity by allowing separate data path and logic for modification of the parametric and routing data. This has some advantages and some drawbacks too:

1. It allows a distributed, scalable, and even asynchronous reconfiguration process by local interactions at low level.
2. This method also offers much higher reconfiguration speed as it does not have the overheads of the FPGA reconfiguration.
3. It requires to dedicate extra hardware resources for reconfiguration of each parameter or switch.
4. It is only limited to Block RAM, RAM and SRL primitives that are four times less abundant than simple LUTs. Therefore, it is not possible to use PIPs as switches with this method.
5. All the RAMs and SRLs in the same SLICEM share the same WE (write enable) and clock signal, which is a limiting factor.

Relocatability

One of the bio-plausibility factors requires the evo-devo processes to specify the density and position of the neurons in the cortex. This requires a structure that allows neuron modules to be plugged in to the substrate anywhere in the middle of the cortex. Two conceivable methods are considered here: Plug-in method and module-based PR workflow.

One method would be to locate the neurons all around the cortex and then plug them in anywhere they are needed using long-range wires as they only have 3 ports (one axonal out put, one dendritic input and one dendritic output). This method has few drawbacks: First, long range wires introduce long delays that significantly impact the performance of the simulation as discussed earlier. Secondly, number of neurons will be fixed and limited. Even if some of the neurons are not plugged into the cortex, their hardware resources is not used in any other way, which impacts the efficiency of the cortex. Finally, the long wires are really scarce in the FPGA and there might not be enough wires to plug neurones in a flexible manner and therefore not only neuron relocation but also the density control requirement of the cortex can not be fully addressed. This method can be used both in conjunction with the virtual FPGA reconfiguration method or with normal FPGA reconfiguration process.

A second method is to have a modular 2D grid structure for the cortex that provides the infrastructure for inter and intracellular communication networks and synapse formation and use a module-based dynamic partial reconfiguration workflow to replace some of the modules with relocatable neuron modules. This requires exact matching of the input-output ports of the modules. To address this, Xilinx proposes an intermediate static circuit called Bus Macro that takes one CLB (2 slices) of the FPGA and provides 16 input or output lines at the edge of the partially configurable module. It also requires that the underlying FPGA resources exactly match with the resources of the original location of the module. As Virtex-5 and many other new FPGAs are quite heterogeneous, the modules placed, and routed for one region (*e.g.* one column or top half) of the FPGA may not be compatible with another region of the FPGA. There are solutions to these problems that reduces the hardware cost of relocatability of the neuron modules and reduces the number of different modules for different regions of the FPGA [20]. In [357] Strunk *et. al.* suggest a detailed approach for such modular grid structure for a similar fine-grain parallel processing network. However, the hardware cost overhead of adding 2 slices for a Bus Macro to each node is simply not justifiable, when one synapse fits in less than one slice, a soma unit can be implemented in about three slices, and each LUT can support up to two 5-port switches.

5.2.4 Feedback

Developmental processes need to receive information about the activity and performance of each part of the cortex. Neurons activity and health are the very basic information that can be fed back to developmental process. In an activity-dependent neurodevelopment process, unused synapses where an axon and a dendrite cross each other can feedback information about their potential to be connected. Used synapses can also feedback data about their redundancy. Every other piece of the communication network can also feedback data about the congestion and activity at that point.

Each neuron in the Digital Neuron model is emitting spikes through the axonal output and sending

membrane potential packets out of its dendritic output. The pulses on these two outputs can be used to evaluate the health and activity of the neuron. As it is not necessary to measure the activity of the neuron with high resolution, it is possible to use a simple circuit consisting of a shift register (or a RAM that is continuously addressed by random numbers) to keep track of the activity level of a neuron. Figure 5.3 shows an example stochastic circuit using a shift register that can measure the activity of the neuron on a scale between 0 and 32 over a time period. The pulse width of a global measurement window signal can be adjusted so that very low activity is measured as zero or a few 1s in the stochastic bitstream, and too much activity saturates the stochastic value. The output of this circuit can be used in a stochastic developmental system for regulation of the neurons activities by evo-devo processes. For a synapse, two most significant bits of the membrane potential packet can be sampled when the synapse has received a presynaptic spike and combined in a similar circuit for a very rough stochastic Hebbian output that can be both used for local unsupervised learning in the synapse or to feedback potentiality or redundancy of the synapse to developmental processes that regulated both connectivity and efficacy of the synapse. Similar designs can be used for outputs of the switches in the communication networks to measure the activity and network congestion to provide developmental process with more information during simulation.

These example measurement circuits do not pose any performance overhead on the cortex model and given the benefits of optimising the microcircuits by evo-devo processes may increase total performance of the system. However, they slightly add to the hardware cost. Each one of these feedback features can be separately added to or removed from the design, that offers a controllable level of complexity to the design, which is one of the feasibility factors of the cortex model.

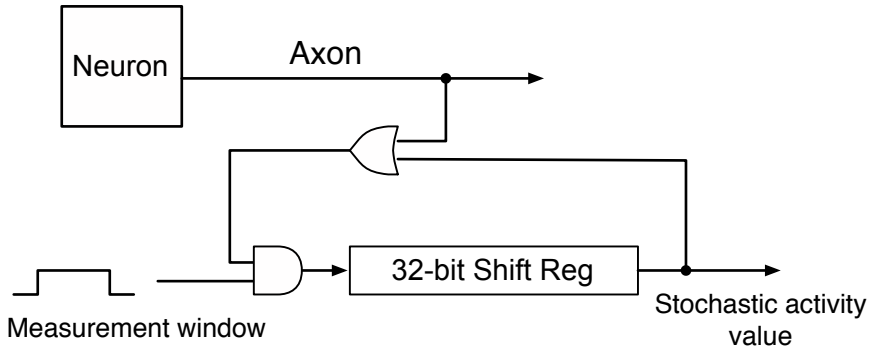


Figure 5.3: An example of a circuit that can be used to gather stochastic measurements of the activity of a neuron over a measurement period.

5.3 Summary of Design Options

Here we summarise different options and approaches in the cortex model design, their challenges, major factors, constraints, and trade-offs. We assumed a minimum required level of bio-plausibility and feasibility to focus the exploration on promising cortex models and prepare for the design of a case study cortex model.

Bio-plausibility

Different approaches, design options and factors that affect the bio-plausibility of the cortex can be summarised as:

1. Using a virtual 3D topology for intercellular communication network adds to the bio-plausibility of the cortex model.
2. Using a 2D intracellular communication network instead of a 3D network topology reduces the bio-plausibility.
3. Routing by evo-devo processes adds to the bio-plausibility of the cortex.
4. Using a mesh or torus topology with local connectivity is more bio-plausible than hyper-cube or fat-tree topologies.
5. Having relocatable neuron modules on the cortex is more bio-plausible as it allows both density and position of the neurons to be controlled by evo-devo processes. However, employing a module-based partial reconfiguration workflow results in restrictions in neuron relocatability that impacts the bio-plausibility.
6. Adding circuits to cortex model to feedback data to developmental processes enables activity-dependent development and provides evo-devo processes with useful information about the activity, health and performance of the neural system and its underlying networks, which can effectively add to the evolvability and bio-plausibility of the cortex model in general.

Bio-plausibility-Performance Trade-offs

Special requirements of the intracellular communication in the Digital Neuron model can be only addressed by a configured switched network and a 3D topology of such network can not be implemented on a 2D FPGA efficiently, which creates a trade-off between performance and bio-plausibility. The virtual FPGA method of cortex reconfiguration is both faster and more bio-plausible as it uses distributed and local mechanisms for reconfiguration. Plug-in approach, with a fixed number of neuron modules around the cortex, needs long wires that impacts the performance of the cortex but it may provide some degree of neuron relocatability.

Compactness

Using a configured switched network creates a constraint on the compactness of the cortex model as it needs a minimum required resources. 2D or virtual 2D (time-multiplexed switching) intercellular communication networks are more compact than 3D networks.

Bio-plausibility-Compactness Trade-offs

Relocatability of the neurons with the module-based PR workflow (using Bus Macros) adds significant overheads to the hardware cost of the cortex substrate. Using virtual FPGA method leads to a more bio-plausible cortex model but significantly increases the hardware cost of the cortex. The interconnection-PE ratio in intra and intercellular communication networks is an important factor that can affect the

bio-plausibility of the cortex as very low connectivity may reduce the possibility of networks with the right characteristics. Using topologies with higher connectivity degrees add to the hardware cost.

Efficiency and Performance-Compactness Trade-offs

Allowing evo-devo processes to perform the networks routing may lead to better utilisation of the resources and improvement of both performance and compactness of the cortex, which effectively increases the efficiency. Providing feedback about available resources and network activity can intensify the effect. Using a 2D torus topology can improve both performance and compactness of the cortex model as well. A virtual FPGA reconfiguration method can significantly improves the reconfiguration speed but adds to the cortex hardware cost, which effectively creates a trade-off between performance and compactness.

Bio-plausibility-Efficiency Trade-offs

A virtual 3D topology for the intercellular communication network (using time-multiplexed switching) increases bio-plausibility, performance and compactness of the cortex model at the same time, which effectively relaxes some of the trade-off between bio-plausibility and efficiency.

Scalability

A low-connectivity configured switched network for intracellular communication can impact the scalability of the cortex model as it highly restricts the neurite growth to local regions around a neuron. This creates a trade-off between compactness and scalability. Using a time-multiplexed switched network for intercellular communication can improve the scalability of the system to larger devices and many FPGAs. Using virtual FPGA method for reconfiguration is more scalable than dynamic partial reconfiguration.

Reliability, Fault-tolerance, and Robustness

Using evo-devo processes for routing can in fact add to the fault-tolerance and robustness of the system. Using virtual FPGA reconfiguration method can be distributed, parallel, asynchronous, and redundant, which significantly adds to the reliability of the cortex in terms of fault-tolerance and robustness. Relocatability of the neurons can add to the fault-tolerance of the cortex.

Simplicity

Using dynamic partial reconfiguration process for configuring the cortex can significantly add to the design and testing complexity particularly if it requires reverse engineering the bitstream format and testing unofficial workflows and using PIPs, etc. Design of a cortex model with relocatable partial reconfigurable neuron modules is very challenging and involves complex testing procedures. A 2D grid-mesh or torus reduces the complexity of the cortex. It also allows any dendritic or axonal signal on the cortex substrate to be simply routed to the edge of the cortex and be probed to monitor the activity and membrane potential of the neurons in the cortex. This increases the observability of the cortex model, which simplifies the testing and debugging processes.

Table 5.4 summarises the above factors and trade-offs for major different options and methods of designing the cortex model. A brief look at the table shows clear advantage of using evo-devo pro-

Table 5.4: Summary of different factors and trade-offs for major competing options and design approaches. $+$, $-$ and $\sim$ show that employing a design approach or option can increase, decrease, or affect a factor respectively. Empty cells represent items where the analysis did not reveal a factor to depend on a design option. Major trade-offs are highlighted in blue, and clear win-win choices in green.

Competing design approaches and general options		Bio-plausibility	Performance	Compactness	Scalability	Reliability	Simplicity
Intracellular comm. network	3D	$+$	$-$	$-$	$-$	$++$	$-$
	2D (configured switching)	$-$	$+$	$-$	$+$	$+$	$+$
Intercellular communication network	2D	$-$	$+$	$+$	$-$	$+$	$+$
	3D	$++$	$-$	$-$	$-$	$++$	$-$
	Virtual 3D (time-multiplexed switching)	$+$	$+$	$+$	$+$	$+$	$-$
Routing	Using evo-devo processes	$+$	$+$	$+$		$+$	$+$
	Run-time routing using hardware	$-$	$-$	$-$		$-$	$-$
Topology	2D mesh	$+$	$+$	$++$	$+$	$+$	$+$
	2D torus	$++$	$+$	$+$	$+$	$+$	$+$
	Hyper-cube	$-$	$-$	$--$	$-$	$++$	$-$
	Fat-tree	$-$	$-$	$-$	$-$	$++$	$-$
Cortex substrate	High Interconnection-PE ratio	$+$	$\sim$	$\sim$	$+$	$+$	
	Low Interconnection-PE ratio	$-$	$\sim$	$\sim$	$-$	$-$	
Reconfiguration	Virtual FPGA	$+$	$+$	$-$	$+$	$+$	$+$
	Dynamic Partial Reconfiguration (DPR)	$-$	$-$	$+$	$-$	$-$	$-$
Neuron Relocatability	Fixed number and location	$-$	$+$	$+$	$+$	$-$	$++$
	Module-based PR workflow	$+$	$+$	$-$	$-$	$+$	$-$
	Plug-in approach (fixed number)	$+$	$-$	$+$	$+$	$-$	$+$
Feedback	Neuron activity and membrane potential	$+$		$-$		$+$	$-$
	Synapse potential and redundancy	$+$		$-$		$+$	$-$
	Network activity and congestion	$+$		$-$		$+$	$-$

cesses for routing, a 2D torus topology for the communication network, and virtual FPGA technique for reconfiguration. It also presents the major trade-offs between bio-plausibility and performance in the intracellular communication network design, bio-plausibility versus simplicity in intercellular communication network, and bio-plausibility versus efficiency in neuron relocatability. In the next section these general insights are used to design a cortex model as a case study.

5.4 Case Study: The Cortex

In this section an example cortex model is designed based on the investigation and analysis of the general options, approaches and design factors in the previous section. The intention is to investigate design and implementation challenges of a cortex model in practice, and create a cortex model that can be used in the next chapters as a basis for further investigation of the other aspects of this study. It also offers a flexible and extendible example cortex model that can be modified and used by other researchers or designers.

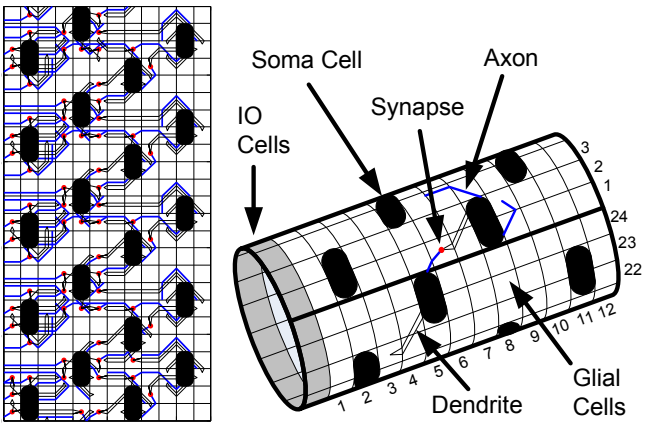
The general design of the cortex follows the analysis of the previous section in the context of investigation of challenges within the timeframe of this project. Therefore design decisions are made on the basis of exploring new areas for improvements rather than exploiting the available solutions. Moreover, due to the time restrictions of this study, simpler design approaches can be followed, particularly if it does not impair the generality of the study. Some of the trade-offs that are highlighted in the previous section will be further investigated in practice given the specifics of the case study.

General Choices and Trade-offs

In the case study, evo-devo processes will be used for routing since it is the bio-plausible winning option according to the analysis. Similarly a 2D torus will be used given the results of the analysis.

Although the virtual FPGA method for reconfiguration of the cortex leads to a more bio-plausible cortex model and faster reconfiguration for the price of higher hardware cost, dynamic partial reconfiguration method is used instead. This is mainly because the hardware cost overhead is very high. Not only extra circuit is needed to be added for supporting local reconfiguration, but also only a quarter of the FPGA resources can be used as reconfigurable elements. Furthermore, many bio-inspired models of evolvable hardware and neural networks in the literature already use Virtual FPGA method and it is fairly investigated and exploited. On the other hand, dynamic partial reconfiguration method is full of challenges yet to be discovered and tackled. Further investigation of dynamic partial reconfiguration method may lead to advances in relaxing the trade-offs. Moreover, this method results in a compact cortex model that is also compatible with the simulation of the developmental processes in software in the next phase.

The trade-off between bio-plausibility and performance of the intracellular communication network is decided based on the simplicity of the 2D configured switching method, and its scalability and relative reliability compared to a 3D network on a 2D chip. Similarly a less bio-plausible but simpler 2D intercellular network can be selected if it can be shown how to extend the case study design to a virtual 3D time-multiplexed switched network. This is an example of manageability of the design, which



(a) (b)

Figure 5.4: (a) A sample 12x24 cortex with 20 neurons. (b) The 2D cylindrical structure of the cortex.

allows a designer to balance a trade-off in the context of requirements and project timeframe. The bio-plausibility-efficiency trade-off in neuron relocatability is investigated further in section 5.4.5 to see if a new method can be found to break this trade-off.

5.4.1 General Architecture

Based on the 2D torus topology from the analysis, a cellular substrate for development of the neural microcircuits in the FPGA (called the Cortex) is proposed. The Cortex consists of a 2D grid of glial cells with neuron soma cells embedded in the middle of them. Here, "glial cells" refer to non-neuron cells that provide the means for routing dendrites and axons, and formation of synapses at their intersections. The grid is wrapped around like a cylinder to create a bio-plausible cell neighbourhood similar to the neural tube. A hexagonal mesh topology is also possible, thanks to the diagonal local connections between neighbouring CLBs in Virtex-5 and other new FPGAs. However, the limited resources of the FPGA logic blocks make a 2D grid simpler and more feasible. To keep the regularity of the cellular structure, it is desirable that soma and glial cells be of the same size. Nevertheless, as functionality of soma cells requires more hardware resources than glial cells, they are two times larger than glial cells and fit into two vertically adjacent grid cells. The vertical option is preferred as it minimises the signal delay between neighbouring cells on the actual chip (see section 5.4.5). A column (ring) of IO cells is also connected to the left side of the cortex that provides the interfacing with the outside world. Figure 5.4 shows a 12x24 Cortex with 20 neurons. Each glial cell receives an axonal and a dendritic input signal and has an axonal and a dendritic output on each side. Soma cells have six of those signals as they are in contact with six neighbouring glial cells.

5.4.2 Soma Cells

Each soma cell consists of a soma unit, six reconfigurable multiplexers and six pipeline D flip-flops (DFF). Reconfigurable multiplexers (from now on we refer to them as MUX) are essentially FPGA LUTs (look-up tables) that are dynamically reconfigured to work as many-to-one switch boxes. Using LUTs for this purpose makes it possible to investigate both (difference-based) dynamic partial reconfiguration

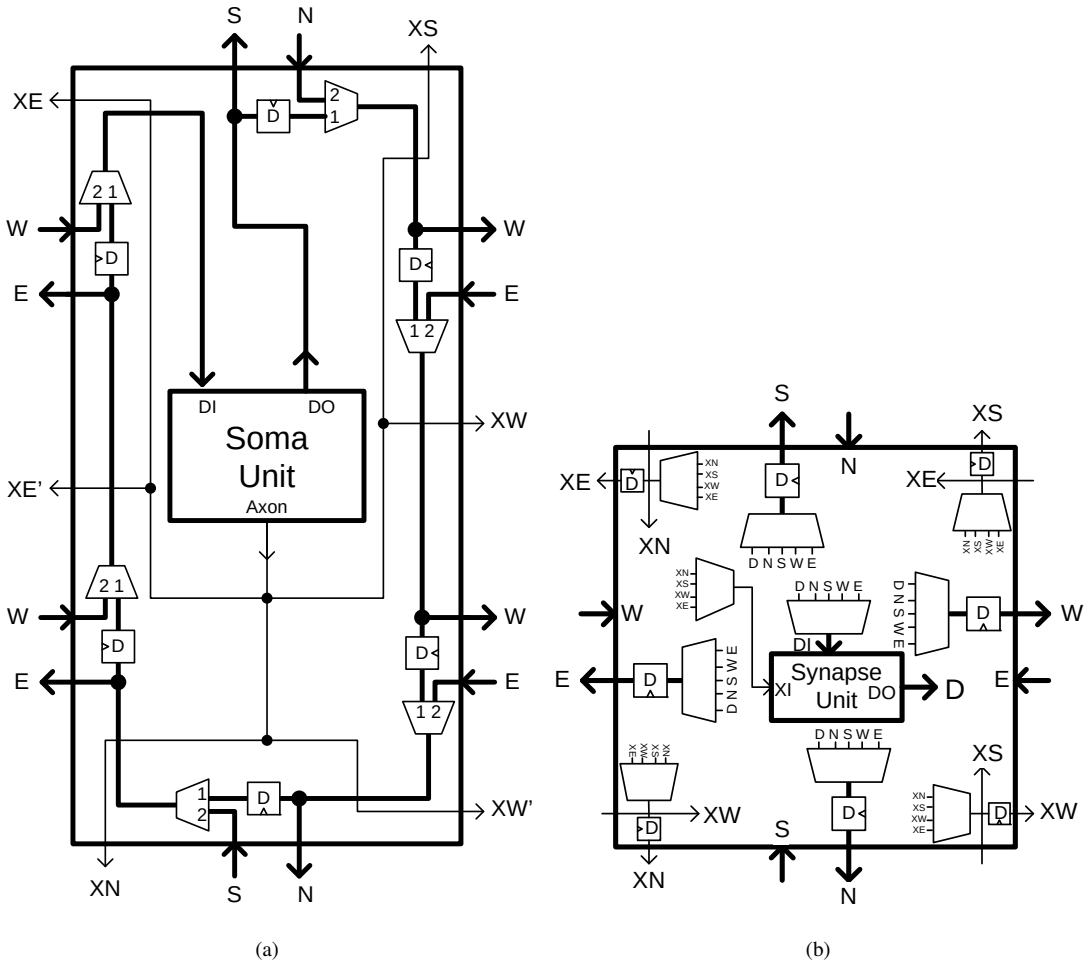


Figure 5.5: (a) Internal Architecture of soma cell. (b) Internal architecture of glial cell

[406] and Virtual FPGA method (using RAMs or SRLs instead). The internal architecture of the soma cell is shown in figure 5.5(a). The axon output of the soma unit is connected to all the six axonal outputs of the soma cell (XN, XS, XW, XW', XE, XE'). This way axons can project out of the soma cell in any direction before branching into branchlets, increasing the flexibility of the model. When there is no dendrite growth, DFFs and MUXs can form the dendritic loop right inside the soma cell by switching all MUXs to their first inputs. A soma cell can start growing a dendrite branch on any of its edges by switching the corresponding MUX to its second input. Therefore, a soma cell can project up to six dendrite branches directly from the cell body before any division into dendritic branchlets. This adds to the flexibility of the routing while resembles to dendrite growth of the biological neurons.

5.4.3 Glial Cells

Figure 5.5(b) shows the internal architecture of the glial cells. Each glial cell consists of a synapse unit, ten MUXs, and eight DFFs for routing axons and dendrites. On each side of a glial cell, there is one axonal output coming from a pipeline DFF connected to a MUX. Each axonal MUX can switch to any of four axonal inputs on the edges of the glial cell (XN, XS, XW, XE). This way, it is possible to route

up to four axons through a glial cell as explained later in the example of section 5.4.4.

A similar circuit can be employed for the dendrite routing. However, each MUX in the dendritic circuit has a fifth input, which is connected to the dendritic output of the synapse unit (DO). The dendritic input of the synapse unit (DI) comes from another MUX that can switch to any of the dendritic inputs on the edges of the glial cell (N, S, W, E). Therefore, the synapse unit can be inserted into any of the dendritic loops routed through the glial cell. The axonal input of the synapse unit can also be connected to any of the four axonal inputs of the glial cell using a 4-to-1 MUX. Therefore, it is possible to form a synapse between any dendrite and axon routed through a glial cell in three simple steps: 1. Copy the configuration of the corresponding dendritic MUX to the dendritic MUX of the synapse unit. 2. Switch the corresponding dendritic MUX to synapse dendritic output (DO). 3. Switch the axonal MUX of the synapse unit. Similarly, a reverse procedure can be use to eliminate a synapse. Although due to the latency of the synapse unit and its input MUX, these steps do not guarantee a glitch-less transition, the glitch can only corrupt one single bit. Worst-case scenario, is that the header bit of the packet gets corrupted and the whole packet is logically shifted right. In this case the neuron may go through a transient change and then return back to normal regime or it may enter into a tonic spiking regime depending on its parameters. Such transient Single Upset Events (SUE) are quite normal in neural systems and they must be designed (or evolved) to be robust to such input and internal noises. However, if the reconfiguration clock and cortex clock (feeding the pipeline DFFs) are the same, the transition will be glitch-less if all timing constraints of the design are met during implementation. This is thanks to the pipeline DFFs in the routing circuit that improve the clock frequency and allow evolution to optimise dendritic and axonal delays by changing the length and path of each branch.

One limitation in this design is that there is only a single synapse unit available in each glial cell. The other option is to assign more hardware resources to glial cells and have two (or even more) synapse units in each glial cell. By increasing the number of synapse units, fan-in of the dendritic MUXs increases (to 6 inputs for 2 synapse units) and hardware resources to implement them grow exponentially. For efficient use of the hardware resources, there should be an appropriate ratio of functional resources to routing resources (interconnection-PE ratio) in each cell. Although up to four different dendrites can project into a glial cell, and a maximum of two dendrites can pass through it, the average number of the dendrites passing through a cell will be less than two in practice. Therefore, one synapse unit per glial cell seems reasonable at this point.

5.4.4 Example

Figure 5.6(a) shows a symbolic view of an example microcircuit. It consists of three soma cells in a 6x4 Cortex. Figure 5.7 shows the active circuit elements of the same microcircuit. The bottom soma in the E2 and F2 cells projected three dendrites and one axon. On the bottom edge, there is no dendrite thus the bottom MUX is switched to input 1 to bypass the external circuit and use a pipeline DFF instead. On the bottom-left edge, a very short dendrite is projected into the F1 cell. Therefore, the bottom-left MUX is switched to input 2. In the F1 cell the dendrite is looped back without forming any synapse by switching the corresponding MUX to input E. The dendritic loop is continued on the top-left edge of the soma cell

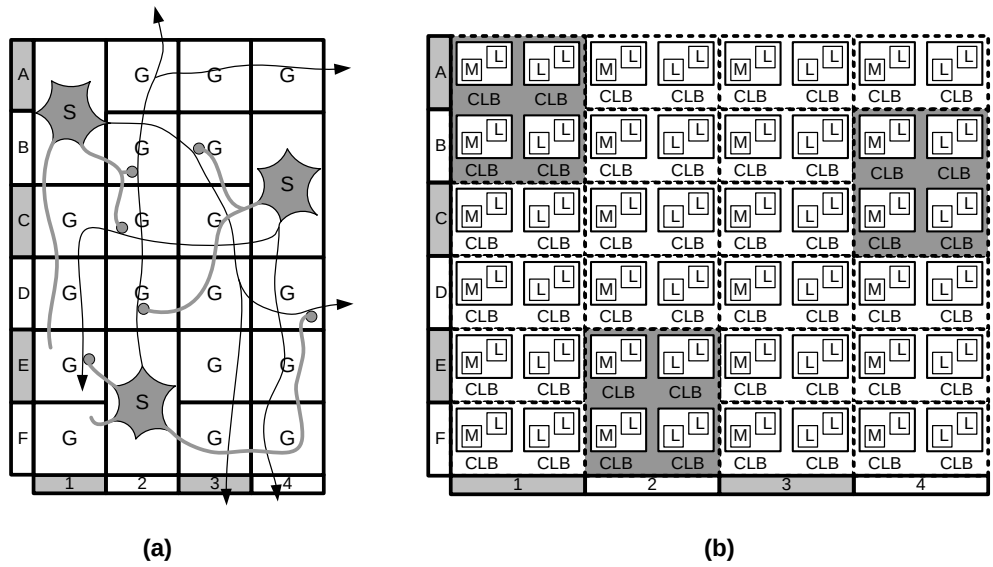


Figure 5.6: (a) Symbolic view of the example microcircuit in a 4x6 cortex. (b) Assignment of FPGA CLBs to glial and soma cells.

with another short projection, this time forming a synapse with an axon coming from north. The other projection of the soma cell on its bottom-right edge has passed through a number of MUXs in different glial cells and formed a synapse with another axon in D4. The dendritic loop of this neuron contains twelve FFs, seventeen MUXs and two synapse units. Its axon has gone through three MUXs and FFs upwards into A2 and then divided into two axons extending outwards. Routing of the projections from the other two neurons can be also tracked in a similar manner. In C3, for instance, a dendrite is divided into two branches. In B2, another dendrite formed a synapse as it extended into C2.

5.4.5 Virtex-5 Feasibility Study

A feasibility study of implementing this cellular structure in the Virtex-5 FPGAs was carried out to verify speed, area and possibility of a dynamic reconfiguration. Two horizontally adjacent CLBs (Configurable Logic Blocks) are assigned to each cortex cell. This is because synapse and soma unit designs make extensive use of Virtex-5 32-bit shift registers (SRL) and only one out of four slices in two horizontally adjacent CLBs is a SLICEM capable of implementing SRL primitives [408]. As soma cells need more hardware resources, they can occupy a square block of four CLBs on the FPGA. This is because assigning 4 CLBs in a row to soma will double the partial reconfiguration overhead as number of frames that must be reconfigured will double. It also leads to employing long-range routing lines of the FPGA for intra and intercellular connectivity. These lines are limited in number and have longer signal delays. Figure 5.6(b) shows how cellular structure of the above example can be implemented in the Virtex-5 CLBs.

VHDL and ISE 9.2i design tools were used for implementation of a sample cellular structure in a LX50T Virtex-5 FPGA. Implementing and floor planning of the soma and glial cells on the chip revealed that it is possible to pack the soma and glial cells in 2 and 4 CLBs respectively. Every two 5-to-1 MUXs with the same set of inputs can be implemented in a 6-input LUT configured as two 5-input LUTs. This way, the whole routing circuit of a glial cell is implemented with six LUTs and eight DFFs, which is less

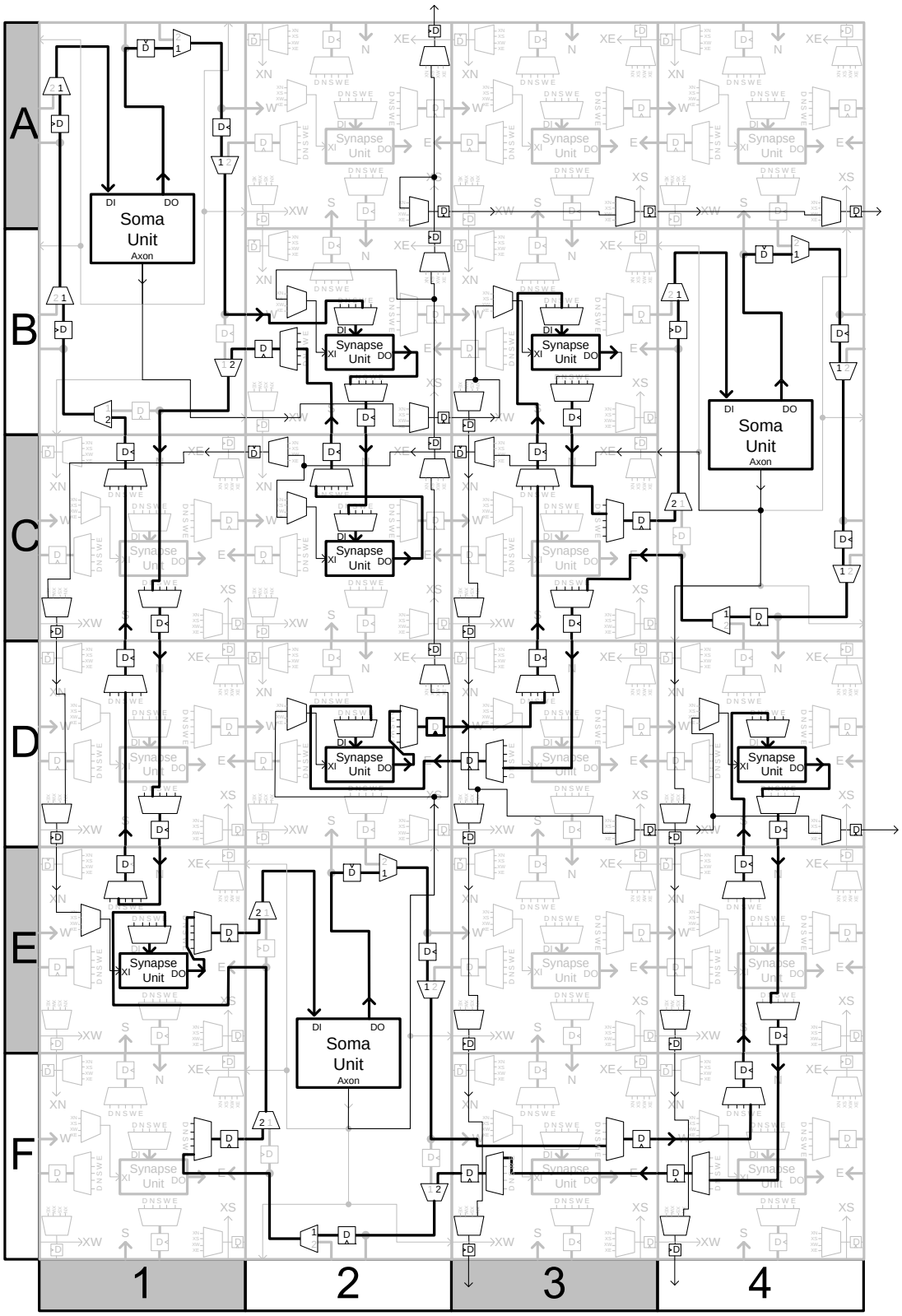


Figure 5.7: Schematic diagram of the active parts of the example microcircuit.

than the available resources in the right CLB of the glial cell. The synapse unit takes most of the left CLB of the glial cell. The routing circuit of the soma cell can be implemented using six DFFs and three LUTs, each configured as two 2-input LUTs. The rest of the hardware resources were more than enough for implementing the soma unit. Therefore, the extra hardware resources in each cell were reserved for future improvements (*e.g.* synaptic plasticity or upgrading to a more bio-plausible soma model) and corrections.

This way it will be possible to pack 1800 glial cells in an entry-level Virtex-5 FPGA (LX50T) and 12960 glial cells in the largest Virtex-5 chip (LX330T). With a 1 to 10 soma to glial cell ratio (each soma cell surrounded by a layer of glial cells), it is possible to implement networks with 150 and 1080 neurons with up to 1500 and 10800 synapses in LX50T and LX330T chips respectively. This might not be the optimum ratio but this can be ideally left to evolution to tune the ratio and placement of the cells in order to optimise the resources and performance. Alternatively, different but fixed ratios and neuron placements can be used to test the effect of neuron placement on the system performance.

Dynamic Partial Reconfiguration and Neuron Relocatability

The cellular structure was first designed to exploit the dynamic partial reconfiguration feature of Virtex-5 FPGAs. Here, the feasibility of different methods for relocation of the neurons at the first stage of the development and runtime modification to parameters and connectivity of the neural microcircuit are studied. The reconfiguration may be carried out in three main steps:

At the first step, the whole area on the FPGA that is assigned to the cortex is configured as glial cells. This is simply done through the standard flow, configuring the device with a bitstream generated from HDL. However, glial cells need to be defined as hard macros so that the exact locations of all MUXs (LUTs) and cell ports be fixed and known *a priori*. Hard macros are blocks of circuit designs that are already placed and routed for specific location(s) of the FPGA substrate and are fixed compared to the rest of the circuit that will be placed and routed later. Hard macros can be placed in any compatible location in FPGA or specific locations using constraints. In both cases the relative location and routing of the internal resources of the macro will stay the same.

In the second phase, soma cells are reconfigured instead of glial cells in the required places using merge dynamic reconfiguration technique [334]. Soma cell should be defined as a hard macro again with its ports carefully matched with the ports of the neighbouring glial cells. The merge reconfiguration technique [334] allows to vertically relocate a module with an arbitrary shape and size (2x2 CLBs in this case). Therefore, a relocatable soma bitstream should be created for each cortex column (2 CLBs wide). In the final phase, soma and synapse parameters and axon and dendrite routings are modified by runtime difference-based dynamic partial reconfiguration of LUTs [406, 376, 377] provided that all the parameters and routings are based on LUTs, RAMs and/or SRLs contents and the exact locations of all these primitives on the FPGA are known. This can be achieved by constraining the placement of hard macros using LOC statements in a UCF (User Constraints File) or original VHDL/Verilog structural description of the cortex. Therefore, it will be possible to grow dendrites and axons and form/eliminate synapses on the fly.

Further investigation of the reconfiguration process on the actual hardware platform revealed that relocatable reconfiguration of soma cells is not such a straightforward process. In a relocatable design, all the routed internal signals of each cell need to be routed only through routing resources inside that cell area and all the cell input/output signals should use the same signals regardless of cell type (soma, glial or IO cell). Finding a streamlined module-based partial reconfiguration method for many relocatable, or even only mutually compatible, modules that can be reconfigured on a modular grid cell structure appeared to be out of the scope of this study.

Therefore, for simplicity, the location of soma and IO cells are predefined and fixed during evolutionary process. Based on those locations, a primary reconfiguration bit-stream will be generated using traditional FPGA design process and tools. Then during the developmental process, the reconfigurable multiplexers (implemented in LUTs, SRLs and RAMs) on the edges of the soma and glial cells are reconfigured occasionally using the results from developmental process. This way the neural simulation can keep running on the chip during development without any interruption. This allows the neurodevelopment process to be provided with the activity data from the network simulation for addition of activity-dependent development and synaptogenesis.

As explained earlier, reconfiguring a SLICEMs can corrupt the content of the RAM and SRLs in the same frame. This is because in the period between reading and modifying to writing the frame it is possible that the content of other RAMs and SRL in the same frame are changed and an old data will be written back to those memory elements. This can be resolved in two ways. First, to freeze the circuit during reconfiguration and second, reconfiguring all those dynamic elements with initial data. The first solution puts the whole cortex on hold even when only one soma or synapse is being modified. This increases the impact of reconfiguration overhead on the simulation performance. The second solution is only useful when all the soma and synapses are reset and a new simulation is started. That means there is no simulation running yet and therefore this method has no advantage over the first method.

Preliminary tests were performed on the hardware platform to find practical ways to reconfigure the chip and to estimate the reconfiguration overheads. The library functions for the MicroBlaze processor provided by Xilinx as the driver for XPSHWICAP IP core, which uses ICAP to reconfigure the FPGA were tested. Preliminary tests revealed that the reconfiguration overhead of using this driver for modifying the content of the LUTs and SRLs is far from the nominal speed of the ICAP (in order of milliseconds) for modifying contents of a single LUT. This is mainly due to the way that SetCLBBits() function works. This function receives the bits that are supposed to be reconfigured in a LUT, along with the position of the LUT in the FPGA (X,Y of the CLB, Slice number, LUT number) as parameters. For the function to be able to set these bits it first needs to read 4 different frames that contain bits for this LUT, modify them and write them back. It also needs to fix the order of the bits depending on the slice type (SLICEM or SLICEL) and position of the LUT in the FPGA. A large amount of overhead is involved in reading or writing each frame of data. Some ICAP initial codes, and commands, frame address, etc. need to be written first before reading or writing a frame. Also because ICAP has an internal frame buffer, for reading each frame, it is needed to read two frames to push the data out of the ICAP.

Similarly two frames of data need to be written back to push the frame into ICAP. This requires a total of 16 frame read/writes for setting a single LUT. This is far from efficient reconfiguration. Another part of the speed problem may relate to the speed of the XPSHWICAP IP core that is documented to work with at a maximum clock frequency of 100MHz with 32-bit words. It appears that depending on where and how the XPSHWICAP IP core is place and routed the performance of the IP core varies. Some results in the FPGA design community suggest that by placing and routing this core carefully, it would be possible to achieve speeds of up to 200MHz. With a few constraints on the placing of the IP core a clock frequency of 125MHz was achieved.

To mitigate some of these reconfiguration overheads, a set of new library functions were developed that allows reconfiguration of 80 LUTs to be configured in one go by reading and writing four consecutive frames (see appendix B for the source code). This reduced the software and interfacing overheads significantly and allowed four frames to be read, modified and written back in 120μs. To develop these library functions some parts of the reconfiguration bitstream format related to partial reconfiguration of LUTs and SRLs were reverse engineered (see appendix A for details). Needed routines for reconfiguration of one LUT, a group of LUTs and all the LUTs in a full slice column of FPGA using MicroBlaze ^{TM}processor was implemented and tested on the actual hardware.

System Performance Estimations

To reconfigure a cortex cell, 4 FPGA frames should be read, modified and written back. These 4 frames also contain data for 19 other glial cells. Therefore, 20 cells can be modified in one read-modify-write operation which takes about 120μs. This should be performed 72 times to completely reconfigure all the cells in a cortex of grid size 12x120. Another set of frames should be accessed for modifying the soma cell parameters and synaptic weights. Thus, each reconfiguration cycle of the whole cortex takes about 17.28ms. Assuming 100 development cycles during simulation, total reconfiguration time for each network would be equal to 1.7s. Since only some of the frames need modification in each cycle, a maximum of 1s reconfiguration time can be expected. A few other techniques can be also used to improve performance. For instance, in case of routing modifications, it is possible to keep the last changes in memory and skip the read operations to effectively cut the frame modification time in half.

Assuming a clock frequency of 160MHz for simulation of the network, and a dendrite length of 72 grid cells (maximum distance between two cells on the cortex), a membrane potential can be updated about 1 million times per second. It results in a simulation speed 1000 times faster than real time simulation of biological neurons with a reasonable resolution of one update per 1ms of simulation. With data sets of 1000 samples of approximate length 1s each, it will take 1 second to simulate the network activity for the whole data set.

In case that it would be necessary to stop the simulation when reconfiguring the chip, and if the microcircuit should be developing during the simulation (only for one cycle of activity-dependent development), maximum estimated evaluation time for each microcircuit would be about 3 seconds (2 seconds of simulation and development + 1 second of simulation). This will give about 30,000 evaluations a day, or 1000 generations a day assuming a population of 30 individuals.

5.4.6 Detailed Design and Implementation

The FPGA used in this study (XC5VLX50T) has 120x30 CLBs. Part of this area is needed for I/O circuits, MicroBlaze embedded processor, and the XPSHWICAP cores. These 3600 CLBs are heterogeneous. Every other column of CLBs have a SLICEM on the left. Also, the two consecutive CLB columns on the left side of the DSP blocks column, in the middle of the FPGA, have a SLICEM. However, a large region of 120x24 CLBs on the right side of the FPGA is fairly homogeneous that can be used for a good size Cortex of 120x12 grid cells. The remaining region of 120x6 CLBs is more than enough for IO, MicroBlaze and XPSHWICAP cores.

The global clock signal of the whole Cortex, which is used by all memory elements (RAMs, DFFs, and SRLs) in the glial and soma cells, needs to be gated by a clock enable signal which is controlled by the processor. This allows processor to freeze the Cortex before reconfiguration, by disabling the clock signal, and then resuming it after reconfiguration. This is necessary to avoid corruption of the neighbouring memory elements in the Cortex during configuration of a SLICEM as explained in section 5.4.5.

A set of IO cells were designed for sending and receiving spikes to the left side edge of the cortex. As delivering and receiving spike trains to and from a cortex of this size and speed requires 240Mbits/s raw throughput, I/O cells must be able to cope with this bandwidth and allow encoding and decoding of the spike trains to a more compressed representation that can be handled by the embedded processor. Spike density coding appears to be a relatively simple and useful method. DSP48E blocks available right on the edge of the cortex were exploited to create spike generators and spike counters that are connected to the MicroBlaze processor as IO cell custom IP cores. This way the processor is able to read and write the spike densities to and from these IO cells with simple memory accesses at high speed. The detailed design of the IO cells is explained in appendix D. Figure 5.8 shows how the Spike Generator and Spike Counter modules are connected to the Cortex. Each spike counter is implemented using a DSP48E block configured to work as eight 6-bit counters that can be used to count the number of spikes received during a 1ms interval. Fifteen DSP48E blocks are used for counting the spike out of the Cortex. Spike generator registers are 32-bit registers that can be written to using the processor to generate spikes in the next clock cycle. The rest of the available DSP48E blocks on the FPGA can also be used as timers for generating spikes with specific densities when needed by the application. This design allows both generation of spike trains with a variable density and density measurement of spike trains from the Cortex in real time. It also supports sending spikes with specific timing and measurement of the timing of received spikes.

Another DSP is used to generate the simulation clock which shows duration of one ms of simulation by a pulse of width equal to 1 cortex clock every n clock cycle. this can also be used to interrupt the CPU for getting more data or something. A timer can also be used as it is easier to get an interrupt signal from a timer in the embedded system.

Few Cortex designs with different neuron placements (discussed in detail later in chapter 6) were implemented and successfully placed and routed in the FPGA. Figure 5.8 shows a sample neuron placement of a smaller 16x12 cortex. Implementing the reconfigurable cortex required using constrained Hard

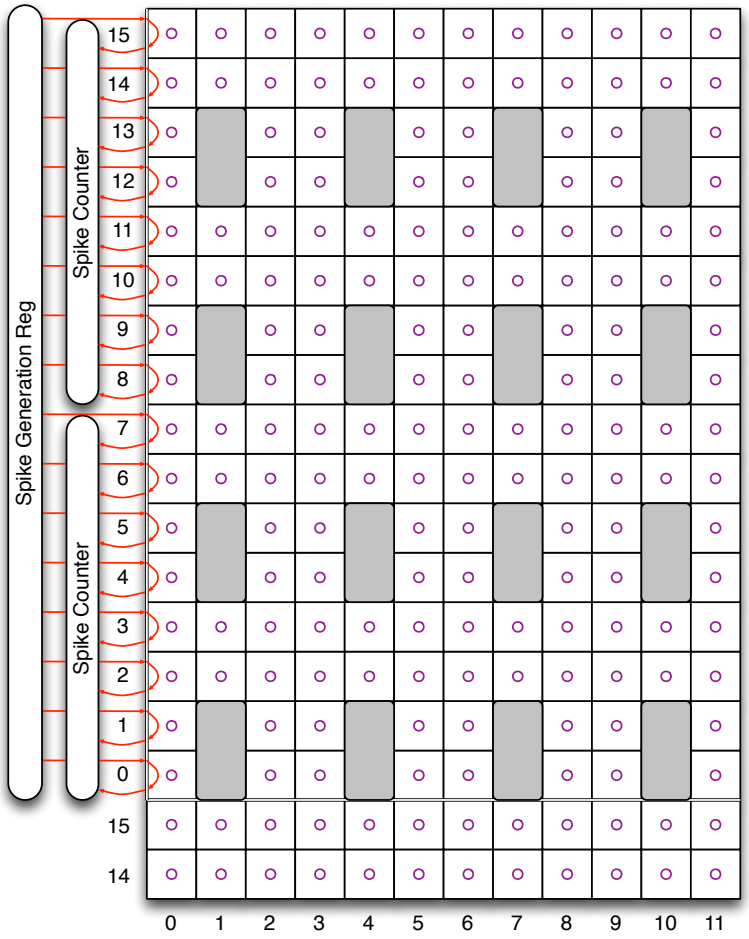


Figure 5.8: Spike Counters and Spike Generators as IO cells connected to a sample 16x12 cortex. This figure also shows the spike signals being looped back in to the IO cell for verification and testing of the spike generator and counter modules.

Macros for Soma and Glial cells, since the suggested workflow by Xilinx using RPMs (Relatively Placed Macros) did not work in the design tool as expected. The proper placement and routing of the cells was a cumbersome and challenging task. It is necessary to constrain both the placement and routing of the reconfigurable elements so that the location of every element is known and LUT and SRL inputs are not interchanged (Using LOC, BEL, and LOCK_PINS statements in UCF file [414]). Different versions of the design tool behaved differently and were sometimes unstable, which immensely increased the time and complexity of this phase of the project. However, Xilinx is announcing now that the partial reconfiguration and other related workflows has been streamlined in the newer versions of the tool.

5.4.7 Verification and Testing

All the hardware or software modules implemented for the Cortex and its reconfiguration through the embedded system were verified and tested at different stages.

Software Verification and Testing

Two simple reconfigurable circuits were designed and implemented that allowed every single LUT in the FPGA to be connected to the processor through a large multiplexer in turn. Then every LUT was

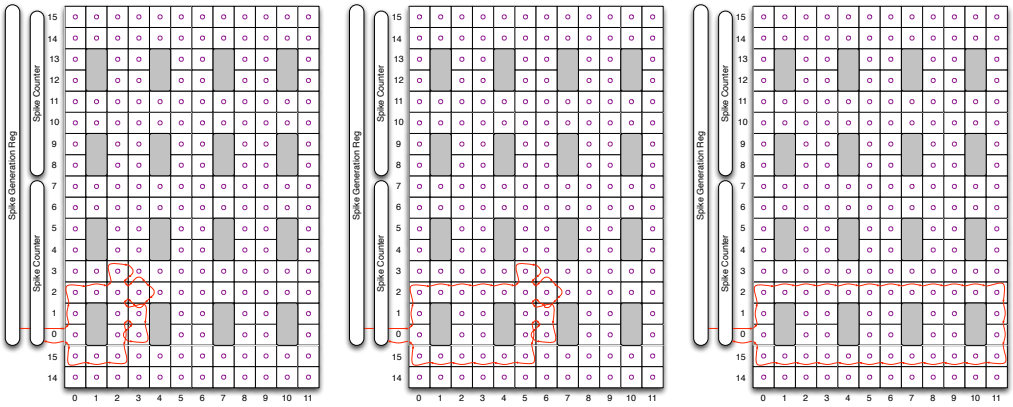


Figure 5.9: Three examples of test cases for testing axonal signal routing around soma cells on the cortex. The particular shape in (a) and (b) insures that every different switch state is at least once tested through an axonal path.

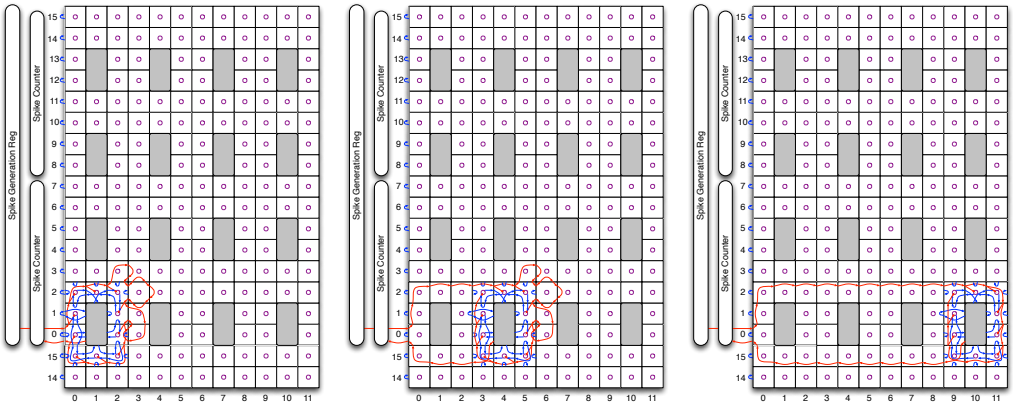
reconfigured with all the possible single bit values through the developed software library and its contents verified by the processor. The software library functions were also tested and debugged during the end-to-end hardware verification test cases (see appendix B and C for the source code of the program developed for the embedded processor.)

Hardware Verification and Testing

Glial, soma, and IO cell designs were simulated using Xilinx simulation tool at the design stage. A program running on the embedded processor automatically reconfigured the Cortex for different test cases and then generated spikes in the input and monitored the spikes on the output.

After implementation, first, IO cells (the spike generators and counters) were tested using axonal loops that connected each cortex input directly to the axonal output in the same cell at the very edge of the Cortex. Figure 5.8 shows these spike signal loops in a 16x12 Cortex.

Secondly, glial cells and reconfiguration of their switches were tested using a pattern that has all the different switching situations to verify that spikes and dendrite packages are passed correctly and with the expected delays. Spikes were fed into cortex inputs and routed in different directions through the dendritic and axonal paths of the glial cell. The axonal paths were configured around each soma cell and spikes were sent through the axon and received on the other end using spike generator and counters. Figure 5.9(a) to 5.9(c) show examples of the test cases for testing axonal routing. In the next step, depicted in examples of figure 5.10(a) to 5.10(c), axons and dendrites were reconfigured around every soma cell in each test case and a single dendritic wire was used to monitor the dendritic packets of the soma through one of the FPGA pins for debugging. Each soma was configured as a simple regular spiking neuron. After testing the soma without any input, one of the synapses was connected to the axon and behaviour of the soma was monitored. Then the synaptic weight was changed and its effect on the activity of the neuron was checked. This process was repeated for all the synapses and soma units in the Cortex using a software program.



(a) (b) (c)

Figure 5.10: Three examples of test cases for testing dendrite signal routing around soma cells on the cortex. The particular shape of the dendrite in (a) and (b) tests all possible different switch states through a single dendrite loop.

Finally, a single soma unit was configured with 10 synapses connected to 10 different Cortex inputs. Figure 5.11 shows the connectivity of the single soma cell in this set of test cases. Each synapse weight register was configured by exponents of two so that by reading different binary values at each update cycle it was possible to feed different presynaptic currents into the soma. Testing and verification process revealed some problems and limitations in the design and implementation of the Cortex and Digital Neuron model. After addressing them through few modifications that are reported in the next section, verification process was repeated.

Modifications to the Cortex and Digital Neuron Model

Final stages of the testing and verification processes revealed a number of issues in the Cortex and Digital Neuron model that needed to be addresses before further development on their basis. There were a few bugs in the implementation detected during the verification process that were fixed before continuing the verification.

Also, in the design of the Digital Neuron model, the zero crossing points of the equation 4.11 are always at the same value of u_r and u_t and only middle point and two ends of the curve are controllable by parameters. This poses a major limitation on the resting and threshold potentials of the neuron that can only be changed by postsynaptic current I or a bias value. At that point it was simply assumed that adding a synapse with an always-active presynaptic input to the dendrite of the neuron will be enough to add a bias to the membrane potential. While this was a valid assumption, now that the the soma cell design is complete, it is clear that still enough resources are available in the soma cell to incorporate that functionality into the soma unit and save that synapse for better use outside of the soma. Therefore, a 16-bit shift register with a feedback loop that stores the bias parameter and a serial adder were added to to the dendritic output of the soma unit. This effectively adds the bias values to the membrane potential in each update cycle before sending it off to the synapse units in the dendritic loop.

Another issue was that when the soma and glial cells around it are configured for very short den-

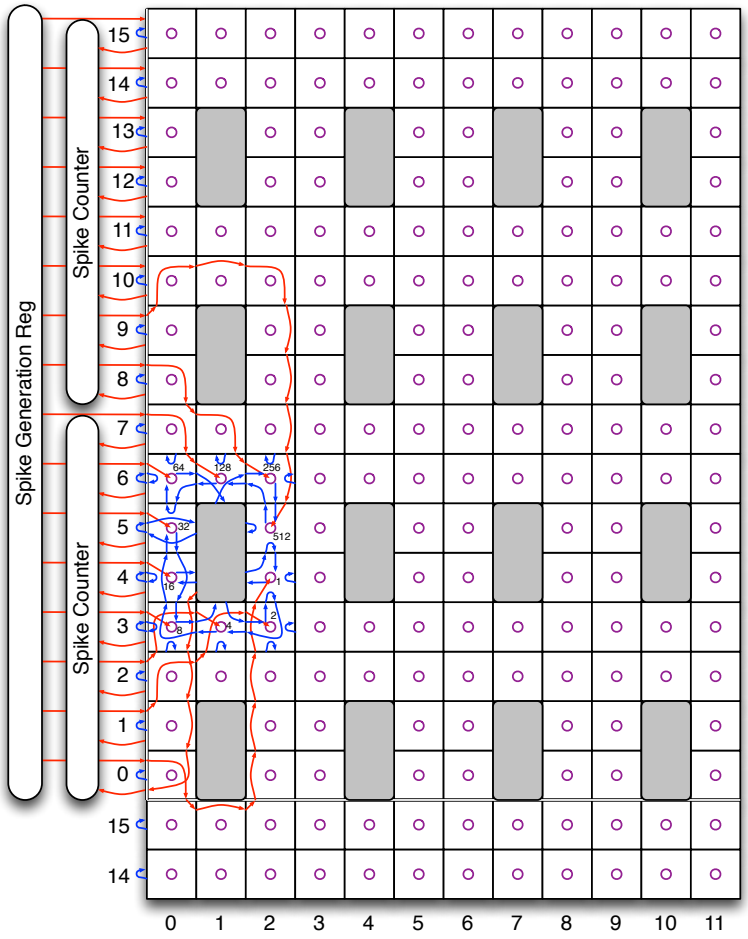


Figure 5.11: Connectivity of a single soma unit with 10 cortex inputs for testing different soma parameters settings and final test of the neuron model behaviour.

drites, the length of the dendritic loop can be too short for the Digital Neuron to work properly. The soma unit will not accept any packets before it is finished processing the last packet. Since there are two taps in the Digital Neuron model update cycle and the first tap is performed when the packet is arriving and the second tap is carried out when the new packet is transmitted, the processing time is equal to the length of the packet (17 bits = 16 bits +1 header bit). During sending the new packet, the soma unit can not receive a packet and therefore a dendritic loop of at least 17 bits is needed. The minimum possible dendritic loop length in the current soma and glial cell designs is six bits (the internal default dendritic loop of the soma cell), which is 11 bits short of the minimum acceptable length for soma unit. To resolve this issue, a 11-bit shift register (called pad) was added to the dendritic output of the soma unit to pad the dendritic loop and delay the packet for 11 clock cycles.

A third issue spotted during verification was resuming the simulation after configuration. Sometimes after a simulation period, when a soma cell was reconfigured and the Cortex was resumed, it could stop working properly. There were two cases for this: In the first case when the Cortex was paused this soma was processing a packet. After reconfiguration, all the SRLs were configured based on the initial state of the soma but soma was not in its initial state and therefore the state of the parameters in the SRLs

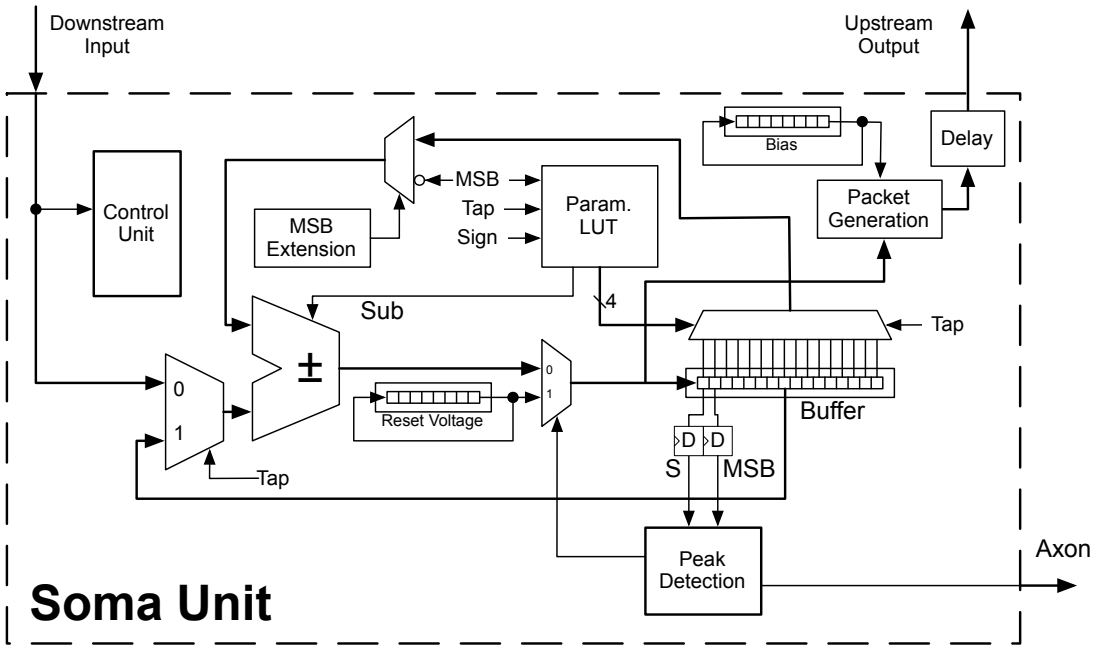


Figure 5.12: Block diagram of the revised soma unit showing the bias register, and the delay block that were added to the soma unit. The global reset and clock signals are not shown here.

did not match with the state of the soma that was stored in the control unit DFFs and SRLs. All these state memory elements also needed to be initialised. This could be done by reconfiguration of those SRLs and DFFs. Initialising SRLs can be carried out at the same time of reconfiguring the parameters in the other SRLs as long as they always belong to the same frame. However, initialising DFFs by reconfiguration creates an extra overhead as those memory elements are located in another frame. It is more efficient to initialise them using a global reset signal.

In the second case, when the Cortex was paused, the soma was in idle mode awaiting a packet to arrive, meaning that there was a packet being processed in the dendritic loop. In this situation soma would be initialised and after resuming the Cortex, it would send a packet immediately. This could effectively create two different packets in the same dendritic loop and potentially disturb the normal behaviour of the soma unit. This can be also resolved by clearing the old packet from dendritic loop by resetting all the pipeline DFFs in the dendritic loop. However as dendritic loop of a soma cell can potentially involve any glial cell in the Cortex, this means to reset all the glial cells after reconfiguration of a single soma, which can clear all the valid packets of other somas in the Cortex. Therefore, there is no way other than reconfiguring all the somas and resetting all the DFFs in the Cortex globally before resuming the Cortex. To resolve this issue, a global reset signal that is controlled by the processor was added to all the glial and soma cells. To reconfigure the Cortex, processor must enable the global reset, then after at least one clock cycle disable the global clock, perform the reconfiguration of all the soma cells, disable the reset and finally enable the clock to start the simulation. Other solutions are also conceivable that involve, for example, adding an initialisation mechanism to the soma unit that makes it wait for the old packet to clear out of the dendritic loop and also initialises all the state DFFs in the soma. The global reset

solution was selected as a fix for this issue due to its simplicity and the fact that allocating resources to that mechanism required manual placement of the soma cell hard macro, which was a complicated and time consuming task due to the issues in the tool chain. These issues are discussed in the next section. All the above modifications and bug fixes were applied to the soma and glial cells. Figures 5.12 and 5.14 show a block diagram and a more detailed diagram of the revised soma unit respectively.

End-to-end Testing

For end-to-end testing of the Digital Neuron model in the Cortex, different soma parameter settings were found to generate all the different behaviours expected from a normal QIF neuron model. The parameter settings were found by trial and error using the similarity of the PLAQIF model with a QIF neuron model. Different routines for testing these behaviours were developed in the software for the embedded processor (see appendix C for source code). Apart from the axonal output of the neuron available to the processor through the Spike Counter, an extra dendritic signal was also routed to the edge of the cortex that allowed dendritic packets to be monitored and used both for debugging and parameter tuning. The sample stimuli were taken from [169]. Figure 5.13 shows the verified spike timings of the Digital Neuron model compared with the response of Izhikevich neuron model along with the PLAQIF soma model function curves used to achieve each behaviour. For Class I and Class II excitabilities, a ramping up postsynaptic current from zero to 2048 during 256 update cycles was used. Note that a normal QIF neuron model is not able to show Class II excitability [169]. However, since a PLAQIF soma model has more degrees of freedom than Izhikevich model, it was possible to use that flexibility to produce a behaviour somehow similar to Class II excitability and tonic spiking using unusual function curves. Table 5.5 gives a list of the parameter settings used for generating different behaviours tested.

Table 5.5: A list of six different behaviours of a Digital Neuron model tested in the Cortex and the parameter settings used for generating each behaviour. T1 and T2 columns show parameter values for Tap 1 and Tap 2 of the PLAQIF soma model for small, large, positive, and negative values of membrane potential. V_{reset} and V_{bias} columns show the reset potential and the constant bias value added to the membrane potential in each update cycle. Figure 5.13 shows the response timing of the neuron model for these settings.

Behaviour	Parameters									
	ParamLUT									
	Negative				Positive					
	Small		Large		Large		Small			
	T1	T2	T1	T2	T1	T2	T1	T2	V_{reset}	V_{bias}
Class I excitability	-12	-12	-3	-3	0	0	12	12	-8000	32
Class II excitability	-3	-4	-4	-4	3	3	-3	-4	-32000	2500
Tonic spiking	-14	-14	-14	-14	2	2	-2	-2	-20000	5
Spike latency	-4	-4	-3	-3	1	1	4	4	0	2048
Integrator	-4	-3	-3	-3	1	1	4	3	0	2800
Bi-stability	-4	-4	-3	-3	3	3	4	4	100	2048

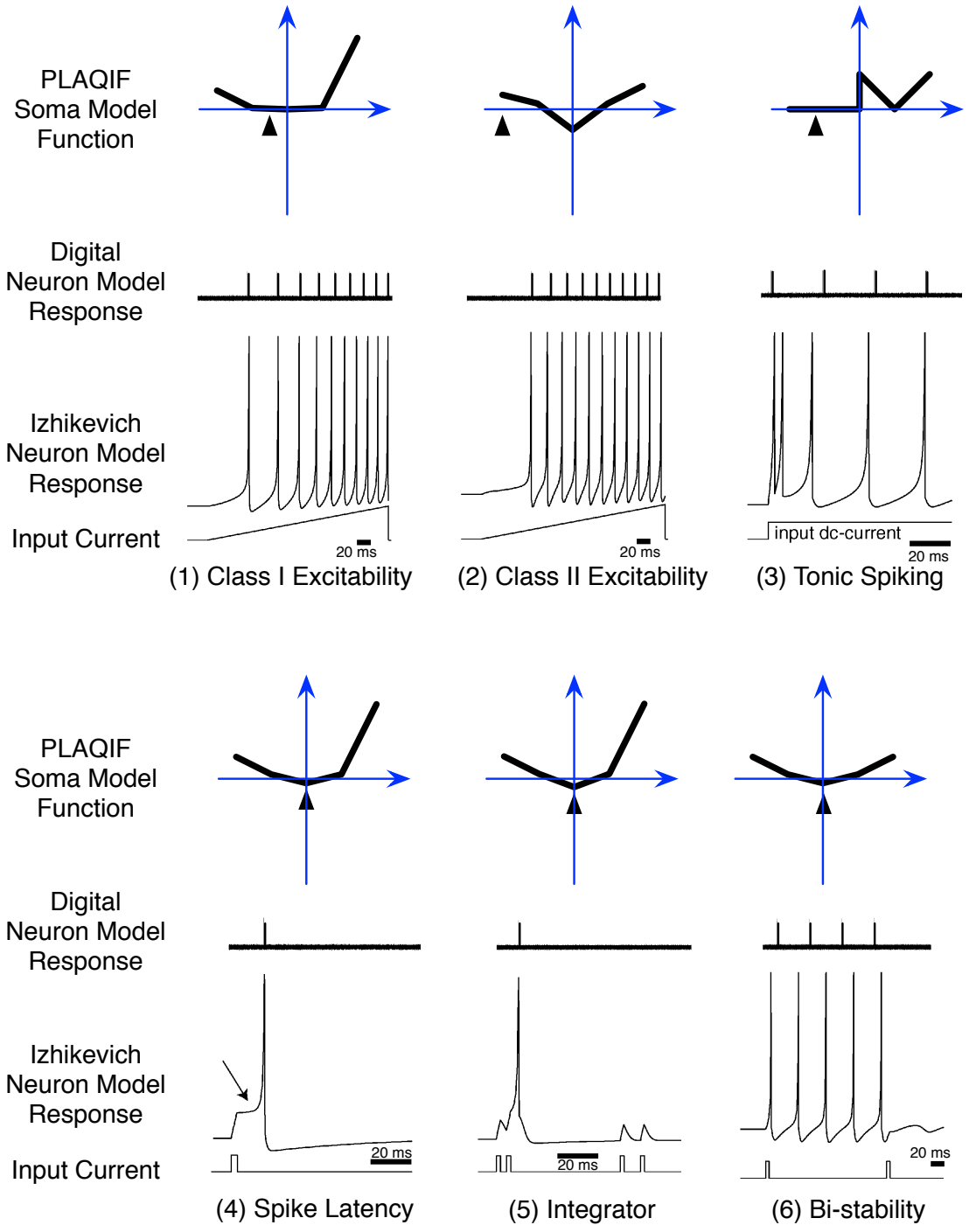


Figure 5.13: Results of the Digital Neuron model end-to-end verification in the Cortex with comparison with the Izhikevich model response. The short black horizontal line represents the time scale of 20ms. For each of the behaviours 1 to 6, the input current of the neuron, response of the Izhikevich model, response of the Digital Neuron model, and the function curve of the PLAQIF soma model used in the Digital Neuron are shown. The small black triangles on the function curves show the V_{reset} parameter value. The parameter settings to achieve these function curves and behaviours are reported in table 5.5.

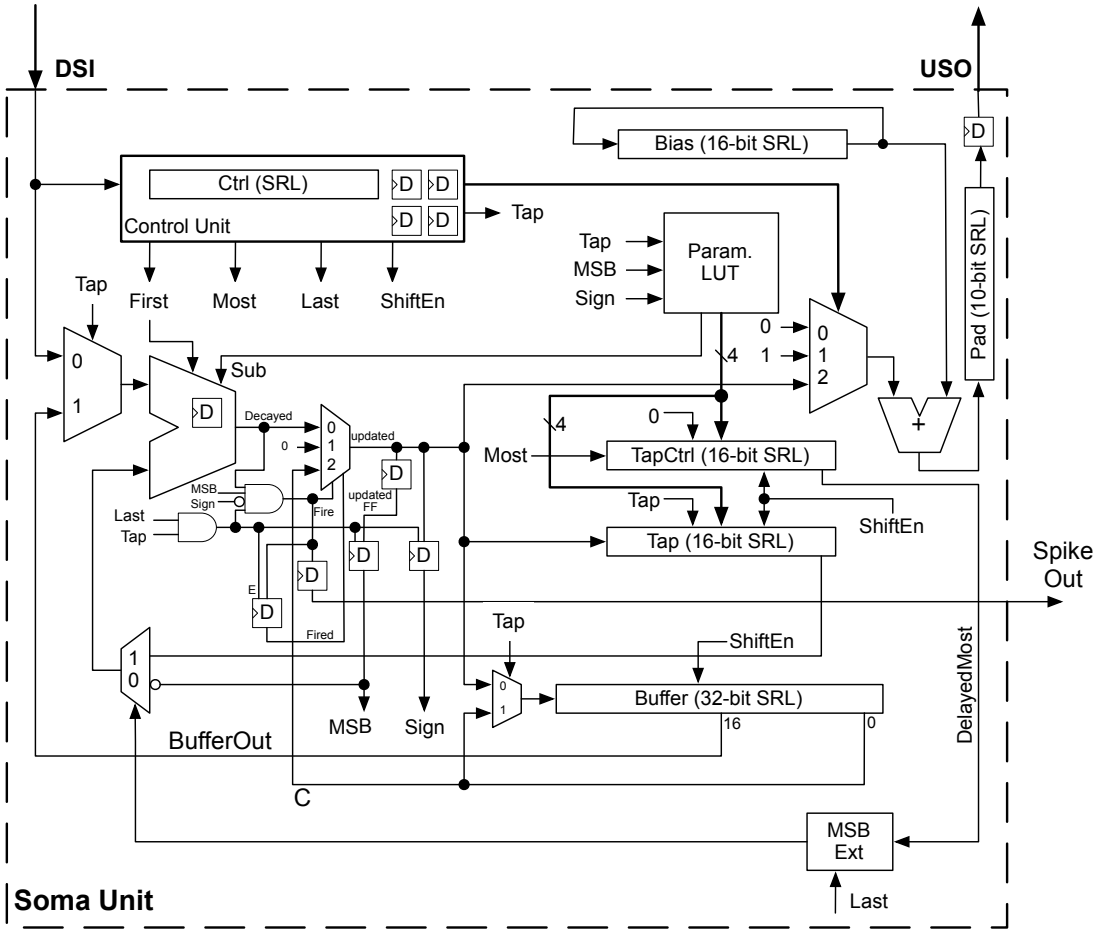


Figure 5.14: A detailed diagram of the revised soma unit showing the bias register, serial adder and padding shift register that were added to the soma unit. The global reset and clock signals are not shown here.

The whole verification process explained in section 5.4.7 was repeated and all the hardware and software modules involved in the reconfiguration of the Cortex and simulation of neurons in the Cortex were successfully verified and tested.

5.5 Practical Considerations

Practical challenges, options, and issues in the detailed design and implementation of the Cortex are further elaborated and summarised in this section. Possible extensions and modification that were not implemented in the Cortex are also discussed and evaluated.

Three very major practical factors impacting the bio-plausibility of the Cortex are: relocatability of the neurons, reconfiguration method (DPR versus Virtual FPGA), and a time-multiplexed switching for the intercellular communication network. It appears that, in practice, despite the low performance of the plug-in method for neuron relocatability, the hardware cost (for Bus Macros) and complexity of creating relocatable neuron modules with the current design tools and devices is still so high that may justify using a plug-in method to achieve some level of relocatability. However, the design tools are changing rapidly. Not only Xilinx is claiming that the very last versions of the design tool (Vivado)

has streamlined the partial reconfiguration workflow, but also new open-source tools are introduced by researchers that allow better verification and analysis of designs for partial reconfiguration, introduce automatic generation of relocatable modules, and a new Bus Macro [325, 47].

A virtual FPGA method was not used in the case study due to its very high hardware cost and abundance of its record in the literature. However, it can be a viable option in a large FPGA. A virtual FPGA method is more bio-plausible than DPR method, but only if the Cortex is reconfigured by distributed developmental processes on FPGA. In that case its detailed design and specifications depend on the local developmental mechanisms in the hardware and how it can be bio-plausibly integrated with those circuits. Therefore, virtual FPGA approach deserves to be revisited in the next chapter when distributed developmental processes in hardware are discussed.

Time-multiplexed Intercellular Network

Extending the 2D intercellular network of axons to a time-multiplexed virtual 3D network can affect the bio-plausibility of the cortex model by allowing longer and denser axonal interconnections to be developed more efficiently on the same 2D infrastructure. To implement a time-multiplexed virtual 2D network for axons in the Cortex, each 4×1 multiplexer needs to be fed by switching data from a schedule table (a $2n$-bit RAM, where n is the scheduling period length). Figure 5.2 shows the general circuit needed for time-multiplexing. All schedule tables need to be addressed by a time-slot counter. A shift register can be used efficiently to implement the combination of a schedule table RAM and a local time-slot counter. It is also possible to store the raw switching logic of different multiplexer states in an LUT that is addressed by a time-slot counter.

In the case study design, a 6-input LUT is used as two 5-input LUTs to implement two axonal switches. One of the LUT inputs is left unused. A single-bit counter connected to this input in all glial cells gives a schedule period of 2 with minimum hardware cost. However, it only doubles the band width of the network as it is not possible to route an axon around an obstacles by using the other time-slot. Figure 5.15 shows two 1-dimensional (Virtual 2D) networks with schedule periods of 2 and 4, and the links between grid cells. Grey arrows represent possible paths and blue path shows how a spike can travel through the network from a source (s) to a destination (d) avoiding an obstacle (O). With a schedule period of two, even with no other congestion, it is only possible to avoid half of the obstacles and there is not much a routing process can do to go around the other ones. However, with more than two time slots (a 4-slot schedule is shown in the figure) it is possible to turn around and avoid the obstacles while still routing towards the destination cell. In the same way, longer schedule periods can increase the network routing capacity super-linearly. However, the memory capacity needed for scheduling grows linearly with the schedule period.

To be able to expand the axonal network to a schedule period of four, it would be possible to used a 6-input LUT for each axonal multiplexer and use the extra two inputs for slot addressing. The original glial cell design uses three 6-input LUTs for switching axons (two LUTs for external links and one for the synapse axonal input). Adding 2 extra 6-input LUTs to the circuit is enough to provide the extra hardware for both switches and storing the switching schedule for a 4-time-slot period. It is possible

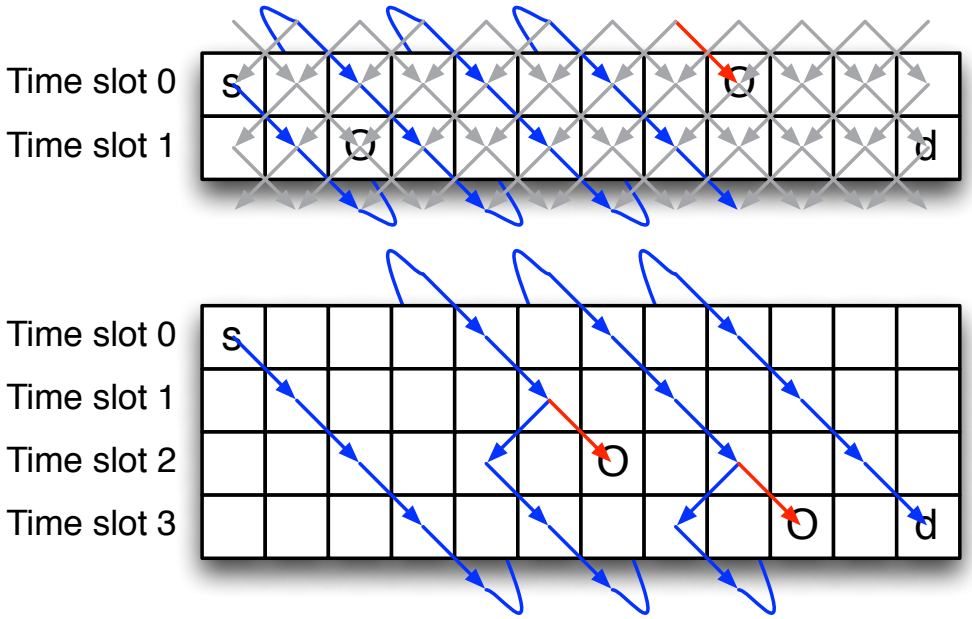


Figure 5.15: Two 1-dimensional (Virtual 2D) networks with schedule periods of 2 and 4, and the links between grid cells. Grey arrows represent possible paths and blue path shows how a spike can travel through the network from a source (s) to destination (d) avoiding an obstacle (O). With a schedule period of two, even with no other congestion, it is only possible to avoid half of the obstacles and there is not much a routing process can do to go around the other ones. However, with more than two time slots (a 4-slot schedule is shown in the figure) it is possible to turn around and avoid obstacles while still routing towards the destination cell.

to allocate two global clock signals and a central counter for time-slot addressing. Also, a very simple 2-bit grey-code counter can be realised inside each glial cell using only two flip-flops if needed. If it was intended to allocate more hardware resources to axons, it is also possible to use 32-bit shift registers as self-counting schedule tables that feed the switching data automatically to the multiplexers and simply increase the schedule period up to 32 bits. However, to achieve that, ten extra shift registers are needed. In a less ambitious design it would be possible to use 32-bit SRLs as two 16-bit SRLs and with only five extra shift registers achieve a schedule period of 16. Table 5.6 shows a summary of the estimated hardware cost overhead compared to the current glial cell design for a few different implementations of the time-multiplexed intercellular network.

This can be viewed as a trade-off between scalability and bio-plausibility versus hardware cost. However, only the hardware cost of the memories grow linearly with the time-slot period, which is a much better trend than linear growth of all hardware in a real 3D network.

Since in the above example designs, the axonal input of the synapse unit is also time-multiplexed, a synapse can be formed for any of the axons passing through a glial cell. It will be also possible for different axons to share the same synaptic weight with no extra hardware cost. However, in all the above designs the soma cell must be modified slightly so that it generates the spike only in the first time-slot or a specified slot and the axonal outputs of the soma cell on each edge deflect the incoming spikes back to the neighbouring glial cells in all other time slots. This way it would be possible to put the incoming axonal switches in glial cells around soma cells to a good use for changing the time-slot of the spikes

Table 5.6: A summary of the estimated hardware cost overhead for each glial cell compared to the current glial cell design for a few different implementations of the time-multiplexed intercellular communication network

Implementation (Schedule period)	Hardware Overhead Estimate			
	# 6LUTs	SRL32s/RAM64s	DFFs	Global signals
Raw encoding w/ local counter (2)	0	0	1	0
Raw encoding w/ central counter (2)	0	0	0	1
Raw encoding w/ local counter (4)	2	0	2	0
Raw encoding w/ central counter (4)	2	0	0	2
Optimised encoding w/ local counter (16)	2	5	0	0
Optimised encoding w/ local counter (32)	2	5	5	0
Optimised encoding w/ central counter (32)	2	5	0	5
Optimised encoding w/ local counter (32)	2	10	0	0
Optimised encoding w/ local counter (64)	2	10	6	0
Optimised encoding w/ central counter (64)	2	10	0	6
Optimised encoding w/ local counter (64)	2	20	0	0

right outside a soma cell. These switches were allocated but unused in the original design.

The clock frequency of the axonal network can be different from the rest of the system. This allows scaling all the axonal delays up or down. However, this needs particular attention in the design of the asynchronous interfaces of axonal network with soma and synapse units, which can be achieved by using latches. Reconfigurable clock management cores available in Virtex-5 FPGAs (DCM_ADV) [409] allow multiplication and division of the original clock signal to generate different clock frequencies for the axonal network. Therefore, it is also possible to allow evo-devo processes to regulate the global axonal delays by changing this frequency.

Compactness (Hardware Cost)

Apart from the trade-off between compactness and bio-plausibility in time-multiplexed design there are other small changes that can affect the compactness of the design. For example, the internal dendritic loop in soma cell is not necessary. This is due to the existence of possible loopback paths in the dendritic network right outside a soma cell in the glial cells. However, having the internal loop allows a soma cell to still work even if most of the glial cells around it are faulty, adding to the fault-tolerance and reliability of the Cortex. Removing the internal loop frees up 6 DFFs and three 6LUTs that can be used for other features such as upgrading the neuron to a piecewise-linear estimation of Izhikevich model.

Performance

Simulation performance can be improved by circuit optimisation and carrying out place and route steps carefully even manually with tight speed constraints, and exploration of different synthesis and place and route options in the design tool. The final verified implementation of the glial and soma cells achieved

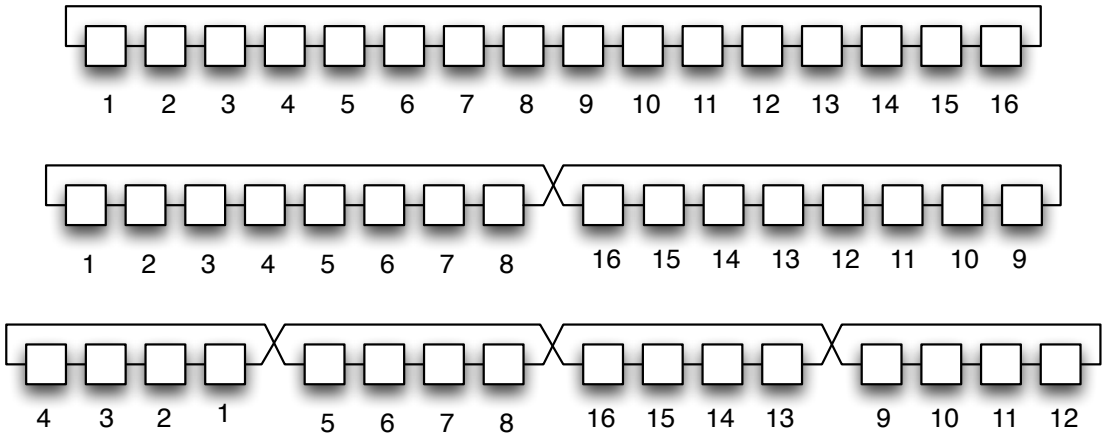


Figure 5.16: Solution to wrap-around wire delays in a 16 node ring network. The top connectivity pattern involves a very long wire between the first and last nodes while other links are very short. In the second pattern this delay is divided in half between two wires by flipping right half of the nodes. In the third pattern a quarter of the nodes at each end of the ring are also flipped again to cut the delays in half again.

clock frequencies of 320 and 200MHz respectively. By spending more time it was possible to improve the the soma performance up to at least 300MHz as well. However, that was not the bottleneck of the simulation performance. The maximum clock frequency of the final cortex implementation was 110MHz as the design tool could not meet any better speed constraints. This was mainly due to very long wrap-around wires connecting the top and bottom of the Cortex. While there were enough long-range wires available to connect all the axonal and dendritic wrap-around connections, the delay of these long-range wires are so high that for some of them a cell to cell delay of about 6.5ns was reported after place and route.

There are a number of solutions to this problem: The first solution would be to add an extra layer of pipeline flip-flops on the receiving end of these lines or in the middle of the FPGA. Although this will help to some extent, finding free flip-flops in the middle of the Cortex that is already filled with glial and soma cell hard macros is not easy. Moreover, that would disturb the homogeneity of the Cortex. The better solution is to flip half of the Cortex upside-down. Figure 5.16 shows how this can distributes the delays over the local connections and avoid having very long wires for wrap-around signals. However, this will slightly complicate the reconfiguration process as the order of the rows in the Cortex will be changed. By applying this technique a few times it is possible to remove the bottleneck from the wrap-around wires.

There are other sources of delay in the implementation of the Cortex. The Xilinx design tool was not able to place and route the Cortex when glial and soma cell hard macros were already routed. This was mainly because when these hard macros were generated it was not possible to route all the local signals through the local wires and some longer ones were going through the neighbouring CLBs. This stopped the place and rout process when it could not use that wire in the middle of next hard macro.This was resolved by unrouting these hard macros allowing the design tool to have more flexibility to change the routing after placement. Although this resolved the place and route problem, it introduced another

problem. The design tool reroutes all the internal signals of each cell ignoring the timing constraints for those signals assuming they meet the timing requirements as they come from a hard macro. Design tool can not perform a timing analysis on hard macros. Xilinx's official workflow for generating a grid of cells such as the Cortex with location constraints is using RPMs (Relatively Placed Macros) but in case of glial cells and soma cell there was no single parameter setting that could synthesise both RPMs successfully within the location and timing constraints. Other possible solutions involve manual place and routing and/or direct routing. With the above problems it was very difficult to achieve a timing closure. The final design was synthesised for much lower clock frequency of 100MHz and was manually examined for timing violations. The software driven end-to-end verification allowed testing the whole system including the Cortex and the embedded processor at 100MHz with success.

There are also potential solutions for improving the reconfiguration performance. One is to place the reconfigurable elements in the glial and soma cells in an order that reduce the number of reconfigured frames during reconfiguration and development. However, a placement, which is efficient for reconfiguration may not be efficient for simulation performance as it may involve longer distances between elements and longer delays. However, this trade-off can be overcome by optimising both aspects at the same time with some manual placement inside hard macros.

Another factor is the maximum speed of the HWICAP IP core. The original XPSHWICAP core from Xilinx is not very fast and efficient. This has been already studied and some very fast IP cores supporting up to the nominal speed of the ICAP at 400MBytes/s are suggested for Virtex-5 [31]. Also some researchers have been over-clocking the ICAP in Virtex-5 reporting much higher speeds of up to 2200MBytes/s (5.5x) [144, 85].

With the ICAP and HWICAP being able to deliver such speeds the bottleneck will be the processor that prepares the packets and manages the HWICAP core. The MicroBlaze is a soft processor core that can work at a maximum frequency of 250 MHz. For simplicity and other reasons discussed in chapter 7 the MicroBlaze processor in this system was using the same clock as the Cortex. However, it is quite acceptable that the processor and other cores connected to it run at different frequencies. It is also possible to use an FPGA device with a hard processor core with better performance. It is also possible to use a PC for preparing the frames and delivering them to the FPGA. Then the bottleneck will be the datalink between PC and FPGA. These are elaborated in length in chapter 7 where integration of the whole system is discussed.

5.6 Summary

Figure 5.17 shows a graphical representation of the investigations carried out in chapter 5. In this chapter the significant and general impact of the cortex model design on the bio-plausibility and feasibility of the whole system were discussed. In section 5.1, general definition of bio-plausibility and feasibility measures from chapter 2 were translated into a set of tangible general design factors and constraints in the specific context of the cortex model. Using those general factors, different general design options and approaches and their trade-offs in different aspects of the cortex model design were investigated.

In section 5.2, Intracellular and intercellular communication networks, their characteristics, and

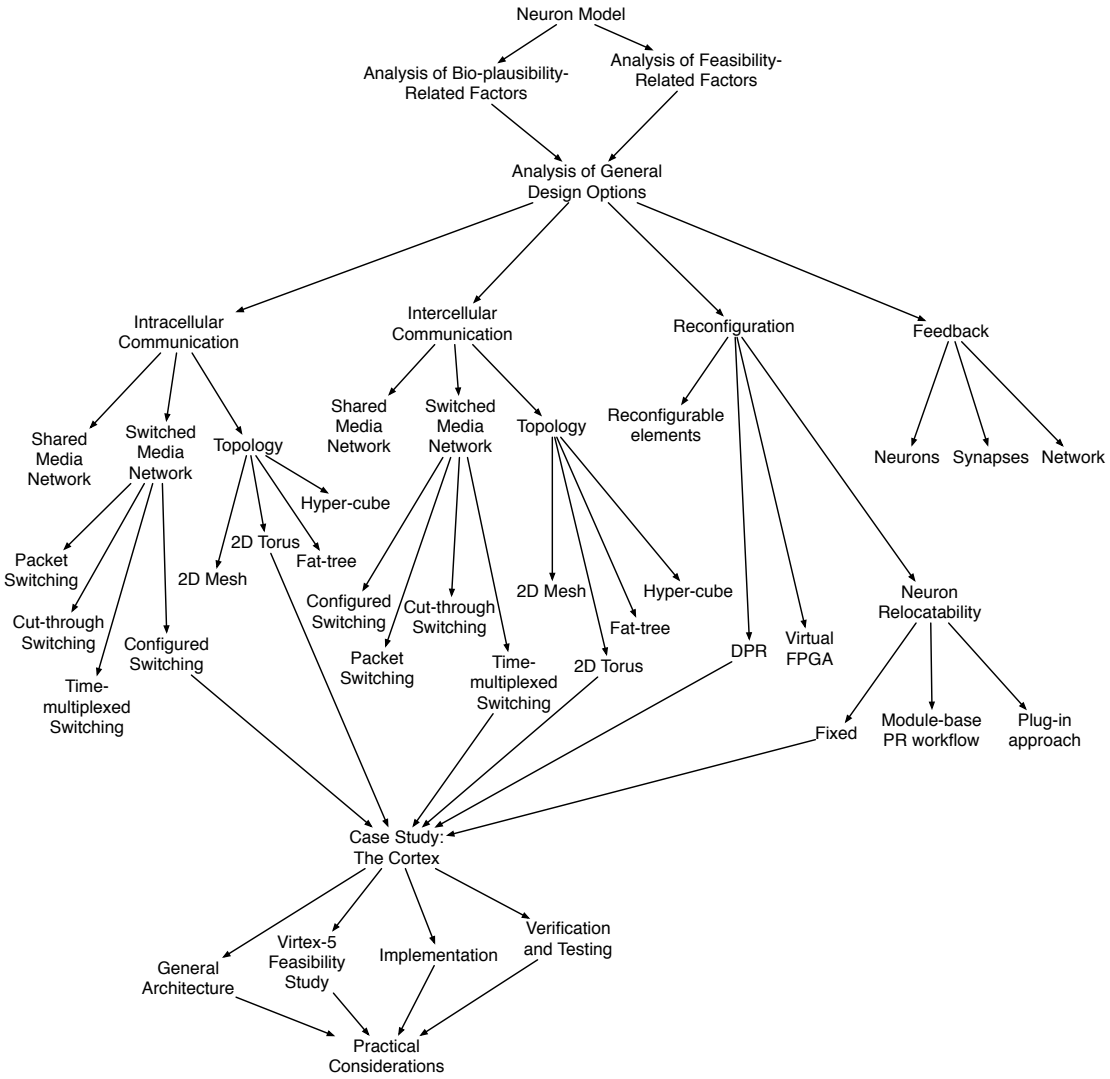


Figure 5.17: A graph of the investigations carried out in chapter 5 regarding the cortex model.

their requirements were discussed and different possible topologies and switching techniques, along with their limitations and trade-offs were investigated. Different reconfigurable elements available to be used in the cortex model were reviewed, and different options for reconfiguration mechanism, and their limitations and trade-offs were examined in section 5.2.3. Also different design options for feedback generation from the cortex was investigated. Based on the general insight from that analysis, and to further investigate the practical challenges, a new cortex model was designed, implemented, and verified in section 5.4. Practical issues, limitations, and tradeoff discovered during the detailed design, implementation and testing of the case study cortex model were also highlighted and discussed in the final section of this chapter. The Cortex model implemented here as a case study provides a basis for investigation of the evo-devo model in the next chapter.

Chapter 6

Evo-Devo Model

Biological brains are developed, maintained, and regulated by the chemistries and interactions between different types of molecules and atoms. These interactions and chemistries are governed by expression of different genes that are, in turn, regulated and adapted by Darwinian evolution. The combined mechanisms of genetic evolution, gene expression and regulation, protein interactions and interactions with the environment that finally produce the traits and behaviours in the phenotype are known as the evo-devo processes [298]. In this study, the evolutionary development of neural microcircuits in FPGA also requires similar processes that resemble biological evolution and development. Here, a combination of these bio-inspired processes is called an evo-devo model. Since these bio-inspired evo-devo processes control and regulate the neuron and cortex models in the system, their bio-plausibility can significantly affect the bio-plausibility of the whole system. Researchers have conjectured, argued, and in some cases proved that many of the desired properties such as adaptability, modularity, scalability, fault-tolerance, robustness, and even efficiency can emerge through such bio-plausible models (see section 2.5.4). Many of these properties have direct impact on the feasibility of the whole system. It is therefore the aim of this section to investigate the challenges, factors, trade-offs and constraints in the design and implementation of the bio-plausible evo-devo processes that can be feasibly used for development of neural microcircuits in FPGAs.

As in the previous chapters, first (in section 6.1), the general definitions of bio-plausibility and feasibility are translated into tangible design factors and constraints in the context of the evo-devo model. In section 6.2, using these factors that can affect the bio-plausibility and feasibility of the system, different general design options and approaches are investigated, and their trade-offs and limitations are highlighted in order to focus further investigations on the promising areas of the design space. To further investigate the practical challenges, design, implementation and testing of an example bio-plausible evo-devo model are presented as a case study in section 6.4. Practical limitations, challenges, and trade-off are highlighted in section 6.5.

6.1 General Design Factors

First in this section we focus on the design factors that can affect the bio-plausibility of the evo-devo model. Different aspects of bio-plausibility of the evo-devo model are highlighted here and their roles

in the bio-plausibility of the whole system are discussed. Secondly, we focus on the design factors that affect different aspects of the feasibility of the evo-devo model. These different aspects are generally the same feasibility measures that were defined in section 2.2.

6.1.1 Bio-plausibility Related Design Factors

Biological evolutionary neurodevelopmental processes are able to generate modular and hierarchical neural networks that show both regularity and randomness [404]. Sometimes evo-devo processes direct a single axon from one region of the brain to the other region in order to connect to a specific neuron [404]. More often, general connectivity of the brain regions are coded in the genome and more variations in connectivity can be seen across different individuals (even with identical genomes), and during individual's lifetime. Many connections are results of synaptogenesis guided by the network activity and in response to stimuli [404], and rewards or punishments during the lifetime of an individual. Developmental processes can detect redundant or faulty connections and cells and eliminate them [404]. Neurodevelopment can generate robust and intrinsically fault-tolerant neural microcircuits so that their performance degrades gracefully when they are subjected to noise, faults and damages. Moreover, biological neurodevelopment can regenerate and repair the damaged circuits by reallocation or generating new neurons and connections. Biological development is sensitive to environmental factors but robust to environmental noise [351, 404].

Biological evolution also shows a high level of evolvability [196, 165]. This is due to many different factors. The genotype-phenotype mapping in biological development is many-to-one leading to neutral mutations that increase evolvability. Different parts of the genome have different mutation rates and some parts are more robust to mutations. This is something that has been evolved itself through billion years. Biological evolution can result in new species with larger and more complex brains if the environment is demanding. Biological genomes are variable in length and complexity of both genotypes and phenotypes can increase gradually but significantly through generations. Modularity can be seen not only as cells, brain regions, clusters, organs, and limbs in the phenotypes, but also as genes, gene clusters, chromosomes, and genetic pathways in the genotypes. The effective fitness of an individual in biological evolution is the result of very fluid and dynamic interactions of individuals with the environment, that unfold new challenges and opportunities for species as they evolve. This can be seen, for example in the coevolution of predator and prey species. Sometimes evolution finds a niche of resources in the environment and a new species emerge to exploit it. In biological evolution, completely different species, or geographically separate species, usually do not crossbreed, which brings diversity to the ecosystem. However, sometimes symbiosis can allow different species to cooperate and probably merge their genomes creating more complex phenotypes and genotypes.

A bio-plausible approach assumes that many of these properties and features can be achieved in artificial evo-devo models by increasing the structural accuracies of the models meaning that the internal mechanisms of the models to be as close as possible to the underlying mechanisms in the biology. To be able to asses bio-plausibility of different evo-devo models, the underlying mechanisms and structures of the biological evolution and development are briefly reviewed here [165]. Here, evo-devo processes

are modelled in a hierarchy of systems. The boundary of a system is usually defined around a cluster of subsystems, which appear to have more interactions between themselves and inside the boundary compared to their interactions with entities outside the system and across the boundary.

We can start from ecosystem as the highest level system that includes all the interactions, although it is not a closed system and is itself interacting with the rest of the universe. An ecosystem can be thought of as a system comprising many many species that interact with each other directly or through the effects they have on the environment. These interactions may appear as cooperative or competitive. These species consist of many populations (usually geographically separated) that have much more interactions inside them than with each other.

These populations consist of individuals embodied in the environment that, apart from interacting with the environment and other species, interact with each other in many forms including competition for shared resources, cooperation as groups and communities, and above all, sexual reproduction. Their cooperative and competitive interactions with each other, with other species, and environment create a selection that might be in favour of one species, population, or individual, something that is modelled as environmental selection [165]. Reproductive selection and mating, on the other hand, can include other factors that might be evolved as traits and preferences that can direct the evolution of a species. These preferences can create internal subgroups inside a population that inbreed or crossbreed allowing regulation of the diversity and fitness of the populations. This is modelled as sexual selection [165]. Reproduction with some variation is the fundamental mechanism of evolution. Reproduction is based on the replication of individual's genome, a number of chromosomes, that store the genetic heritage of each individual and are replicated during reproduction, albeit with some random noise, known as mutations.

Chromosomes are DNA (or RNA) strings of a four-letter alphabet, each letter being implemented by one of the bases represented by letters A, C, G, and T. Asexual reproduction creates an imperfect copy of these chromosomes with some mutations in the chromosome of the offspring [165]. Sexual reproduction not only replicates the parents‘ genome, but also randomly recombines two different copies from the parents. This recombination, is performed by matching the similar chromosomes of the parents and using substrings from each copy to construct the offspring's chromosome [165]. This random process, known as crossover, involves switching between two copies at a number of places along the length of the chromosomes where the offspring's chromosome switches from one parent's copy to the other's [165]. The matching process is always imperfect and can result in deletion, extra copies of the substrings, shifts and so on [165].

Some parts of each chromosome that can be transcribed into RNA and proteins are known as coding sequences. Each group of three letters in the coding sequences can transcribe into a molecule which will integrate with other molecules constructing even larger molecules known as proteins [165]. These proteins, depending on their sequence of atoms and environmental conditions fold into specific shapes and can interact and integrate with other proteins and molecules resulting in structures that build the cells and body of the individual. These structures and the interaction of the proteins with each other and environment is the basis of the functioning of the cells and the whole organism. These proteins and

molecules also interact with the chromosomes and get involved in the transcoding of the chromosome. They may promote or suppress the process of transcoding depending on the neighbouring substrings in a piece of chromosome [165]. Each piece of chromosome that is involved in the transcoding of a protein or a piece of protein is known as a gene. Therefore, transcoding of each gene, known as gene expression, produces a protein, and each protein can interact with genes and affect their expression. Proteins also interact with each other and with environment inside and outside of a cell. These proteins and their interactions form the structure and function of the cells and bodies of the individuals. The intricate gene-protein and protein-protein interactions can form very complex networks known as Gene-Regulatory Networks (GRNs) [165]. This mechanism brings both coding and non-coding sequences of the genome into play. Even those segments of a chromosome that are never involved in anything in one individual, have played a role in one of individual's ancestors or might one day play a role in one of its descendants due to random mutations, recombinations, and the dynamic environment.

Multicellular individuals also have interactions between their cells. Some molecules and proteins can move from one cell to the other and interact with the proteins and genes in the other cell. This process, known as signalling, allows cells to know their position in space and differentiate accordingly to work together to create much more complex structures and functions [404]. Cells can stick together, push each other, bend, or twist to form shapes, tissues, organs that work together providing the individual with higher-level functions and structures [404]. These chemical signals not only provide positional and other information to the cells but also can guide the neurites to grow towards their target cells [404], or signal cells to duplicate (mitosis), or die (apoptosis). All these interactions create a complex system that allows properties such as adaptability, fault-tolerance, robustness, regeneration, scalability and modularity to emerge. Evo-devo processes have been shown to be responsible, directly or indirectly in all of these properties in the biological systems [211, 404, 165].

Looking at such a complex system from bottom upwards, we can clearly see a hierarchy of modules. Interacting atoms construct the molecules. Long and complex molecules that form genes and proteins interact with each others inside cells. Interacting cells form tissues and organs that make individual bodies. Individuals interact with each others and with non-living entities in an ecosystem. Interacting individuals form groups and populations of species that also interact with each other and their environment. A bio-inspired evo-devo model needs to incorporate enough levels of this hierarchy with sufficient detail. Ignoring some levels or extreme abstraction of interactions may deprive the model from some emergent properties. On the other hand, including all levels and details is not feasible as it requires massive energy, time, and computation power. Design factors related to the feasibility of the evo-devo model are discussed in the next section.

With this biological background we can expect that a bio-plausible evo-devo model needs to have similar counterparts for most of these functions and structures, counterparts for proteins that affect the functioning, and construct the structure of the neural microcircuits, and gene-regulatory networks with protein-protein and gene-protein interactions that create a dynamical system regulating the proteins, intercellular signalling including the positional information, genes, chromosomes, genomes, individuals,

populations and even species and their interactions.

6.1.2 Feasibility Related Design Factors

Feasibility of the whole system is affected by a number of factors that can impact the performance, hardware cost, scalability, reliability, complexity and availability of the system. Here we focus on every one of these groups of factors in the design and implementation of the evo-devo model and their possible effects on different aspects of the system feasibility.

Factors Affecting the Performance

Performance of the system depends on the number of evaluations the evolutionary model needs to evolve a solution. Each evaluation requires development of the individual before and during the neural simulation. Therefore the time that the system needs to evaluate an individual is a mixture of the development time, the cortex reconfiguration time and neural simulation time. In the simplest form an individual is first developed, then the cortex is reconfigured accordingly, and then simulation is carried out to evaluate the individual. In case of activity-dependent development, the individual first goes through an initial stage of development until it is ready for the initial reconfiguration and some simulation. Then during the simulation, developmental process needs to be executed concurrently to reconfigure the neural microcircuits every now and then. This increases the total evaluation time of each individual. It is therefore desired to minimise both the number of evaluations needed for evolving an acceptable solution, and the total development and reconfiguration time.

Factors Affecting the Hardware Cost

The evo-devo processes may need dedicated hardware resources for their execution or may necessitate adding special hardware to the cortex model. These processes may also share part of the hardware resources already available in the system or may be partially migrated to software modules running on the embedded system processor or a PC connected to the FPGA. In either case, it is desired to minimise the hardware overhead of the evolutionary and developmental processes to minimise the hardware cost of the whole system.

Factors Affecting the Scalability

The evolutionary and developmental processes are required to work for smaller or larger cortex sizes that might be implemented on a single or multiple FPGA devices. They must also allow scalability of the problem in terms of the complexity of the problem and the size of the input/output vectors and size of the stimuli dataset used for evaluation.

Factors Affecting the Reliability

The evo-devo processes must not only be reliable in the sense that they do not decrease the reliability of the system but also they are expected to allow fault-tolerance, robustness, regeneration and self-repair to emerge, which can increase the reliability of the whole system. This type of reliable neural microcircuits can be very useful when developed in a huge cortex with large numbers of neurons and glial cells. Fabrication of such a huge cortex in a very large VLSI chip involves low yield factors or high number of faulty cells in the cortex. SEUs (single-event upsets) and unit failures are more common in such large

systems. A fault-tolerant and robust cortex can resolve these problems. Although evolutionary process will be able to evolve networks that are intrinsically robust to loss of nodes and links, a bio-plausible developmental process can also contribute to fault-tolerance and robustness of the system. This can be achieved, for example, by regeneration or by neurites avoiding the faulty cells. Errors and faulty cells can be detected by the activity feedback information from the cortex (as explained in chapter 5 or by rather traditional methods such as post-fabrication test, POST (Power-On Self-Test), BIST (Built-In Self-Test), DMR (Dual Modular Redundancy), or TMR (Triple Modular Redundancy) [313] or by more innovative and bio-plausible methods such as artificial immune systems [365].

Factors Affecting the Complexity

There is no doubt that incorporating the evo-devo processes into the system will affect the complexity of the design and testing of the whole system. It is always desired to minimise the time and complexity of the system design and testing. This is particularly important as the testing of the modules related to these processes may need running the whole system. Therefore a manageable, modular, and structural design is required that simplifies the design and allows separate testing of each module before integration to the rest of the system.

6.2 General Design Options

The tangible factors and constraints that can affect the feasibility and bio-plausibility of the system analysed in the previous sections are summarised in table 6.1. Based on these factors and constraints, now, it is possible to explore different general design approaches and options to focus on the promising methods for further investigation. Looking at table 6.1 , it is possible to classify different functions that the evo-devo model needs to implement as follows:

1. A dynamical system (gene-regulatory network) that organises the structure and regulates the parameters of the neural microcircuits and receives feedback from it.

2. An evolvable genetic representation of this dynamical system with an evolvable mapping from genome to the description of the dynamical system.

3. An evolutionary algorithm including a selection mechanism that maintains a population and selects potential parents from the population allowing speciation and population diversity, and recombination and mutation operators that can reproduce new offspring genomes with required genetic representation.

4. An application-specific fitness function that evaluates each new individual microcircuit and passes its fitness value to the evolutionary algorithm.

Figure 6.1 shows how these different functions work together in the evo-devo model and how they can be divided into separate developmental and evolutionary processes. In this section, different general approaches and options for design and implementation of the three former functions are discussed. As fitness evaluation depends on the specific application of the whole system it is discussed in the next chapter when system integration is investigated.

Table 6.1: A summary of the tangible design factors and constraints in the design and implementation of the evo-devo model that can affect the bio-plausibility and feasibility of the system.

Bio-plausibility Related Design Factors	Feasibility Related Design Factors
Proteins that regulate the functioning, and construct the structure of the neural microcircuits	Evolution speed (minimising the number of evaluations needed for evolving an acceptable solution)
Gene-regulatory networks with protein-protein and gene-protein interactions that create a rich and evolvable dynamical system regulating the proteins concentrations receiving information from environment and neural microcircuit feedback	Development time (minimising the total development time and number of reconfigurations needed during activity dependent development)
Chemical signals providing positional information and allowing differentiation, division, apoptosis, and guiding neurite growth	Compactness (Minimising the hardware overhead of the evolutionary and developmental processes)
Genes with adjustable robustness to mutations	Scalable to a smaller or larger cortex, more or less number of inputs and outputs, and a simpler or more complex problem
Variable-length chromosomes, and genomes with crossover and mutations that allow deletion, duplication, and modification of the genetic information	Emergence of fault-tolerance, robustness, regeneration and self-repair without impacting the reliability of the other parts of the system
Population of individuals with both environmental and sexual selection	Simple, manageable, modular and structured design
Interactions inside and between populations and species that allow both competitive speciation and cooperative symbiosis	

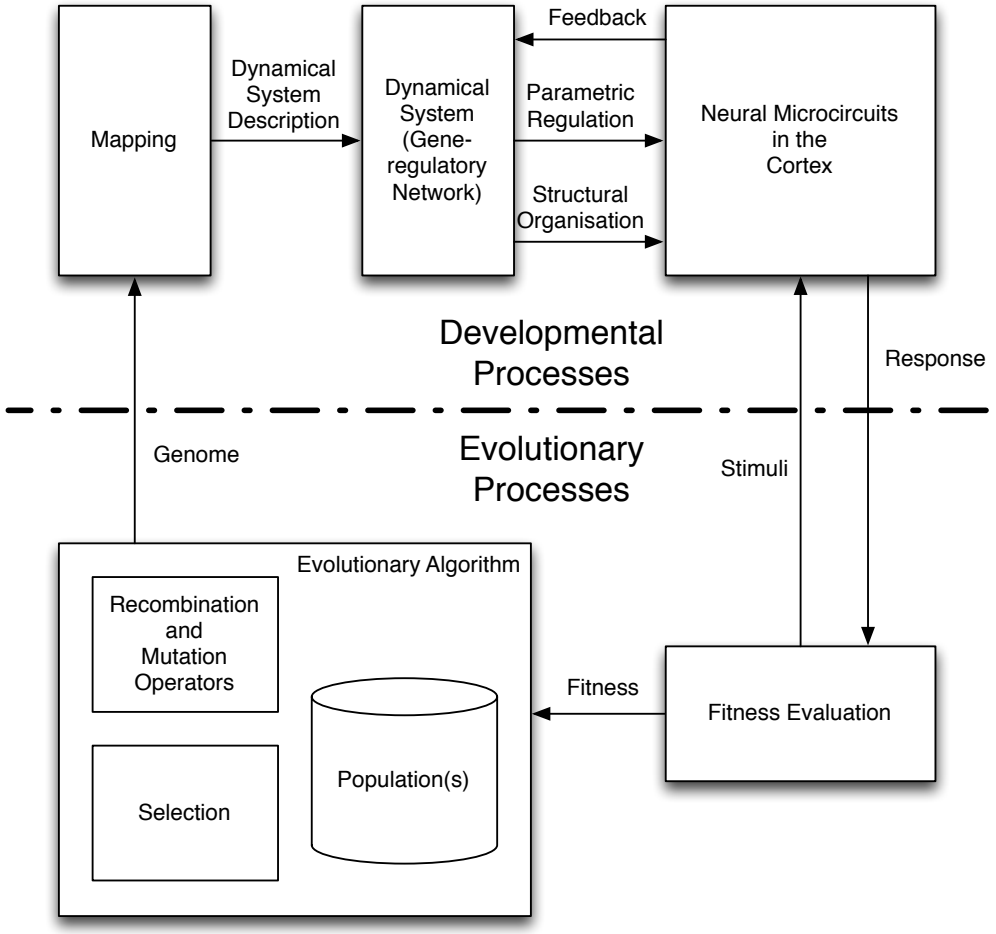


Figure 6.1: Different functions of the evo-devo model and their interactions. Evolutionary and developmental processes are separated from each other. Both processes need to share the same genetic representation.

6.2.1 Dynamical System (Gene-Regulatory Network)

A dynamical system is needed in the heart of the developmental system that models the biological gene-regulatory network. This dynamical system receives feedback data about the health and activity of the soma, glial, and IO cells in the Cortex. These are local information that change through the simulation time. The dynamical system is required to produce two types of signals. Structural signals control the differentiation of the cells, growth, death, retraction, and trimming of the axons and dendrites, and also formation and elimination of the synapses (synaptogenesis). Regulatory signals control the soma cell and synapse unit parameters such as, reset potentials and synaptic weights. These are also local signals that might change through the simulation time.

If the current state of each cell located at (x, y) at time t is represented with a vector $\vec{S}^t_{x,y}$, the dynamical system can be formulated as a function f of the current state vector of a cell and its neighbouring cells states ($\vec{S}^t_{N(x,y)}$, where $N(x, y)$ is the set of cells in the geometrical neighbourhood of (x, y) in the substrate), and local feedback ($\vec{F}^t_{x,y}$) with an equation of the form:

$$\vec{S}^{t+\Delta t}_{x,y} = f(\vec{S}^t_{x,y}, \vec{S}^t_{N(x,y)}, \vec{F}^t_{x,y}) \tag{6.1}$$

.

Locality of these signals, their time dependence, and direct imitation of the biology may lead a designer to a multicellular, distributed, iterative dynamical system as proposed in [311]. However, as examined in the following, it might be possible to have an abstracted model of multicellular time-dependent development.

Abstracted models

As discussed in [347] comprehensively, it is possible to evolve one or a set of related functions over a Cartesian space to produce the spatial patterns needed for organisation and regulation of a phenotype without having a multicellular time-dependent development with all the chemical signals and local interactions. An example of such abstracted models of development is HyperNEAT [349] and its extensions. In the original HyperNEAT method, the structural organisation and parametric regulation (synaptic weights) of an ANN is specified statically by an evolvable function over a 4-dimensional space of the connections between neurons that are located in a 2-dimensional substrate.

Instead of using local intercellular signalling to create the positional information such as the anterior-posterior, and dorsal-ventral axes [404], a functional description starts from a predefined Cartesian space. This abstraction can save significant computation power that is needed both for development of those positional information signals, and for evolution of the genes that control them. These models abstract the dynamical system from time and local interactions into a much simpler static function of the form:

$$\vec{S}_{x,y} = f(x, y). \tag{6.2}$$

Stanley suggested, in [347], to add necessary feedback information from the development environment as inputs of this functional description to make the abstracted model respond to these factors. These factors can be static ($\vec{F}_{x,y}$) or time-dependent ($\vec{F}^t_{x,y}$) that gives a static or dynamic function of the forms:

$$\vec{S}_{x,y} = f(x, y, \vec{F}_{x,y}) \tag{6.3}$$

and

$$\vec{S}^t_{x,y} = f(x, y, \vec{F}^t_{x,y}) \tag{6.4}$$

respectively. The time-dependent version of the model, requires re-evaluation of the function every time that the feedback data is updated. In [347], Stanley also suggested to add the necessary local states to the input of the functional description to create an adaptive system of the form:

$$\vec{S}^{t+\Delta t}_{x,y} = f(x, y, \vec{S}^t_{x,y}, \vec{F}^t_{x,y}). \tag{6.5}$$

For example, in [304], Risi and Stanley showed that it is possible to evolve an adaptive ANN that updates its synaptic weights ($\vec{S}^t_{x,y} = w_{ij}$) using an evolved function of the position of the pre and post synaptic neurons ($x, y = (x_i, y_i, x_j, y_j)$) and their activities ($\vec{F}^t_{x,y} = (a_i, a_j)$).

The only difference between this later version of the abstracted model (equation 6.5) and the original multicellular dynamic system (equation 6.1) is the local interactions between cells. Such abstracted models assume that these local interaction are only necessary for producing positional information that is already directly available to the functional description in the abstracted model. However, apart from being less bio-plausible than using local interactions, it is not clear how scalable, fault-tolerant, and robust such abstract models can be in response to run-time changes in the size, geometry and connectivity of the problem, substrate, inputs, or outputs. For example, a set of faulty inputs, links or nodes in the substrate can disturb the local interactions in a multicellular model, which can automatically warp the virtual space of the substrate around the damaged area. In contrast, the abstracted model that relies on the fixed Cartesian coordinates, needs to evolve special mechanisms to cope with such situations. Adding more resources and scaling up the problem size (such as adding an extra chip for the Cortex and doubling the number of inputs) requires such abstract models to be already evolved for the larger cortex or have special provisions to cope with a larger substrate. However, biological evidence shows that local interactions are a very effective means for scalability of complex structures. It appears that the use of local interactions can bring intrinsic scalability, fault-tolerance, and robustness to artificial development that otherwise may require special regenerative mechanisms to be evolved separately in abstracted models [78].

To regulate the placement and density of the neurons in the substrate, these abstract models need to sample a function for every possible position in the substrate and based on the value of the function (or its variance as shown in [305]) decide about the position of the cells and their parameters. This requires time consuming computations (at least for one initial iteration of the development) that these abstract models are intended to avoid. Moreover, if an adaptive placement or regeneration is desired, these computations need to be repeated.

Such abstract models not only save the initial time and computational power that is needed for evolving the required genetic material and developing the positional information, but also they save all the computational power that is needed for sustained local interactions during development. As it is discussed in the following, multicellular models are computationally more complicated and expensive but can also use some of these tricks to save on some computations.

Multicellular Models

Multicellular models or cell chemistry models are based on two types of primary interactions: gene expression (gene-protein interactions) and chemical signalling (protein-protein interactions). Proteins in the cell can have effects on the expression or suppression of genes [404, 211, 165]. Also, when a gene is expressed, its protein products are synthesised and added to the proteins in the cell, increasing the concentration of that protein in the cell. Furthermore, these proteins can interact with each other. One of the very important types of the protein-protein interactions is that some proteins on the surface of the cell membrane are able to pass other specific types of proteins in or out of the cell [404, 211]. This allows some proteins to travel longer distances outside of the cell and into other cells, interacting with their genes and internal proteins. Proteins also decay through time, which allows these protein concentration

to work as time dependent signals [404, 211].

If concentration of each type of protein in a cell is represented by one component of the cell state vector, the gene-protein interactions, by their own, can be formulated as a dynamical system of the form:

$$\vec{S}_{x,y}^{t+\Delta t} = f(\vec{S}_{x,y}^{t}). \tag{6.6}$$

As the concentration of some of the proteins in the cell depends on their concentrations outside of the cell, which in turn depends on that value inside the other neighbouring cells, the above equation turns into an equation such as:

$$\vec{S}_{x,y}^{t+\Delta t} = f(\vec{S}_{x,y}^{t}) + g(\vec{S}_{N(x,y)}^{t}). \tag{6.7}$$

Adding the effect of the local feedback ($\vec{F}_{x,y}^{t}$), arrives at an equation very similar to the equation from the original general dynamical system (equation 6.1):

$$\vec{S}_{x,y}^{t+\Delta t} = f(\vec{S}_{x,y}^{t}, \vec{F}_{x,y}^{t}) + g(\vec{S}_{N(x,y)}^{t}). \tag{6.8}$$

Therefore, there are two functions that are needed to be described by the genome and calculated for each cell through evolution. One is a gene expression function $f(\vec{S}_{x,y}^{t}, \vec{F}_{x,y}^{t})$ of the current state and feedback in the cell, and the second one is a protein diffusion function $g(\vec{S}_{N(x,y)}^{t})$ of the state of the neighbouring cells or area. Looking at a few different models of the protein diffusion in literature (see section 2.5.3), shows how researchers were trying to unburden the developmental model from this repetitive computation. Some of them (in time-independent models) assume a time-independent function of distance (equation 2.15 and 2.16) abstracting both time and discrete nature of the cells. This way calculation of the diffusion function is of complexity order of $\mathcal{O}(N_u N_c N_p N_s)$ where N_u, N_c, N_p and N_s are representing the number of updates during development, number of cells, number of proteins (that can travel beyond a cell membrane), and number of sources respectively. This will help to limit the computation to only the source cells that produce a protein and only update their contribution when the protein concentration has changed at the source. More bio-plausible models (for example equation 2.17) use equal iterative contributions from each neighbouring cell (according to the connectivity topology of the substrate) that take time into account. The computational complexity of these bio-plausible models is $\mathcal{O}(C N_t N_c^2 N_p)$ where C is the number of neighbours depending on the topology of the substrate, and N_t is the number of development cycles. These models have to perform the computations for all the cells, all potential sources in all cells and at every development iteration. Some researchers used simplified models such as equation 2.18 that reduces the hardware cost of calculating the function by removing addition and division operations.

Apart from computational cost of the diffusion function, the system needs many evolutionary iterations until the genetic material for producing the positional information in the substrate emerges. One way to skip this step is to start the evolution with a seed population with pre-evolved or even hand designed genomes that produce the necessary positional information in the first generation and during the first iterations of the development. Another way is to start the development only with some maternal factors. Maternal factors are proteins that have initial concentrations in the cells or even gradients in the

substrate when a phenotype starts to develop [404]. This can give the developmental processes the positional information right away at the very first iteration without evolution. However, this technique allows evolution to change the dynamics of these maternal factors through the development of the individual rather than using static positional information as in abstract models.

Based on the local interactions between cells, multicellular models are able to use bio-plausible methods of cell differentiation such as lateral inhibition [404] for regulating the position and density of the neurons in a substrate. The same mechanism is able to regenerate a new neuron if the old one has died. In contrast, in the abstract models, these functions need special attention and necessitate iterative processing through time.

6.2.2 Genetic Representation and Mapping

The functions used in the dynamical system or gene-regulatory network of the evo-devo model need to be encoded in a genome. This requires a representation protocol that is both evolvable and flexible enough to represent the required functions needed to develop desired phenotypes. Here, different general options for the genetic representation are discussed. A brief but general review of the representations used in the field of evolutionary computing and artificial life is used here to investigate possible options. Based on the bio-plausibility of the multicellular models, all the genetic representations are evaluated here in the context of a multicellular developmental system.

Evolutionary algorithms used for evolving functions can be classified in two general groups. The first group consist of those that assume a very fixed formulation for the function and only evolve the parameters of the function. Evolving the coefficients of a degree-n polynomial, evolving a Bezier curve, a Fourier series, or a wavelet transform are all examples of different methods in this class. All the methods in this class use a fixed structure for the function and perform a parametric evolutionary search. The second class is the group of methods that can evolve the structure of the function as well as the parameters.The rich and diverse structural complexity of the biological gene-regulatory networks leads us to investigate the second group as a promising bio-plausible option.

Two fundamental different evolutionary methods for evolving the structure of the functions are based on tree representations and directed-graph based representations. GP (Genetic Programming - Koza [205, 206]) and CGP (Cartesian Genetic Programming - Miller [266, 262]) are very well-known representatives of these two approaches. GP uses a tree structure while CGP uses a directed graph for representation of the functions. Looking at the natural and biological structures and specifically GRNs, it is evident that using directed graph is structurally more accurate than trees. Structural accuracy is one of the main definitions of the bio-plausibility in this study as discussed in section 2.1. A tree structure appears to be more suitable for a mathematical symbolic representation when human understanding of the structure is desired. Therefore, in the following sections we focus on investigation of graph-based representations for the dynamical system in the developmental model. Figure 6.2 shows a taxonomy of a few evolutionary algorithms used for evolving functions with emphasis on methods that use directed graphs for genetic representation of the function. Although, there exist a spectrum of different representations, we try to capture this diversity by examining a few representatives of the genetic representations

that can be used to evolve dynamical systems.

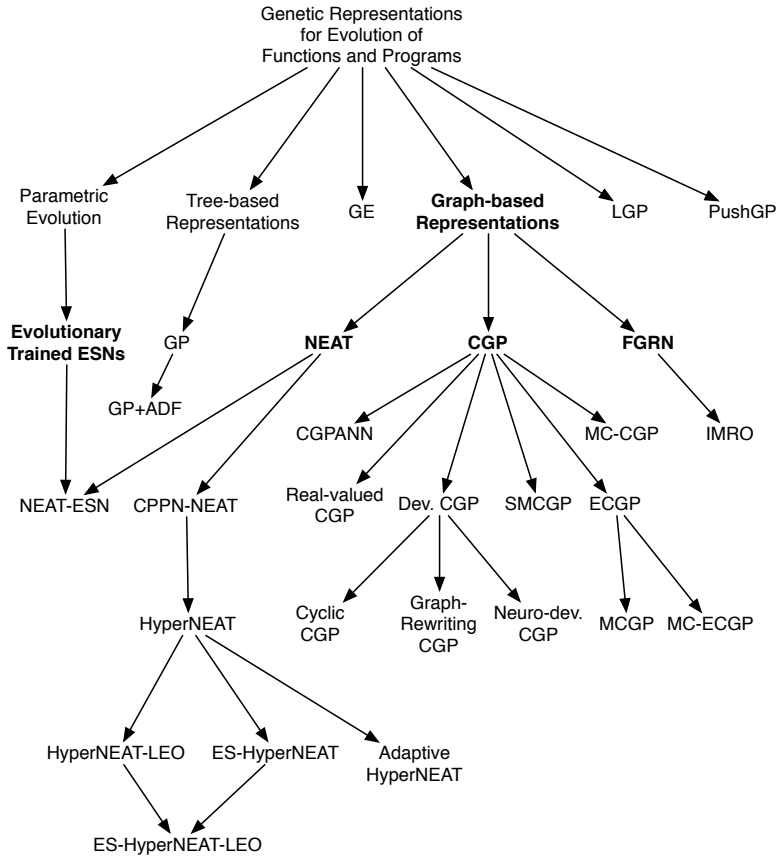


Figure 6.2: A taxonomy of a few evolutionary algorithms used for evolving functions with focus on methods using directed graphs for genetic representation.

CGP

CGP is the foundation of a successful lineage of other genetic programming methods that has been used for evolving functions, dynamical systems, gene regulatory networks, neural networks and many many other applications [262]. It is a generic, simple, flexible, relatively bio-plausible and computationally low-cost [265] method for genetic representation of functions as directed graphs. Even in its original and simplest version it can evolve a set of functions with any number of inputs based on mathematical, logical or any other type of primitive operators. It allows non-coding genes and neutral mutations that contribute to the evolvability of this method. Although the original version is limited to fixed length chromosomes and directed acyclic graphs, with no crossover, in its abstract form, it can support variable length chromosomes [262], crossover [262], cyclic graphs (Cyclic CGP [262]), Modules (ECGP and MCGP [262]) , self-modification (SMCGP [262]), and much more [262].

CGP and its more advanced forms has been used effectively in evolution of different functions and controllers. Particularly, there were used successfully in evolving dynamical systems for developing robust, scalable, and fault-tolerant boolean and electronic circuits, neural networks, bio-plausible neural microcircuits, and 2D shapes (flags) [420, 192, 261, 224, 263, 223, 260]. The genetic representation of the original version consists of integer-valued genes that describe the type of the primitive operator for

each node in the graph and indices of the nodes connected to the inputs of this node. Same chromosome can be used to generate many outputs if they are related. Otherwise, different chromosomes can be used to evolve completely separate functions in the same individual. This original representation uses only mutations and a very small elitist population. Here, when referring to CGP in comparison with other techniques, only this original and simplest form of CGP with Boolean functions as primitives is intended.

Using a fixed grid of gene indices poses some limitations on the original representation that makes applying crossover operations difficult or disruptive. However, different methods and techniques has been proposed to tackle this limitation. One of these methods is the historical marking of the genes as used in NEAT.

NEAT

NEAT (Neuro-Evolution of Augmented Topologies) was originally used for evolution of ANNs. While it adopts the same fundamental graph-based representation of CGP, it employs a number of techniques to improve the evolvability of the original representation. It uses a separate chromosome for describing the nodes and their indices to allow complexification of the neural networks with variable length genomes. Moreover, it adds historical markings to each new node or connection that show the chronological order of the new genes appearing in the gene pool. These historical markings allow matching related genes easily during crossover and also measuring the similarity of the genomes for speciation. NEAT starts from a minimal uniform seed population and progressively evolves toward increasingly more complex solutions. Using sigmoid neurons with real-valued outputs makes NEAT a more bio-plausible option for modelling GRNs than CGP with boolean functions. However, NEAT uses some bio-plausible and some implausible but useful techniques for efficient crossover, speciation, and fitness sharing and has shown great success and flexibility in tackling different problems. Although it is not usually used for evolving dynamical systems for developmental models some works has been reported in that line [79]. A different form of NEAT called CPPN-NEAT is much more applied to generative models.

CPPN

A Compositional Pattern Producing Network (CPPN) is a network similar to an ANN but with a more diverse set of transfer functions. ANNs are limited to transfer functions such as sigmoid or hyperbolic tangent, while CPPNs can use different functions such as absolute value, Gaussian, sine, cosine and so on as the transfer function of each node in the network. CPPN-NEAT uses the NEAT evolutionary processes and genetic representation to quickly evolve patterns that resemble the morphogens from a developmental process [347]. Since CPPN-NEAT uses complex functions that produce symmetry or repetition, it can quickly evolve patterns that can be used as a function for directly describing a phenotype or as the dynamical system of a developmental model. From the bio-plausibility point of view however, using course-grain functions such as Gaussian or Sine appears more as an efficiency trick that abstracts a great deal of details out of biological gene-regulatory networks. It appears that CPPN-NEAT to be highly optimised for direct generation of the morphogens in abstract models of development rather than a general method for evolving GRNs for multicellular or iterative developmental models. CPPN-NEAT

has been used in such generative models for evolving ANNs in HyperNEAT and its extensions (see section 2.5.5). However, they are not immediately compatible with the requirement of a hardware-based neural network in FPGA and specifically the Cortex model of chapter 5. This is due to the fact that they all specify the links and corresponding synaptic weights between neurons without dealing with the routing problem. However, it would be possible to use the general idea of using CPPNs or RNNs as a function that generates the local properties of the phenotype or a function for the dynamical system governing the developmental process. The basic similarity between GRNs and RNNs lead us to look at other methods for evolving RNNs, specifically Echo State Networks.

ESN

Another method to evolve a dynamical system being used in the literature is to evolve ESNs (Echo State Networks). ESNs are random fixed recurrent neural networks (originally of leaky integrator neurons) with only an output layer of neurons being trained in a supervised manner (see section 2.4.4). As the structure and weights of the recurrent part of the network is randomly generated, evolving only the output weights is computationally less expensive. In [63] genetic representation and evolutionary algorithm of NEAT was used to evolve ESNs themselves with supervised and reinforcement learning to tackle complex control tasks. Studies on evolution of ESNs is still very immature, and their application for evolving GRNs is limited to works such as [79, 80] where ESNs were used as a dynamical system for development of 2D shapes. Compared to NEAT, ESNs showed very competitive results in evolving robust and scalable development of 2D shapes with self-repair. However, in one of these works, only the output weights were evolved using an evolutionary strategies method. It was shown that the decision mechanism for termination of the development has a critical effect on the robustness and fault-tolerance of the developmental process. ESNs use a sigmoid transfer function and leaky integrator neurons similar to those appear in biological GRNs, which makes it slightly more bio-plausible than methods such as NEAT. Although there are many similarities between GRNs and ESNs, the random nature of the main part of the network, while the output weights need tuning, is somehow not very bio-plausible. Therefore the more bio-plausible approach of evolving the whole ESNs or other RNNs currently rely on genetic representations such as NEAT and are subject to the same bio-plausibility issue of their underlying representations. More bio-plausible models such as Fractal Gene-Regulatory Networks exist that can address the biologically implausible aspects of methods such as NEAT and CGP (*e.g.*historical markings, fixed indices, boolean functions) and offer more richness and dynamism.

Fractal Gene-Regulatory Networks

Fractal Proteins or Fractal Gene-Regulatory Networks (FGRN) [25, 24, 27, 26] are artificial bio-inspired models of gene regulatory networks utilising fractals to model the protein folding processes allowing complex protein-protein and gene-protein interactions. It is based on the same fundamental representation of functions as directed graphs. But instead of using fixed or historic indices for describing the node connections, it uses a dynamic pattern matching of the protein shapes. Protein and gene promoter shapes are sampled subsets ($n \times n$-pixels square windows of size z centred at x, y) of the Mandelbrot set, that interact with each other. Both fractal proteins and gene promoters are described only by a (x, y, z)

triplet. Each sampled pixel of the Mandelbrot set is a number between 0 and 200. Figure 6.3 shows an example of a protein shape and the subset of Mandelbrot set it was sampled from. Existing proteins (proteins with a non-zero concentration) interact with each other using a maximum function resulting in a merged protein consisting of pixels with maximum values over all merging proteins [26]. This can be expressed by:

$$Vm_i = \max_{j, C_j \neq 0} V_i^j \quad \text{for } i = 1..n^2 \tag{6.9}$$

where n^2 is the number of pixels, V_i^j and Vm_i are value of pixel i in the shape of protein j and merged protein shape respectively, and C_j is the concentration of protein j. Non-zero pixels of the merged protein then interact with the non-zero pixels of promoter shape of each gene j resulting in a total absolute difference δ^j [26]:

$$\delta^j = \sum_{i=1, Vp_i^j \neq 0, Vm_i \neq 0}^{n^2} |Vm_i - Vp_i^j| \tag{6.10}$$

where Vp_i^j is the value of pixel i in promoter shape of gene j. The probability of expression of gene j is then calculated using a sigmoid function of form [26]:

$$P(E^j | \delta^j, T_A^j) = \begin{cases} \frac{1+tanh\left(S_c(T_A^j + \delta^j - T_c)\right)}{2} & \text{if } T_A^j < 0 \\ \frac{1+tanh\left(S_c(T_A^j - \delta^j - T_c)\right)}{2} & \text{if } T_A^j \geq 0 \end{cases} \tag{6.11}$$

where T_A^j is the affinity threshold of the gene j (appended to the gene promoter), S_c and T_c are two constants that control the threshold position and sharpness of the sigmoid function (normally set to 0.02 and 50 [26]). When gene j is expressed, the concentration of the protein it is coding (C_j) is increased (or decreased for negative values) by [26]:

$$\sigma = A_c \cdot c^j \cdot tanh(\frac{c^j + T_C^j}{W_c}) \tag{6.12}$$

where T_C^j is the concentration threshold of the gene j (appended to the gene promoter), A_c and W_c are two constants (normally set to 0.5 and 30, controlling the amplitude and sharpness of the sigmoid function respectively [26]), and c^j (total concentration seen by promoter of the gene) is calculated using [26]:

$$c^j = \frac{\sum_{i=1, Vp_i^j \neq 0}^{n^2} C_{\text{argmax}_k V_i^k}}{N} \tag{6.13}$$

where N is the number of non-zero pixels in the promoter shape of gene j. At each development step, concentration of all proteins are updated using [26]:

$$C_j^* = C_j - \frac{C_j}{D_c} - 0.2 \quad \text{for all } j \tag{6.14}$$

where $C_j^*$is the updated concentration of protein j and D_c is the decay constant (normally set to 5 [26]). The last term in equation 6.14 is to ensure that the protein concentration can drop to zero instead of tending towards zero indefinitely.

The genome consists of a single variable-length chromosome of genes with 9 fields of the following form:

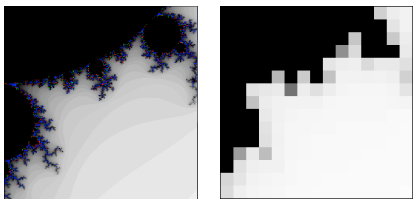
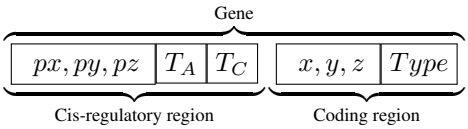


Figure 6.3: Left: A square subset of Mandelbrot set used for a protein shape. Right: The 15x15 protein shape sampled from the subset.



The px, py, pz triplet specify the fractal promoter shape of the gene. These three, along with the affinity threshold (T_A) and concentration threshold (T_C) form the cis-regulatory region of the gene. The coding region of the gene consists of another triplet (x, y, z) coding the shape of the protein that gene is coding, and the protein type field. The protein type is a binary string determining what combination of roles this protein can play in the system. Four major protein types are: regulatory, behavioural, environment, receptor. A protein can be any combination of these types. However, the cis-regulatory region of a gene coding an environment or receptor protein is ignored and that gene is always expressed with the highest protein concentration (200). Therefore, it cannot effectively play the role of a regulatory or behavioural gene. Environment proteins are all merged and then masked by the zero-valued pixels of the receptor protein (only one receptor protein is allowed) before contributing to the merged protein. Environment proteins can be used for initialising the development (similar to maternal factors) or as inputs to the development process. Depending on the application, behavioural proteins can be used in different ways as outputs of the process [26].

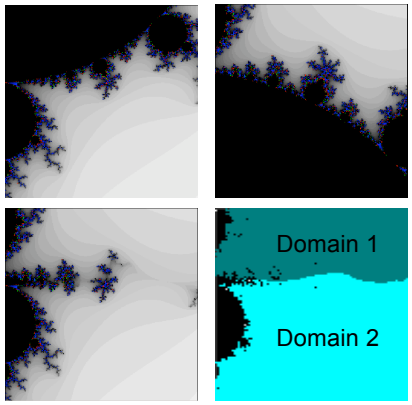


Figure 6.4: Top: Two fractal protein shapes. Bottom left: The merged protein. Bottom right: Protein domains in the merged protein. (From [26])

In the merged protein shape, each sampled pixel value is the maximum corresponding pixel value

from all proteins with non-zero concentrations. This makes the merged protein shape a patchwork of complex regions each belonging to one of the proteins present in the cytoplasm. We term the set of pixels in the merged protein originating from one protein as the domain of that protein. Figure 6.4 shows an example of two protein domains in the merged protein. If concentration of a protein drops to zero during development, that protein does not exist and so cannot have a domain in the merged result; instead other proteins may fill the region with their domains. This results in changes in the shape of those protein domains. This is analogous to the protein-protein interactions in biology resulting in proteins shifting their shapes.

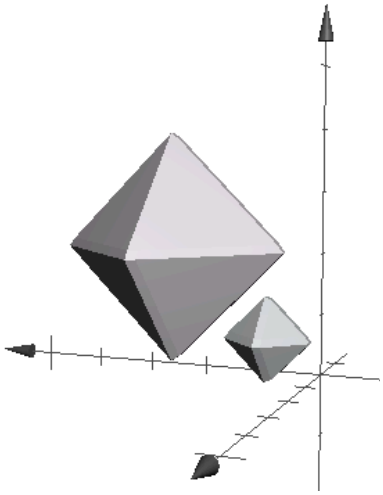


Figure 6.5: Two sample subspaces of MPS space (Merged Protein State space) define by two promoters in a 3D (3 pixels) space.

The value of different pixels in the merged protein at each development time step can together represent a single point coordinate in a multidimensional state space, each dimension being the value of one pixel. We shall refer to this state space as the MPS space (Merged Protein State space). The pixel values of the promoter and the absolute value of the Affinity threshold collectively describe a convex subspace in the MPS space (gene expression subspace), specifying when this gene can be expressed. Figure 6.5 shows an example of two such subspaces defined by two promoters in a 3D (3-pixel) MPS space.

This creates a different GRN for each combination of proteins present. Every time a protein concentration drops to zero or rises from zero, it can affect the shape of the other protein domains and change the shape of the merged protein. Each merged protein shape correspond with a point in MPS space and therefore each new shape switches some genes on and some others off, depending on their promoter subspaces. The expression of each gene is affected only by the pixel values of those protein domains that lie under the domain of the gene promoter. This allows promoters to ignore some pixel values in the merged protein shape, effectively stretching their expression subspaces infinitely in the corresponding directions in the MPS space. The sign of the affinity threshold determines if this gene is expressed or repressed when the current MPS dwells inside this subspace. The absolute value of the affinity threshold specifies

the size of this subspace. The hyperbolic tangent function (in equation 6.11) creates a smooth transition for probability of gene expression at the surface of this subspace. This can improve the evolvability of the GRNs by randomisation of some parts of the fitness landscape, which smoothes out the effect of some mutations. Using this mechanism, FGRN allows many different GRNs to be embedded in it with genes that can be switched on or off by existence of one or a combination of proteins.

The concentrations of individual proteins at each developmental time step can together represent a single point coordinate in a multidimensional state space, each dimension being the non-zero concentration value of one protein. We shall refer to this state space as the PCS space (Protein Concentration State space). When a gene is expressed, the concentration of the protein encoded in the gene is increased (or decreased) by a multiplicative sigmoid function (equation 6.12) of a linear combination of the concentration of those proteins with their domains covered by the gene promoter domain (equation 6.13). This linear combination is determined by the proportion of the areas of protein domains that lie under a gene promoter shape.

The Fractal Proteins algorithm can also be viewed from the perspective of pattern recognition, where the cell receptor gene performs input feature selection and scaling by masking some parts of the environment proteins. The rest of the GRN can be seen as a reservoir or a Liquid State Machine (see section 2.4.4). From this viewpoint, genes work as leaky integrator nodes with a multiplicative sigmoid transfer function, interacting through protein concentrations in a recurrent network. The areas of those protein domains that lie under a gene promoter domain define the input weights for that node (gene), and the concentration threshold (T_c) works as a bias. The behavioural genes work as the readout map (see section 2.4.4) translating the current multidimensional PCS into outputs. Even randomly generated reservoirs can be effectively used for pattern recognition and chaotic time-series prediction [176]. However, recent research [332] shows that bio-plausible features such as hierarchy and modularity in the reservoir network architecture can increase the performance and robustness of the reservoirs. Statistical studies also reveal such properties in biological GRNs [39]. Therefore, it is quite likely that, using fractal protein domains, this system is able to evolve the suitable network structures for a given problem. Existence of inactive genes and complete (or partial) dominance of one protein domain on other protein domains result in neutral networks in the fitness landscape - another of the reasons for the evolvability of this system observed in [24]. Neutral mutations can make the expression subspace of inactive genes drift. The randomness at the edge of these gene expression subspaces can give evolution some clues about the promising inactive genes that should be turned on to smoothly evolve a GRN into a fitter GRN. Also fault-tolerance and robustness, and reliability of the FGRN has been demonstrated in [24].

6.2.3 Evolutionary Algorithm

Two major functions of the evolutionary algorithms used in evo-devo models are selection and genetic operations. These processes are discussed here only briefly since, as it is realised later in this chapter, the evolutionary algorithm is not the focus of this work.

Selection

Selection is the mechanism that both maintains a diverse and relatively fit population of potential parents and selects parents from that population to be used in reproduction of new individuals. A few different methods for maintaining the diversity of the population are available. Speciation and explicit fitness sharing is already implemented in NEAT and CPPN-NEAT. NEAT uses the similarity measures of the genomes for classifying the population into different species that do not cross-breed. The original evolutionary algorithm used with FGRN does not include a speciation method. However, similar and other possible methods for maintaining and improving the diversity of the population such as deterministic crowding [243] can be easily added to this evolutionary algorithm.

For environmental selection, different types of fitness evaluation that can improve the efficiency of the evaluation and encourage complexification, such as using tournament selection, progressive fitness functions, multi-stage evolution, are possible and have been applied to the above evolutionary algorithms. Generally, methods that increase the number of evaluations without any benefit must be avoided when neural simulation is computationally expensive. For example, methods such as tournament selection, that require two neural microcircuits to compete, may appear to be biologically more plausible but they may also prove to be computationally more expensive than methods that have a specific fitness measure such as an error rate or score. The original evolutionary algorithms used with both NEAT and FGRN use a score as the fitness function.

A progressive fitness function that allows partial evaluation of a neural microcircuit in the beginning of the evolutionary run may prove to be helpful in reducing the computational cost of fitness evaluation. Such methods can use only part of the training or testing or both datasets to evaluate the individuals. As the average or the maximum fitness of the population increases the fitness can get more challenging by using the rest of the datasets. This requires a definition of the fitness function that is not dependent on the size of the datasets. In such methods, an inaccurate fitness evaluation can lead to stagnation of an elitist algorithm. To avoid that, the evolutionary algorithm used for FGRN removes the old individuals from the population, despite their high fitness, when they had a chance to pass on their inheritable genetic material.

The computational costs of the algorithms used for speciation, diversity maintenance and improving the performance of selection and fitness evaluation are usually negligible compared to the amount of computation savings that they are expected to offer. The complexity of the design and testing and hardware cost (if implemented in hardware) is a more important factor to consider for these methods than performance.

Genetic Operators

Genetic operators recombine selected parental genomes and mutate the result to produce offspring genomes. These operators must be compatible with the genetic representation used. The original CGP does not use recombination operators as it is generally found to be destructive on graph-based representations. NEAT and CPPN-NEAT use the historical markings to match related gene to allow constructive crossovers between two chromosomes. FGRN also uses similarity of the genes (using a sum of differ-

ence function and common bits in the type field) to find related genes in two chromosomes and then uses one of them in the offspring. Between these three methods, FGRN approach appear to do what NEAT is doing with a computationally more expensive algorithm but it is also biologically more plausible than NEAT method of historical markings. An even more bio-plausible method would be to do uniform or single point crossovers inside two matched genes.

Different bio-plausible mutation options based on the selected representation are available. A common mutation method is simple single-point mutations that change the value of the smallest modifiable genetic unit such as a single integer in CGP, a connection weight in NEAT and CPPN-NEAT, or any single real value in FGRN. Drift or creep mutations can slightly change the real values in the genes. Other more sophisticated mutations such as duplication, and deletion of genes, or adding connections between genes are both possible and already available in NEAT, CPPN-NEAT and FGRN. Duplicate mutation can add an extra copy of a gene to the chromosome that adds to the length of the chromosome. Delete removes a gene by random, decreasing the length of the chromosome. Adding a connection between two nodes in the directed graph can be quickly realised by a single mutation in NEAT, CPPN-NEAT, and FGRN. In FGRN a coding region of a gene must be copied to the promoter region of another gene or vice versa. As mutation probabilities are usually low the complexity of these mutation methods usually have an insignificant impact on the computational performance and computational complexity of the evolutionary model. The bio-plausibility of the mutation operators are usually limited by the bio-plausibility of the genetic representation and when representation allows, very bio-plausible and complex mutations can be designed and implemented without impacting the performance of the system. However, if complex algorithms are needed for mutations it may add to the hardware cost (if implemented in hardware), and significantly increase the complexity of the design and testing.

6.2.4 Implementation Options

Apart from the type of dynamical system used in the developmental model, the genetic representation and operators, and the selection methods used in the evolutionary model, there are other factors in the implementation of the above functions that impact both bio-plausibility and feasibility of the evo-devo model. Decisions such as which function to be implemented in hardware and which one in software, distribution of the processes over different processing elements, using deterministic or stochastic computing, and the choice of arithmetic methods in the hardware can all affect the final system. Here we focus on every one of these factors separately.

Hardware versus Software Implementation

Direct implementation of different functions of the evo-devo model in the hardware instead of implementing them in a piece of software running on one or more processors, on the FPGA (such as MicroBlaze) or connected to the FPGA such as a host PC, can both affect the feasibility of the system and change the scope of this study. Here implementation of the above functions of the evo-devo model in software and hardware are compared in terms of different feasibility measures of performance, hardware cost, scalability, reliability, and complexity.

Evaluation of the dynamical system needs to be repeated for each cell and for every iteration of the

development, if an iterative approach is used. Moreover, if a cell-chemistry model with local interactions is implemented, calculations for the protein diffusions need to be also repeated for each cell in each iteration. Although at different levels of abstraction these iterations (over time and space) can be reduced, generally, the computational complexity of the dynamical system and diffusion calculation in a bio-plausible developmental model are of order $\mathcal{O}(N_u N_c N_p)$ and $\mathcal{O}(C N_u N_c N_p)$. C, N_u, N_c, and N_p are representing the number of neighbours for each cell, number of development iterations, number of cells, and number of proteins respectively. Such a highly homogeneous and massive calculation can surely benefit from a parallel implementation on FPGA. A parallel implementation increases both the performance and the hardware cost but it also improves the scalability, fault-tolerance and reliability of the system. In a sequential implementation, the development time grows linearly with number of cells and proteins, which impact the scalability of the system. A parallel implementation of the dynamical system is also more bio-plausible as it is structurally more similar to the parallel process of cellular development.

Mapping of the genome to the dynamical system needs to be carried out once for each individual. Since the design of the Cortex model does not allow evaluation of more than one individual on the FPGA at a time, fitness evaluation of different individuals need to be performed sequentially. Therefore, it may make sense to map the genome to the dynamical system on software, particularly if it is a complex and heterogeneous process with many exceptions. However, in case of FGRN, calculation of Mandelbrot set with many samples from many proteins may gain some speedup from a parallel hardware implementation. Nevertheless, it is not a bottleneck compared to the computational cost of the dynamical system. Similar to the dynamical system, a parallel implementation of the mapping is more scalable and bio-plausible than a sequential software implementation. The sequential computation time of mapping grows linearly with the number of proteins. A parallel implementation may also improve the performance of the system depending on other factors.

With fitness evaluation of individuals being carried out sequentially, results of all the other functions in the evolutionary model (genetic operators and selection) will be used sequentially and given the bottleneck of fitness evaluation, there is no point in parallel implementations for such heterogeneous processes. Therefore, all those functions are better to be implemented in software running on the embedded processor or the host PC connected to the FPGA. Since a host PC is needed for the initial configuration of the FPGA, using that PC for some light-weight sequential computations does not impact the performance and hardware cost of the whole system. A software implementation also provides a more flexible environment for design and testing different evolutionary algorithms and their parameter values in an experimental setting. This also reduces the design, implementation and testing complexity of the whole system and adds to the observability of the evolutionary processes during experiments.

When a function is implemented in hardware it will be subjected to the limitations and trade-offs of the hardware implementation on an FPGA and it lies inside the scope of this study. With the above analysis, all the evolutionary (not developmental) processes are better to be implemented in software and thus are out of the scope of this work. Moreover, a plethora of studies on bio-plausible evolutionary

processes without respect to the limitations or benefits of a parallel implementation in an FPGA exist in the literature that makes pursuing that tread of investigation redundant here. Therefore, in the following sections we focus on the investigation of the challenges in the design and implementation of the developmental process.

Distributed Models

In both abstract and multicellular models, one or more functions need to be evaluated for each cell (or for different positions in the substrate). These evaluations can be performed in parallel in all cells by distributing the computation power over the development substrate. Since all the input data for evaluation of the functions is locally available (either cell state vector, neighbouring cells states or local feedback from the Cortex), it requires the minimum communications between local processing elements (PEs). The same operations are performed on different data. This is known as SIMD (Single Instruction Multiple Data) in the field of computer architecture. Architectures such as GPUs (Graphics Processing Units) are very efficient at such computations that involves minimum communication between processing elements, local data and identical operations. Similar but custom architectures can be also designed on FPGAs to carry out those computations very efficiently.

The hardware cost of each PE depends on the complexity of the functions and the amount of memory needed for storing the state vector of each cell. Also the time that each PE needs to update the state vector of a cell depends on the complexity of the functions. However, the total hardware cost and performance of the whole developmental model depends on the number of PEs. It is possible to allocate one PE to each cell, or allow cells to time-share a PE. For example, for a cortex of size 120x12 cells it is possible to have only 12 PEs each processing the state vectors of 120 cells, or to have 60 PEs each responsible for 24 cells, and so forth. There is a well known trade-off between performance and hardware cost in time-sharing of PEs. In cases, such as this, where all the data is locally available, it is theoretically possible to achieve a linear speed-up by increasing the number of PEs. The final decision about the suitable number of PEs depends on the design constraints and criticality of the performance and hardware costs in each design. At one extreme, the number of PEs is equal to the number of cells. This is the fastest but hardware intensive design option. It is also the most scalable option, as the development time will not depend on the size of the Cortex. Such a distributed design will be also very reliable and fault-tolerant as a faulty PE can not affect more than one cell. As the number of PEs decreases, performance, scalability, and reliability of the design are reduced. At the other extreme, only one PE can be responsible for processing of all the cells. This can be considered a centralised model discussed in the following section.

Centralised Models

A centralised model with one or very few number of PEs for processing all cells, generally has the lowest performance, scalability, and reliability, but also the minimum hardware cost. However, some techniques exist, which can be used to improve the performance of the centralised models that are not possible or efficient in distributed models. For example, caching data that is repeatedly computed or accessed in different cells allow a centralised model to avoid repeating computations or memory accesses, which increases its efficiency.

A centralised model can also store the instructions or data that define the dynamical system functions locally and access them efficiently. While a distributed model needs to either initially send the data to all the PEs and store it locally for each PE, which increases the local memory hardware cost significantly, or stream the instructions to all the PEs, which increases the global communication hardware cost and requires the synchronisation of the PEs.

With relatively low clock frequency of most of the FPGAs compared to high-end PC processors, a centralised model implemented in FPGA (in hardware or software) is not justified. Moreover, availability of GPUs on most of the PCs these days, allow much more efficient parallel implementations on a centralised host PC than a lightly-parallelised implementation on an FPGA.

Stochastic and Deterministic Implementations

Similar to the neuron model investigated in chapter 4, the developmental model can be also implemented using deterministic or stochastic arithmetic. As it was examined in section 4.3 and 4.4, it is possible to use both stochastic and deterministic computing for designing a dynamical system. Generally, stochastic computing is more bio-plausible as it better mimics the detail of the chemistry between single molecules and their collective effects on the concentrations and interactions of different proteins. Stochastic computing is also more robust to noise. However, it has an intrinsic performance-accuracy trade-off that limited its use in the design of the neuron model. Nevertheless, performance of the evo-devo model is not as critical as in the neuron model. This is mainly because the developmental processes are much slower than neural processes in biology. Moreover, the activity dependent developmental processes require mean activity feedback data over many update cycles of the neural system, as explained in section 5.2.4.

Unlike a neuron with an estimated signal to noise level, the accuracy and noise of different pathways in biological development is different. Estimations and measurements of the steady-state noise levels in concentrations of a few proteins show SNR values ranging from 8dB to 76dB [299]. As a matter of fact, evolutionary processes appear to be able to tune the accuracy and reliability of different pathways according to the needs and circumstances [404]. Therefore, the developmental model is required to be able to support a high level of accuracy in case it is needed in a critical pathway. In chapter 5 a trade-off was also observed between the compactness and accuracy of the stochastic neuron models. This was mainly because in the stochastic neuron model it was required to convert the stochastic variable into a binary representation for function generation and detection of the action potential. It must be re-examined here if a stochastic developmental model still needs such conversions. Stochastic simulation of the chemical processes and particularly biological gene-regulatory networks using FPGAs has been already proposed and studied (see for example [323] and [267]). Synchronous and asynchronous implementations of stochastic models are suggested in the literature. Figure 6.6 shows an example of a biological GRN that is translated into an asynchronous stochastic logic circuit on FPGA and simulated with 60,000 times speed up over software simulations [267]. Deterministic simulations of GRNs based on binary arithmetic and differential equations are also implemented on FPGAs with success (see [323] and [267] for references).

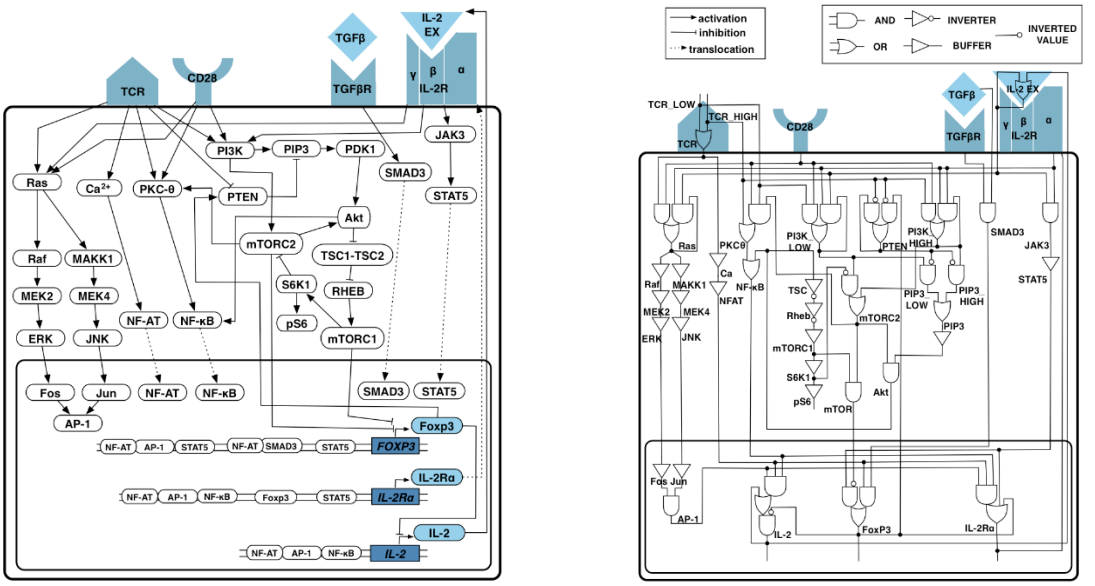


Figure 6.6: Example of translating a biological GRN into a stochastic asynchronous logic circuit (T-cell differentiation network from [267]). a) Molecular interaction map of T-cell differentiation GRN. b) Gate-based logic implementation of the same GRN.

Further investigation of suitability of these computation methods for a developmental model depends also on the type of functions used for modelling the dynamical system (GRN) and the genetic representation. Most of the bio-plausible evo-devo models are represented as differential equations and translating them into stochastic processes adds to the complexity of the design process.

Bit-serial and Bit-parallel Binary Arithmetic Implementations

Again similar to the neuron model, serial and parallel processing of bit values in a binary arithmetic is possible. Bit-parallel arithmetic has higher hardware cost and provides higher performance but with no advantage in scalability, or reliability of the system. The hardware cost of a parallel implementation grows with the number of bits used for the value representation, while the hardware cost of a bit-serial arithmetic implementation is fixed. The performance of a bit-serial implementation degrades with the number of bits. A parallel arithmetic design may be slightly simpler to design and test due to availability of all the bits at the same time.

6.3 Summary and Comparison of Design Options

Different approaches to the design and implementation of the developmental model, their challenges, important factors, constraints, and trade-offs are compared and summarised in this section. As explained in the previous section, the evolutionary model is perfectly justified to be implemented in software and therefore the focus of this work is on the bio-plausible developmental models. The design options, factors, and their trends and trade-offs are outlined in table 6.2. Different design options are grouped in three comparison sections of: dynamical system abstraction, genetic representation, and implementation methods and then sorted based on their bio-plausibility to reveal the related trends. CPPN appear be

be completely out of place (with all the measures at their lowest) as it is more suitable for an abstract generative model, while the genetic representations are evaluated here in the context of a multicellular developmental system.

Bio-plausibility

For the dynamical system of the developmental model, multicellular (cell chemistry) models have higher levels of bio-plausibility since they take into account the local interactions between cells, cell signalling, and time dependence of the different developmental processes. However, there is a range of different possibilities between these two extremes of abstraction and bio-accuracy. Starting from an abstract model, depending on inclusion of time, feedback, intercellular signalling, diffusion, and physical cell interactions in the space, increasingly more bio-plausible and complicated models can be achieved. However, as we move from abstract models to more bio-plausible multicellular models with local cell interactions, the performance, compactness and simplicity of the model decreases, but its scalability and reliability improves.

Looking at the genetic representations, to examine the structural accuracy and bio-plausibility of each one of the examples investigated in the previous section they can be compared with biological models of gene-regulatory networks . As explained in [46], the whole process from gene expression to protein synthesis is regulated by a number of different factors. One is the interaction between proteins in the cell (transcription factors) and the cis-regulatory region of a gene. When a gene is expressed the coding region of the gene is transcribed into a strand of nRNA. Strands of nRNA can degrade before they go through the next step of splicing, which produces mRNA strands. Strands of mRNA also degrade before they finally produce proteins. These proteins can also group together to create protein complexes that may behave differently than single protein molecules. In [46], the transcription rate of the nRNA is a non-linear function (similar to sigmoid function) of the concentration of the transcription factors (other proteins) that need to bound to the cis-regulatory region for expression of the gene. Splicing and protein synthesis and degradation processes are modelled as linear processes with fixed rates. As a result the rate of protein synthesis is a non-linear function of the concentration of the transcription factors. Depending on the number of bounding sites this non-linearity can be steeper to smoother. In the most abstract form the concentration of the synthesised protein can be modelled as Boolean AND and OR functions of the transcription factor concentrations.

With this knowledge from biological models, CPPN with its course-grain functions such as Gaussian and Sine appear quite implausible and more suitable for abstract models based on spatial coding of the morphogen patterns than GRNs in multicellular models. CGP, in its simplest form with Boolean primitives, can capture the Boolean nature of the GRN nodes but not real-values of the concentrations. Also it does not directly provide any adjustment for the steepness of the non-linearities unless used in a stochastic system. Neural network-based models such as NEAT and ESN can capture the real-valued nature of the protein concentrations, and by using weight adjustment may also support changes in the non-linearities. However, the original NEAT does not model the recurrent structure of a GRN, concentration integration, and protein degradation processes while ESN with leaky integrator neurons and a

recurrent network architecture can also capture the protein concentration integration and decay. FGRN appears as a more bio-plausible model in respect to the way it models the concentrations, nonlinearities, protein degradation and adjustments for different aspects of the whole process. Moreover, FGRN models the one-to-many and many-to-one mapping in the biological processes of transcription, and protein folding and assembly using a fractal mapping and protein merging. However, other investigated models do not account for those processes.

Regarding implementation options, a hardware-based distributed and stochastic model is more bio-plausible than other options. Bio-plausibility of a distributed software-based implementation (stochastic or deterministic), although possible, depends on many other factors and is out of the scope of this investigation. A centralised deterministic implementation is the least bio-plausible option, as distribution of the processes over different PEs, and distribution of the protein concentrations over small chunks (stochastic bits) adds to the structural accuracy of the model.

Performance

Abstract models can have better performances (in terms of development time) depending on what they abstract out. However, if they need to include time-dependence and local interactions of the cells they will be very similar to the multicellular models, which have lower performances than abstract models.

Regarding genetic representations, Boolean networks encoded by CGP are computationally cheapest to decode and run. NEAT and ESN are computationally more expensive compared to Boolean CGP. CPPN needs complex computation of course-grain functions such as Gaussian and trigonometric functions that are much slower to compute. FGRN also uses simple integration and sigmoid functions similar to NEAT and ESN with more linear computations for merging proteins, promoter bounding and concentration integrations that makes it one of the computationally expensive representations for the dynamical system.

A software implementation of the developmental model on a PC can be in fact faster than a centralised implementation on FPGA due to low clock frequencies of FPGAs compared to high-end PC processors. However, a distributed (deterministic or stochastic) implementation on FPGA with enough number of PEs can run faster than a similar (deterministic or stochastic) implementation in software. Bit-parallel implementations have higher performances than similar bit-serial implementations. Stochastic implementations are always the slowest.

Compactness

Similar to performance, abstract models can be implemented in hardware quite compactly compared to multicellular models that need extra hardware for local interactions between cells.

The compactness of the hardware needed for mapping these genetic representations is proportional to the computational cost of the unique operations in the mapping. The compactness of the hardware needed for the dynamical system also depends on both the homogeneity and complexity of the primitive functions used in the dynamical system. Therefore, Boolean networks encoded in CGP is the most compact form while CPPN and then FGRN are the most hardware intensive representations due to course-grain functions of CPPN and complexity and dynamism of FGRN.

A software implementation is the most compact option as it does not add to the FPGA hardware cost if running on the host PC. Even using an on-chip processor (such as MicroBlaze) needs less hardware resources than a custom hardware for complex developmental algorithms such as FGRN. Distributed deterministic bit-parallel implementation has the highest hardware cost (depending on the number of PEs). Stochastic and bit-serial implementations have the next places. Centralised implementations have the same order but they are all much more compacter than distributed implementations.

Scalability

Multicellular models with their local cell signalling can better scale to the size of the substrate while some studies suggest that abstract models may not scale well to larger cortex areas, more number of inputs and outputs, and more complex problems as multicellular models do.

One of the major factors that affects the scalability of different genetic representations is the intrinsic capacity of the genetic representation for evolving modules and reusability of modules. The original forms of CGP, NEAT, and CPPN are not designed to reuse modules and did not show any sign of that. All these methods have more advanced versions (such as ECGP, MCGP, HyperNEAT-LEO) that allow and promote modularity to achieve better scalability. Here pure versions that are simpler to implement and analysis are used as representatives of different approaches. ESN uses a regulated random network generation that depending on the bio-plausibility of the network generation method (As in Liquid State Machines) may promote modularity. FGRN, on the other hand, uses a fractal mapping for specifying connections in GRNs. Both these methods are shown to promote modularity to some extent. Moreover, FGRN is able to switch genes on and off in different contexts, which allows it to organise and reuse modules particularly in different types of cells. Although an accurate and conclusive comparison of scalability of different representations needs further investigation based on fair benchmarks, it is possible to conclude, based on the available literature, that FGRN and then ESN are more scalable than the others due to their intrinsic mechanisms that promote modularity in the GRN.

Regarding scalability of different implementation methods, distributed implementations are generally more scalable while a centralised and a software implementation on a hardware that is not scaled with the cortex size is much less scalable.

Reliability

Fault-tolerance, regeneration, self-repair, and robustness, are main features and factors of the evo-devo model affecting the reliability of the system. Fault-tolerance, regeneration capacity and robustness of the multicellular models compared to abstract models has been shown in a few studies.

Among different genetic representations ESN and FGRNs are shown to produce fault-tolerant, robust, and reliable dynamical systems. CGP has also shown robustness and fault-tolerance when used in developmental systems. Overall, if all these methods are used in a multicellular developmental system that allows regeneration, there is no evidence of significant difference in the reliability of the whole system between different genetic representation. It can be conjectured that CPPN is slightly less fault-tolerant as it uses courser-grain functions compared to other methods. Also more evidence is available in the literature for fault-tolerance, robustness and reliability of the developmental systems based on CGP,

ESN and FGRN compared to NEAT.

The implementation method has a significant impact on the reliability of the whole system. Generally, distributed models have better fault-tolerance and robustness compared to centralised and software-based implementations. Stochastic implementations are also more robust compared to deterministic ones as been already analysed in chapter 4.

Simplicity

Abstracted models are usually simpler to design, implement and test than multicellular cell chemistry models. Comparing the complexity of the genetic representations, CGP is the simplest and the most straightforward method for hardware implementation. NEAT and ESN have their own complexities but are still simpler than FGRN to design, implement and test in hardware. CPPN with its complicated primitive functions is the most complicated method to design, implement and test in hardware.

Software implementation of the developmental system is the simplest method for design, implementation and testing, although a stochastic software implementation would be slightly more complex to design and test. Then a centralised deterministic method (bit-serial or parallel) is next complex option. A centralised stochastic and a distributed deterministic (serial or parallel) are equally more complex than previous options. The most complex method for design, implementation, and testing is a distributed stochastic implementation in hardware.

Bio-plausibility-related Trends

By sorting different design approaches from low to high bio-plausibility in table 6.2 a few general trends related to bio-plausibility of the evo-devo models are revealed. The usual bio-plausibility-feasibility trade-off is evident in the form of the impact that abstracting the local intercellular interactions has on the bio-plausibility, performance, and compactness of the system. The same trade-off exists, to some extent, in the genetic representation method and implementation. This is less emphasised in the implementation methods, as performance and compactness cancel each others out. However, considering performance and compactness as two multiplicative factors of efficiency, reveals the same trend. This can be presented as a bio-plausibility-efficiency trade-off in the evo-devo model design.

Moreover, a similar trade-off is present, across the table, between bio-plausibility and simplicity of the design and testing. More-bio-plausible approaches and methods are more complex in design, implementation, and testing.

Scalability and Reliability Benefits

Unlike the general trend in the trade-offs between bio-plausibility and feasibility measures (suggested in section 1.3), the scalability and reliability measures of the evo-devo model generally tend to increase with bio-plausibility. This trend is observable in the all three comparison groups, although less pronounced in the reliability of genetic representations. This trend can be viewed as an emergent property of the developmental and evolutionary processes and can be presented as the main advantage of a bio-plausible approach in the evo-devo model.

Table 6.2: Summary and comparison of different approaches and methods in the design of the evo-devo model and their trade-offs. Different approaches and methods in each section of the table are sorted according to their bio-plausibility revealing its impact on the other factors. The $\sim$ symbol shows that a design or implementation approach can both increase and decrease a measure depending on other factors.

Competing design approaches and general options		Bio-plausibility	Performance	Compactness	Scalability	Reliability	Simplicity
Dynamical system	Abstract Models	−	+	+	−	−	+
	Multicellular (cell chemistry) models	+	−	−	+	+	−
Genetic Representation	CPPN	○ ○ ○	● ○ ○	○ ○ ○	○○	○○	○ ○ ○
	CGP (Boolean)	● ○ ○	● ● ●	● ● ●	○○	●●	● ● ●
	NEAT	● ○ ○	● ● ○	● ● ○	○○	●○	● ● ○
	ESN	● ● ○	● ● ○	● ● ○	●○	●●	● ● ○
	FGRN	● ● ●	● ○ ○	● ○ ○	●●	●●	● ○ ○
Software Implementation		$\sim$	● ● ○○	● ● ●●	○○	● ○ ○	● ● ●
Hardware Implementation	Centralised deterministic bit-serial	● ○ ○	● ○ ○	● ● ●○	○○	○ ○ ○	● ● ○
	Centralised deterministic bit-parallel	● ○ ○	● ● ○	● ● ○○	○○	○ ○ ○	● ● ○
	Centralised stochastic	● ● ○	○ ○ ○	● ● ●○	○○	● ○ ○	● ○ ○
	Distributed deterministic bit-serial	● ● ○	● ● ●○	● ○ ○○	●●	● ○ ○	● ○ ○
	Distributed deterministic bit-parallel	● ● ○	● ● ●●	○ ○ ○○	●●	● ● ○	● ○ ○
	Distributed stochastic	● ● ●	● ● ○○	● ○ ○○	●●	● ● ●	○ ○ ○

6.4 Case Study: Neural Evo-Devo Model

Investigation of different approaches in the previous sections showed that bio-inspired systems with bio-plausible modelling of biological neurodevelopment are more promising in terms of emergent properties such as fault-tolerance, robustness and scalability. In this section the design process of a new multi-cellular neurodevelopmental model, and some experiments and results are reported as a case study.

Among different approaches, Fractal Gene-Regulatory Network was selected for a closer practical investigation as it showed a good level of evolvability in different applications [25, 24, 27, 26] and as it seems more bio-plausible than other genetic representations. Therefore, here a multicellular version of a similar evo-devo model is designed. From the implementation point of view, a distributed stochastic implementation in hardware provides highest bio-plausibility for the price of complexity and lower performance and compactness. Performance is not a critical factor here as developmental processes are computationally less intensive than neural processes in the system. However, compactness, and complexity of the design and testing are both very critical factors in this case study as the hardware resources and project time are both very restricted. Since most of the hardware resources in the FPGA

have been allocated to the computationally more intensive processes of neural simulation and communication, there are not many more resources available for implementing a useful example of a bio-plausible distributed evo-devo model on the chip. A centralised model in hardware, while still too complex for the time-constraints of this project, does not offer any advantages in terms of bio-plausibility, reliability, or scalability. A software implementation, on the other hand, can offer simplicity of the implementation, allowing to design, implement, and test a bio-plausible evo-devo model that meets both the time and performance constraints of this project, while allowing us to explore slightly different designs. However, as this study is focused on the hardware design and implementation and its challenges and trade-offs, in this case study, an evo-devo model that is also suitable for distributed hardware implementation is preferred.

As discussed in section 6.2.4 the sequential fitness evaluation of individual neural microcircuits on the FPGA platform, makes a parallel implementation of the evolutionary processes redundant. A serial implementation of the evolutionary processes in FPGA, apart from being slower than software implementation on a PC (due to lower clock frequencies of the FPGAs compared to PC processors), is very time-consuming to design, implement and, test, and it significantly reduces the flexibility of the model for exploring different settings and approaches. With this analysis, it is decided to implement the evolutionary processes in software running on the host PC that is connected to the FPGA. As this study is focused on the hardware-based models investigation of the challenges and trade-offs of a software implementation is out of the scope of this study. Moreover, a plethora of different studies is already published on this subject. Therefore, this case study focuses on the neurodevelopmental model and simply adopts an evolutionary algorithm from [24, 25].

6.4.1 Neurodevelopmental Model

The neural evo-devo model proposed in this work is similar to the FGRN [26] in terms of using the same bio-plausible general genome structure and basic protein-protein and gene-protein interactions. This is mainly motivated by the evolvability of the fractal proteins in different successful applications [25, 24, 26, 207, 208, 209]. Adoption of these features is based on the analytical study in section 6.2.2.

The FGRN system uses a fractal protein translation and folding mapping into 2D shapes using Mandelbrot set. Implementing the Mandelbrot set in hardware is not straightforward and requires complex numbers and operations. Since the fractal mapping needs only to be executed once for each individual this can be implemented in software. Nevertheless, a faster and simpler protein-folding mapping into 1D protein shapes (Called LGRN - Logistic GRN) is introduced in this study to demonstrate how other protein folding mappings techniques are also possible to be plugged into this system. The protein folding process is a separate module and can be replaced with any other mapping (or removed altogether as shown after this study in [209]) in order to compare the evolvability and performance of different similar methods.

The neurodevelopmental model proposed here, fully exploits the inter-cellular signal proteins using novel behavioural protein-protein interactions. However, these protein interactions are also examples of different possible bio-plausible methods that can be used in the design of a neurodevelopmental model

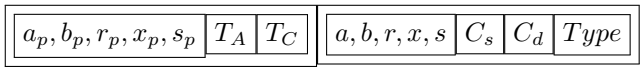
and can be replaced with other equally promising methods.

Definition of Proteins

Proteins are defined here as strings of real numbers in [0,1] of a certain length (L). In a hardware implementation these numbers can be mapped to a range of integers or stochastic representation according to the implementation method used. The values of the real numbers collectively define the shape of a protein. Figure 6.8 shows two samples of protein shapes of length $L = 10$. These values are calculated by the protein-folding mapping explained in this section.

Genome Structure

The genome consists of a single chromosome of a variable number of genes. The variable length chromosome allows complexification of the FGRN. The number of chromosomes can be increased if needed. Each gene consist of 15 fields:



The first five values (a_p, b_p, r_p, x_p, s_p) specify the shape of the promoter using the protein-folding mapping. These values along with T_A (affinity threshold) and T_C (concentration threshold) form the cis-regulatory or promoter region of a gene. The next five values (a, b, r, x, s) specify the shape of the protein synthesised by this gene (through protein folding again). These values along with C_s (stability coefficient - specifying the decay rate of the protein), C_d (diffusion coefficient of the protein) and $Type$ (protein type) form the coding region of the gene. All values are real numbers except for $Type$, which is a bit-string that can specify any combination of the protein types. In this system, proteins are of eight different types (written in *italics*). Figure 6.7 depicts the classification of the different protein types in this system. They can be classified into two major groups: transcription factors and structural proteins. Transcription factors, which regulate the expression of the genes, include maternal factors (*soma cell maternal protein, glial cell maternal protein, IO cell maternal protein*) and *regulatory proteins*. A group of regulatory proteins are known as intercellular signal proteins. Structural proteins, which are virtually part of the cell structure and influence the behaviour of the cell, include behavioural proteins (*axon growth protein, dendrite growth protein, synapse formation protein*) and *cell receptor proteins*. Maternal factors work as the inputs to the GRN dynamical system. Regulatory proteins are feedback signals in the recurrent part of the GRN. Intercellular signal proteins are signals between GRNs of different cells. Structural proteins are the outputs of the GRN. Among them, cell-receptor proteins in each cell can control the signalling between cells (connections between GRNs in different cells). The role of each protein type is explained further later in this section.

Logistic Protein Folding

Protein folding is the process that translates a set of a, b, r, x, s values (or a_p, b_p, r_p, x_p, s_p values in case of a gene promoter) into a protein (or promoter) shape which is a string of length L of real values. This is performed using the logistic map [251]. The logistic map is a very simple dynamical system of the form:

$$x_{k+1} = \mu x_k (1 - x_k) \tag{6.15}$$

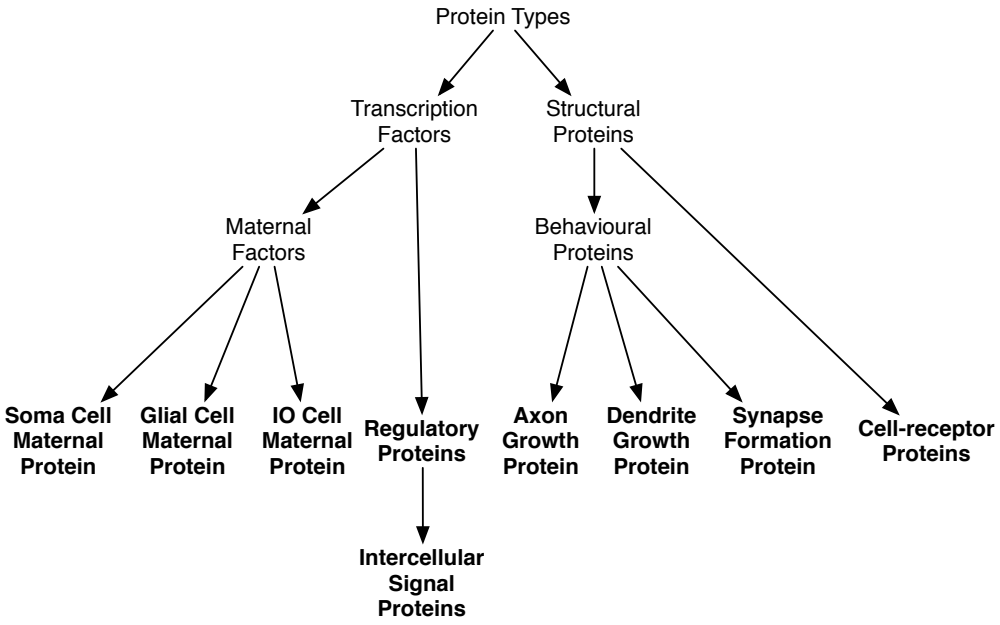


Figure 6.7: Classification of the different protein types used in the case study neural evo-devo model. The actual protein types are shown in bold.

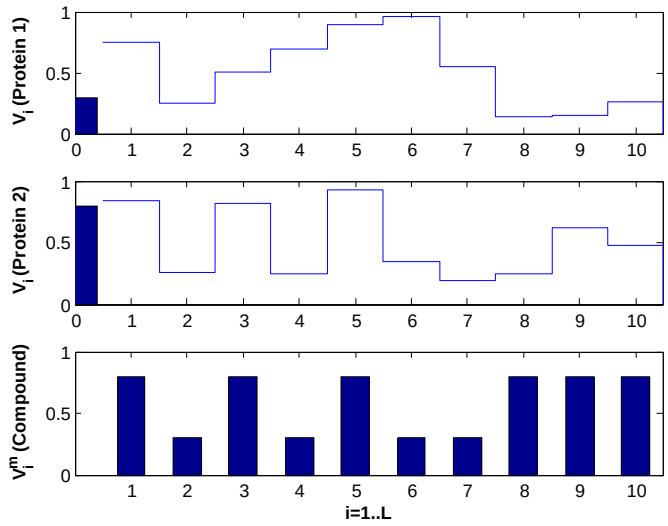


Figure 6.8: Example of two protein shapes ($V_i, i = 1..L$) of length $L = 10$. Their concentrations are shown as bars on the left. Merging these proteins, results in the protein compound shape (V_i^m) at the bottom.

that can create very complex time series with steady, transient, periodic, or chaotic behaviour. In this equation x_k is the value of the time series in step k, and μ is the logistic map parameter. In this system μ is calculated based on r (or r_p) from a gene using the following equation:

$$\mu = 3 + tanh(5|r|). \tag{6.16}$$

This equation allows generation of logistic map parameters in the range of $[3, 4)$. The $tanh$ function allows finer tuning of the logistic parameter in the chaotic region of the system.

The x field in the gene specifies the initial value of x_k in this equation. The logistic map equation (6.15) will be first iterated for $n = ||\lfloor L \cdot s \rfloor||$ times. Then the x_k values in subsequent iterations of the equation are scaled and offset by a and b using equation:

$$V_{k-n} = (2a - 1)x_k + 2b - 1 \tag{6.17}$$

to calculate all the protein shape values V_i for $i = 1..L$. This way, s controls the number of skipped iterations before using the x_k values. This allows the system to pass the transient part of the logistic map or use the transient part to generate the protein shape. The a_p, b_p, r_p, x_p, s_p fields are used instead in case of promoter translation. All the protein and promoter shapes are calculated using this mapping and stored before starting the iterative developmental processes. Figure 6.8 shows two sample protein shapes. These shapes can be shifted horizontally by changing the value of s. Protein shapes can be scaled and shifted vertically by changing a and b respectively. The r value in the gene specifies the behaviour of the dynamical system, thus shape of the protein. The x value in the gene can significantly affect the shape of the protein, particularly when the dynamical system has a chaotic behaviour and $|s| \gg 0$. This is because of the sensitivity of a chaotic system to initial conditions. However, this sensitivity can be smoothly controlled by evolution using both s and r values. This technique and its related empirical equations are results of a preliminary experiments using simulation aimed at evolving proteins and compound proteins of exact required shapes.

Protein Diffusion

For the diffusion of the proteins, a bio-plausible diffusion system that can be implemented as parallel processes and provides protein concentration gradient is needed. Stochastic and non-deterministic diffusion systems can be implemented using minimum hardware resources in parallel. For example, using Cellular Automata with Margolus neighbourhood [15] was explored through simulations. However, the trade-off between the noise level, speed, and hardware resources justifies the selection of a deterministic approach resulting a fast and parallel implementation that is scalable and reliable. The following diffusion model is the result of a preliminary study and related simulations to design a diffusion model that is straightforward and efficient to implement in FPGA and also in software or GPUs (Graphics Processing Units).

For each protein described in the genome, a real-valued concentration in the range $[0, 1]$ is stored for each cortex cell (thus two concentration values for two half cells of a soma cell). Similar to protein shape values, these concentration values can be mapped to a range of integer numbers suitable for a

compact hardware implementation. Before any protein-protein or gene-protein interactions take place, the amount of proteins diffused into neighbouring cells should be calculated. Here, a simple weighted average of concentration values of the cell and its neighbouring cells of form:

$$c_0^{t+1} = C_s\big((1 - C_d)c_0^t + \frac{1}{4}\sum_{i=1}^{4} C_d \cdot c_i^t\big) - 0.002 \tag{6.18}$$

is used where c_0^t is the concentration value in the centre cell at development step t, and $c_i^t, i = 1, 2, 3, 4$ are the concentration values in four neighbouring cells. The C_s is the stability coefficient of the protein, which is a real number in $[0, 1]$, with 1.0 meaning no decay. The C_d is the diffusion coefficient, again a real number in $[0, 1]$, with 0 meaning no diffusion. Both of these values come directly from the gene. Zero diffusion coefficients are useful for those proteins that cannot cross the cell membrane and diffuse in the Cortex. The -0.002 offset makes sure that concentration can actually drop to zero instead of converging to zero [26]. Concentrations outside of the $[0, 1]$ range are clipped back to $[0, 1]$ range.

Protein-protein Interactions

There are different types of protein-protein interactions depending on the protein types. Proteins can merge to create a protein compound. The protein compound is also a string of real values of length L. Each value in the protein compound string is equal to the concentration of the protein with the highest value in that position of the string:

$$V_i^m = C_j \text{ where } j = \underset{k, C_k \neq 0}{\operatorname{argmax}} V_i^k \quad \text{for } i = 1..L \tag{6.19}$$

Figure 6.8 shows how two different sample protein shapes of length $L = 10$ are merged to result in a protein compound of the same length. Note that protein compounds do not have a concentration of their own. Protein compound values are actually the concentrations of the proteins with maximum value over all merged proteins at each shape location. Only proteins with non-zero concentration (existing proteins in a cell) can contribute to the shape of the protein compound. This is to create a very diverse and dynamic set of protein compounds, shaped by the protein concentrations in different regions of the cortex. All the proteins in the genome, which are tagged as *cell receptor protein* in the $Type$ field of the genes, are merged in each cortex cell to create a compound cell receptor shape for that cell. This string of real values is then used as a mask to filter the shape of those proteins that are tagged as *intercellular signal protein*, meaning that only those shape values with a corresponding non-zero value in the mask are used [26]. The masked shapes of the intercellular signal proteins and all the proteins that are tagged as a transcription factors (*regulatory protein* or any type of maternal factors) are then merged together to create a protein compound in that cell. Figure 6.9 shows the above process that produces a protein compound inside each cell.

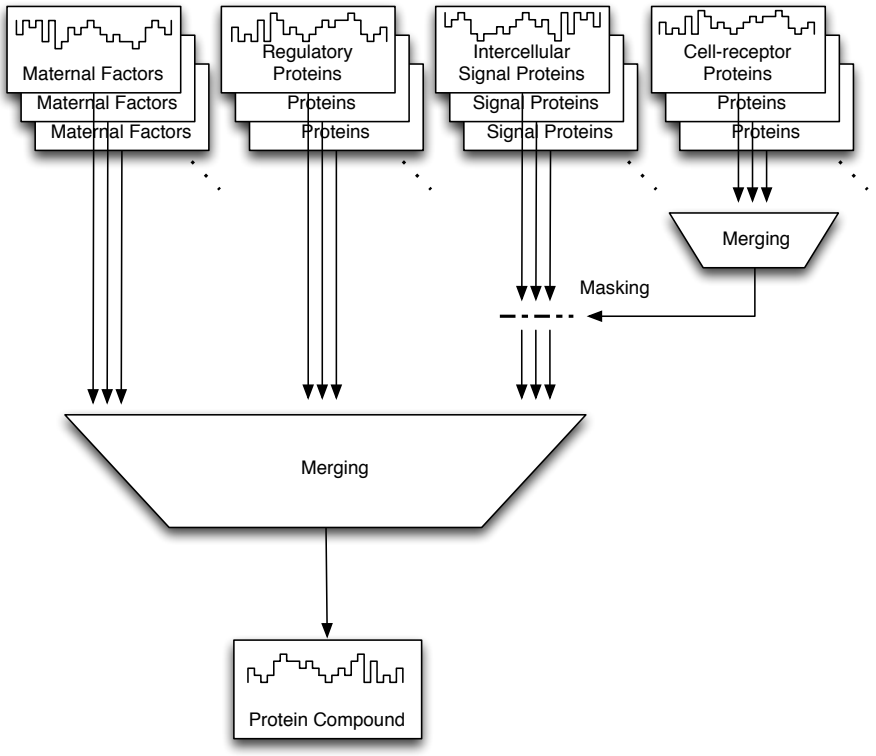


Figure 6.9: This diagram shows how different types of proteins interact inside a cell using two different operations of merging and masking to produce a protein compound inside a cell.

Gene Expression (Gene-protein Interactions)

In each cell, the protein compound interacts with the shape of the promoter in each gene [26], resulting a difference value δ, defined as:

$$\delta = \frac{\sum_{i=1, V_i^p \neq 0}^{L} |V_i^m - V_i^p|}{\sum_{i=1, V_i^p \neq 0}^{L} 1} \tag{6.20}$$

where V_i^m and V_i^p are the ith values in the protein compound string and in the promoter shape string of the gene. The probability of the gene expression is then defined as [26]:

$$P(E|\delta, T_A) = \begin{cases} \frac{1+tanh\left(30(2T_A-1+\delta)\right)}{2} & \text{if } T_A < \frac{1}{2} \\ \frac{1+tanh\left(30(2T_A-1-\delta)\right)}{2} & \text{if } T_A \geq \frac{1}{2} \end{cases} \tag{6.21}$$

where T_A is the affinity threshold of the gene promoter. In each development cycle each gene is randomly expressed with this probability. If a gene is expressed, the concentration of the protein coded in the gene will be increased (or decreased) by [26]:

$$\sigma = c_p \cdot tanh(c_p + T_C) \tag{6.22}$$

where T_C is the concentration threshold of the gene promoter, and c_p (total concentration seen by promoter of the gene) is calculated using [26]:

$$c_p = \frac{\sum_{i=1, V_i^p \neq 0}^{L} C_{(\underset{k, C_k \neq 0}{\operatorname{argmax}} V_i^k)}}{\sum_{i=1, V_i^p \neq 0}^{L}} \tag{6.23}$$

Most of the gene expression mechanism and equations come from the original FGRN literature [26] and is kept untouched as they are designed based on empirical results and bio-plausibility assumptions. One of the main differences is in the calculation of the protein compound (equation 6.19) that is slightly changed to make the GRN more dynamic and responsive to the protein concentrations. In FGRN, the compound protein shape (merged protein) is calculated using the maximum value of each pixel over all existing proteins (equation 6.9). Figure 6.10 demonstrates how the original FGRN method of calculating the shape of the protein compound works. However, in this model, the value of each location in the compound protein shape is the concentration of the protein with the highest value in that location over all existing proteins (equation 6.19, and figure 6.8). For example, if comparing the first shape value of all proteins, the third protein of the genome has the highest value, then the concentration level of the third protein will be used as the first shape value of the compound protein, and so on for the second, third and other shape values in the proteins. This modification was made since a slight change in a protein concentration that drops it to zero could suddenly turn genes on or off in a binary manner. With the concentrations involved in the shape of the protein compound, the evolutionary process is able to adjust exactly at which range of protein concentrations a gene can be turned on or off. This method results in a rather more dynamic but simpler protein compound shape compared to the original FGRN model. It is not clear, without further investigations, if this can have any benefit to the evolvability and other features of the developmental model. However, this dynamic protein compound shape is slightly more bio-plausible than FGRN without adding to the complexity of the system (both values results of the equations 6.19 and 6.9 are needed to be calculated in the original FGRN). The protein compound model can be made even more bio-plausible for example by using the multiplication of these two values so that it produces a complex shape controlled by the shape of the original proteins while the shape is still dynamic and changed by the concentration of the dominant protein in each domain. Nevertheless, this requires additional multiplication operations that is computationally more expensive specially in hardware.

Neurite Growth and Synapse Formation

In order to model the neurite growth and guidance, a set of specific behavioural proteins are used. Currently each soma sprouts six axonal and six dendritic growth cones at the beginning of the developmental process. However, the probability of the development of a growth cone can also be controlled by same or separate behavioural proteins (to be added to the protein types). At each development step, the likelihood of growth of the growth cone j towards side d of the glial cell (where routing resources are available) is

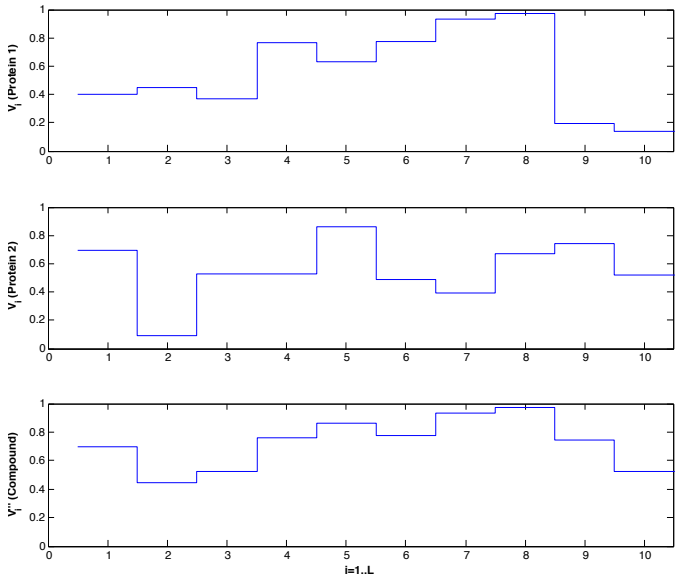


Figure 6.10: An example of the other form of the compound protein shape calculation based on the maximum value at each place of two (or more) proteins that is compatible with the original FGRN method.

calculated using:

$$\Lambda(G_j^d) = \frac{\sum_{i=1}^{L} V_i^{mg_j} \cdot V_i^{m\Delta_d}}{L} \tag{6.24}$$

where $V_i^{mg_j}$ is the ith value in the growth protein compound (merging all growth proteins tagged as *axon growth protein* or *dendrite growth protein*) in the mother cell of growth cone j, and $V_i^{m\Delta_d}$ is the ith value in the gradient compound of all proteins across side d. This gradient compound is calculated using the following equation:

$$V_i^{m\Delta_d} = C_{ij}^{\Delta_d} \text{ where } j = \underset{k, C_k \neq 0}{\operatorname{argmax}} V_i^k \quad \text{for } i = 1..L \tag{6.25}$$

This is similar to the way that protein compounds are calculated, except that the concentration gradient $C_{ij}^{\Delta_d}$ (difference across side d of the glial cell in the concentration of the protein j, which has the maximum value at location i of the protein shapes) is used instead of the maximum value $max_{k,C_k \neq 0} V_i^k$ itself. For each side of a glial cell (processed in a clockwise order), the growth cone with the highest positive likelihood will be routed towards that side. Figure 6.11 summarises the process of the neurite growth in a simple example with protein shapes of length $L = 3$.

Clearly, the likelihood of growth into soma cells and out of the right edge of the cortex must be zero. Moreover, dendrites cannot grow into IO cells. Each IO cell has an axonal growth cone in its adjacent glial cell. Axons of other soma and IO cells can also grow and connect to IO cells. Currently, no neurite branching is allowed and when a growth cone grows into a neighbouring cell, it moves to that cell and does not duplicate. However, the Digital Neuron model, the Cortex model, and the Neural Evo-Devo model allow that functionality just by adding more behavioural proteins to the system for generation of growth cones, branching, or just by setting a constant threshold for growth likelihood to detect branching.

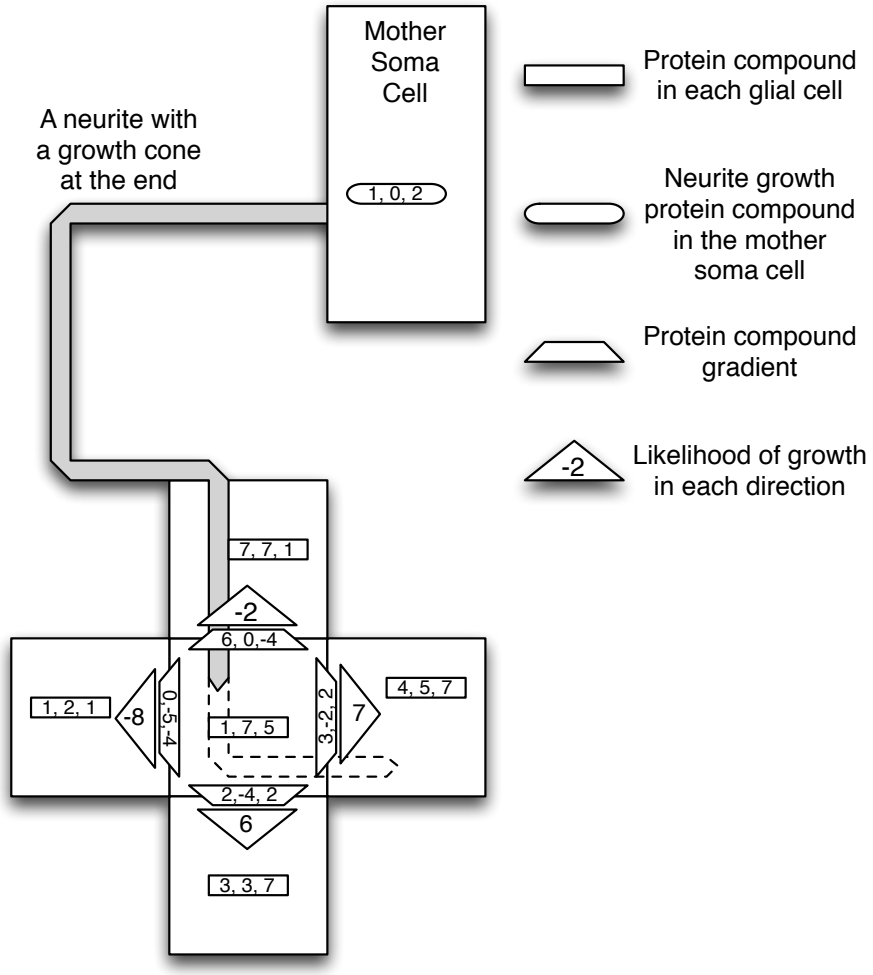


Figure 6.11: An example demonstrating the neurite growth in direction of the highest growth likelihood with very simple protein shapes of length $L = 3$. The growth likelihood $\Lambda(G_j^d)$, shown as triangles, is calculated by the inner product of the neurite growth protein compound in the mother soma cell V^{mg_j} and protein compound gradient $V^{m\Delta_d}$ in each direction d. Protein compound gradient in each direction is calculated by subtracting the shape of the glial compound protein V^m from the shape of the compound protein in each neighbour.

The formation of a synapse (given that a free synapse is available in a glial cell) between any pair of dendrite and axon in a glial cell was controlled by a probability based on the interaction between synapse formation proteins of the pre and post synaptic soma cells and the local protein compound of the glial cell. The probability $P(f|j,k,l)$ of a synapse formation between axon j and dendrite k in glial cell l is calculated as:

$$P(f|j,k,l) = \frac{1 + \tanh\left(10 \frac{\sum_{i=1}^{L} V_i^{aj} \dot{V}_i^{dk} \dot{V}_i^{ml}}{L} - 5\right)}{2} \tag{6.26}$$

where V_i^{aj}, V_i^{dk}, and V_i^{ml} are values at position i of the presynaptic formation compound protein (in the mother cell of the axon), postsynaptic formation compound protein (in the mother cell of the dendrite), and compound protein of the glial cell respectively. The synapse formation compound proteins are calculated using all the proteins that are tagged as a pre or post synaptic formation protein. The hyperbolic tangent function allows to adjust the sensitivity of the probability in a more evolvable manner and ex-

pand the distribution of the random values more uniformly. This is particularly needed since the result of the multiplication of three random values in $[0, 1]$ will be a very small number. The actual coefficients and constants (10 and 5) can be adjusted empirically. This allows the local compound protein to interact with the specific synapse formation compound proteins of the pre and post synaptic soma cells and also depend on the locality of the glial cell giving total control of the synapse formation to the evo-devo model. Every time that a neurite grows into another cell or a synapse is formed, the configurations of the associated multiplexers in the Cortex can be updated to reflect the latest changes.

General algorithm

The general neurodevelopment algorithm repeats the same procedure for all the cells in all development cycles as follows:

Initialise the cortex and arrange the soma cells
Calculate and store all protein and promoter shapes using equations 6.15 to 6.17
for all development steps **do**
  for all cortex cells **do**
    for all proteins **do**
      Diffuse protein using equation 6.18
    end for
  end for
  for all cortex cells **do**
    for all genes in the genome **do**
      Express the gene with prob. $P(E|\delta, T_A)$ and increase (or decrease) the associated concentration using equation 6.19 to 6.23
    end for
    if cell type = glial **then**
      Process glial cell:
      for all dendrite or axon growth cones in the cell **do**
        for all available growth directions **do**
          Calculate the growth likelihood using equations 6.24 and 6.25
        end for
        Grow the neurite in the direction with the highest positive likelihood
        if a synapse is available in the cell **then**
          for all pair of axon and dendrite in the glial cell **do**
            Form a synapse randomly with a probability calculated using the equation 6.26 with a similarly developed synaptic weight (assumed fixed here)
          end for
        end if
      end for
    end if

if cell type = soma **then**

Process soma cell

Calculate Soma cell parameters using behavioural protein concentrations

end if

Update the MUX configurations of the Cortex model accordingly

end for

Reconfigure the hardware platform accordingly

end for

Processing a glial cell includes synapse formation and neurite growth. Synapse formation involves checking if a free synapse unit, at least one axon and one dendrite exist in the cell and then forming a synapse between two neurites randomly with probability of synapse formation. If two different pairs of neurites were racing for synapse formation in the same glial cell in the same development step, the pair with higher probability wins. Neurite growth involves calculating the growth likelihood of all growth cones in the cell towards each side and then growing the ones with the highest non-zero likelihood. At the end, the corresponding multiplexers involved in the synapse formation and neurite growth are updated (reconfigured) accordingly.

In the case study model, the behavioural proteins are limited to the very basic behaviours of neurite growth. However, new types of proteins can be simply added to the model and the collective concentrations of the proteins of the same type (by summation, merging, or other methods) can be used to set the Cortex model parameters such as soma cell parameters. The case study provides the examples for such behavioural proteins.

6.4.2 Implementation

The neural development algorithm was implemented in a synchronous and sequential manner in software running on a PC. However, with some inter-thread coherence and synchronisation precautions, it is possible to have parallel threads for protein diffusion, gene expression, and neurite growth processes in each cortex cell. The software was written in C++, interfacing with a Matlab engine for statistical analysis and visualisation of the neural microcircuit network. For statistical analysis of the neural microcircuits, an open-source Matlab toolbox called Brain Connectivity Toolbox (BCT) [319, 318] was used.

This algorithm lends itself to massively-parallel architectures such as GPUs (Graphics Processing Units), and FPGAs. Here, parallel and distributed implementation of the neurodevelopmental processes in FPGA is discussed briefly. The protein diffusion process is a rather standard and efficient function known as Laplacian filtering and Gaussian blur in image processing and design of real-time video processors. However, the filter matrix must be generated based on the diffusion and stability coefficients of each protein. Hardware implementation of the protein-protein interactions needs a global lookup table that contains sorted protein indices based on the values in their shapes. These indices can be used as addresses to fetch the local concentration of each dominant protein for each value in the shape of the local compound protein. A custom circuit can be also designed for masking the inter-cellular signal proteins and calculation of the gene expression probabilities and protein synthesis speeds in each cell.

Stochastic computing can be used for calculating these two values to reduce the hardware cost of sigmoid functions in these computations. Similarly, custom circuits can be designed for producing neurite growth likelihood values or other behavioural functions. These behavioural proteins control the parameters and connectivity of the Cortex. Different values for different proteins in each cell can be processed sequentially as the algorithm is the same for most of them, and performance is not very critical, and also to keep the hardware cost low. If the Cortex is using a virtual FPGA method, the output of the neurodevelopmental model should be used to locally reconfigure the cell. However, if a dynamic partial reconfiguration method is used, all these local data must be gathered by the embedded system that reconfigures the Cortex. The complexity and hardware cost of a hardware implementation for such a bio-plausible model is rarely acceptable unless the underlying neural processes can be executed a few order of magnitude faster than what is possible on the current Cortex model.

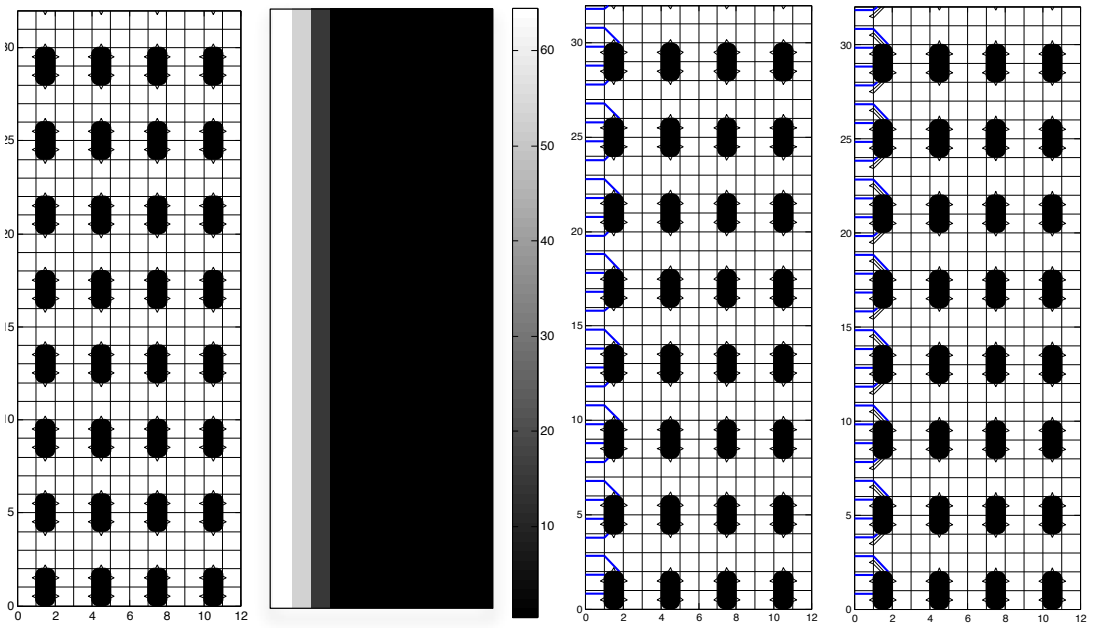
6.4.3 Verification, Testing, and Debugging

A notable challenge in verification and testing such a bio-plausible model was that black-box testing and end-to-end verifications are not very helpful if possible at all. An implementation of a bio-plausible developmental model may contain many bugs and errors but it may still appear to work. This is partly due to the high level of robustness in such systems and partly due to the fact that the correct expected output of such complex bio-plausible system is not alway known. Therefore, it is required to perform controlled unit tests by monitoring the inputs and outputs of each module in the system. For integration test, it was found useful to disable most of the functionalities and separate modules (*e.g.* diffusion or gene expression) and perform integration test by enabling each module separately, and then different combinations of the modules, until all the modules are enabled and tested together. Applying very simple inputs (*e.g.* short handcrafted genomes, protein concentrations, or maternal factors) that must result in predictable outputs is useful in both unit and integration testing.

The correct functioning of the neurodevelopmental model was tested using visualisation of the protein concentration and neurite growth patterns using Matlab and debugger features with very simple handcrafted genomes. The behaviour of the protein diffusion, gene expression, neurite growth, and synapse formation processes were tested one by one by cross checking the behaviour of the model with the expected behaviour of the handcrafted proteins and genomes using white-box testing. First only protein diffusion was tested using maternal factors and different initial concentration values. Then gene expression was tested by setting the diffusion and stability coefficients to zero and one respectively and initialising the concentrations in different cells. After verification of these modules, behavioural processes for growth of neurites and synapse formation were tested one by one in a similar manner. Then modules were all enabled one by one for integration testing.

Figure 6.12 shows the development of a phenotype from a single gene chromosome as an example of the method used for verification of the protein concentrations and neurite growth. The protein was, first, tagged as an IO cell maternal protein with protein diffusion and stability coefficients equal to 0.5 and 1.0 respectively. Other values in the gene were set to zero. Figure 6.12(a) shows the developed microcircuit and the concentration of the single protein after five development cycles. In the second step,

the same protein was also tagged as an axon growth factor. Although the protein concentration was the same in the second step, the protein also worked as an axonal guidance signal and axons were grown towards the sources of the protein, namely the IO cells (figure 6.12(b)). In the third step (Figure 6.12(c)), the protein was also tagged as a dendrite growth factor, and their growth towards IO cells was tested in this way.



(a) (b) (c)

Figure 6.12: Example of the method used for verification of the protein concentrations and neurite growth processes: (a) shows the developed microcircuit (left) and the concentration of an IO cell maternal protein (right) after five development cycles, (b) shows the developed microcircuit when the same protein was also tagged as an axon growth factor, (c) shows the same when the protein was tagged as IO cell maternal, axon growth, and dendrite growth factors.

It was noted that tuning every single parameter and setting of such complex and bio-plausible model needs comprehensive statistical analysis requiring a large amount of computation and effort. Some preliminary experiments were carried out to find some promising and useful ranges for the parameters of the model. However, it appears that a separate study might help to explore the search space and suggest better settings.

The correct implementation of the whole neurodevelopmental model in software was verified, tested and debugged successfully before further experiments.

6.4.4 Experiments

Before adding any evolutionary processes to the model, it is always necessary to verify if the new developmental model is able to produce the desired phenotypes with the expected properties at all. Three experiments were carried out at this stage:

- Experiment 1 - Network Characteristics: To examine the suitability of this model for development of useful neural microcircuits based on the statistical analysis of their network characteristics.

- Experiment 2 - Modularity and Scalability: To investigate the possibility of developing repeating connectivity patterns and motifs that are necessary for scalability of the Cortex.
- Experiment 3 - Fault-tolerance: To examine if the developmental processes can show the very basic signs of fault-tolerance by avoiding to use faulty cells.

The objective and setup of the experiments, and their results are reported here. The protein size (L) and max development cycles were set to 10 and 200 respectively in all these experiments.

Experiment 1 - Network Characteristics

The aim of this experiment was to check the possibility of growing useful networks using the new neurodevelopmental process. Brain networks and animal nervous systems show the properties of small-world networks, that is higher clustering coefficients and shorter characteristic path lengths (average shortest path between any two nodes) compared to random networks [39]. Three sets of 1000 networks were developed using three different neuron placement patterns of 120 neurons in a 120×12 Cortex with randomly generated genomes of length 16. The real values in the genes were set to random numbers in range $[0, 1]$ and the protein types were set to random binary strings. The characteristic path length and clustering coefficient of the all the resulting networks were recorded.

Results

Figure 6.13 shows the distributions of the characteristic path lengths and clustering coefficients, along with the distribution of their ratio of the developed networks with three different neuron arrangements. All the distribution histograms are cropped at the top to show details, as peak values of the histograms are not indicative of the desired network characteristics. All three arrangements showed almost the same distribution of characteristic path length with a fat tail on the left side, meaning that developing networks with short characteristic paths is possible using this system. The clustering coefficient was slightly influenced by neuron arrangement and some networks with clustering coefficients of greater than 0.5 was recorded in case of the third arrangement. Development of networks with both a high clustering coefficient and a short characteristic path length at the same time is captured in the distribution of the ratio of these two, shown in the third column of figure 6.13. The right tail of the distribution of this ratio showed that generation of such networks with this neurodevelopmental process is possible. The characteristic path length, clustering coefficient and their ratio of some of the generated networks were similar to statistics of the nervous system networks reported in [39], namely those of *C. elegans*. A visualisation of a section of a microcircuit developed from one of the random genomes is shown in figure 6.14.

Modularity and Scalability

A very simple genome of five genes was handcrafted to demonstrate how this genotype-phenotype mapping lends itself to emergence of scalability and modularity. Figure 6.15 shows a schematic of the gene regulatory network of the designed genome. Soma and IO cells each have a maternal factor of their own ($\mathrm{M_S}$ and $\mathrm{M_{IO}}$), which is sustained at saturation level by positive feedback loops of gene 1 and 2 respectively. These maternal factors have a diffusion coefficient of zero, meaning that they are internal

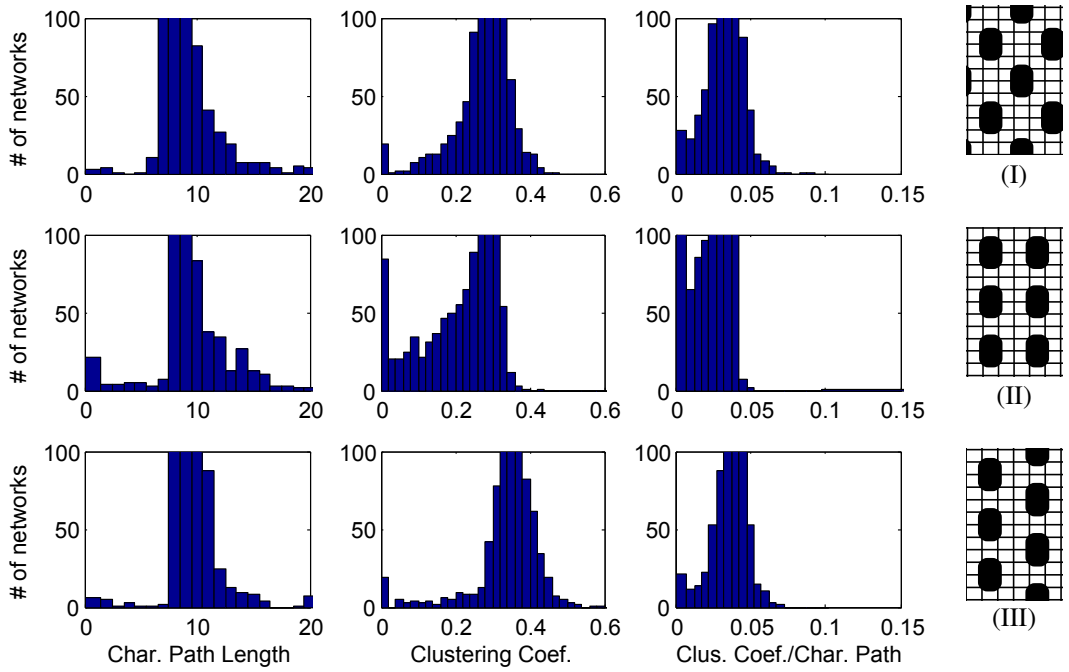


Figure 6.13: Distribution of the characteristic path length, clustering coefficient, and their ratio for 1000 developed networks using randomly generated genomes with 3 different neuron placement patterns (I, II, and III).

proteins and cannot cross the cell membrane. Each of those maternal factors exactly match and enhance the promoters of the gene 3 and 4, which lead to synthesis and diffusion of intercellular signal proteins (S_S and S_{IO} - with diffusion coefficients of 0.5 and 0.99). The soma cell maternal factor also exactly matches the promoter of the gene 5, which synthesises another internal protein in soma cells (F_G - with diffusion coefficient of zero) that works both as axon growth factor and dendrite growth factor. The shape of this growth factor can interact with the merged gradient of intercellular signal proteins diffused form both soma and IO cells resulting in growth probably P(G). This genome was used to develop networks using two different cortex sizes (12×12 and 12×24 with 9 and 18 neurons).

Results

Figure 6.16 shows the results of the second experiment for two different cortex sizes of 12×12 (figure 6.16(a)) and 12×24 (figure 6.16(b)). The same connectivity motif was repeated vertically for both cortex sizes demonstrating a very simple but scalable mechanism using a minimal genome. Figure 6.16(c) shows the diffusion patterns of the five proteins in the Cortex at the end of the development process.

Fault-tolerance

The aim of this experiment was to demonstrate the basic fault-tolerance capability of the neurodevelopmental model. For this experiment, another simple genome was designed that simply grows an axon from one neuron to another neuron. The IO cells' maternal factor (also an intercellular signal protein) was used to initiate the cell differentiation of two neuron types as its concentration would be higher in the left neuron (bottom-left concentration pattern in figure 6.17). This differentiation is clear in the top-right

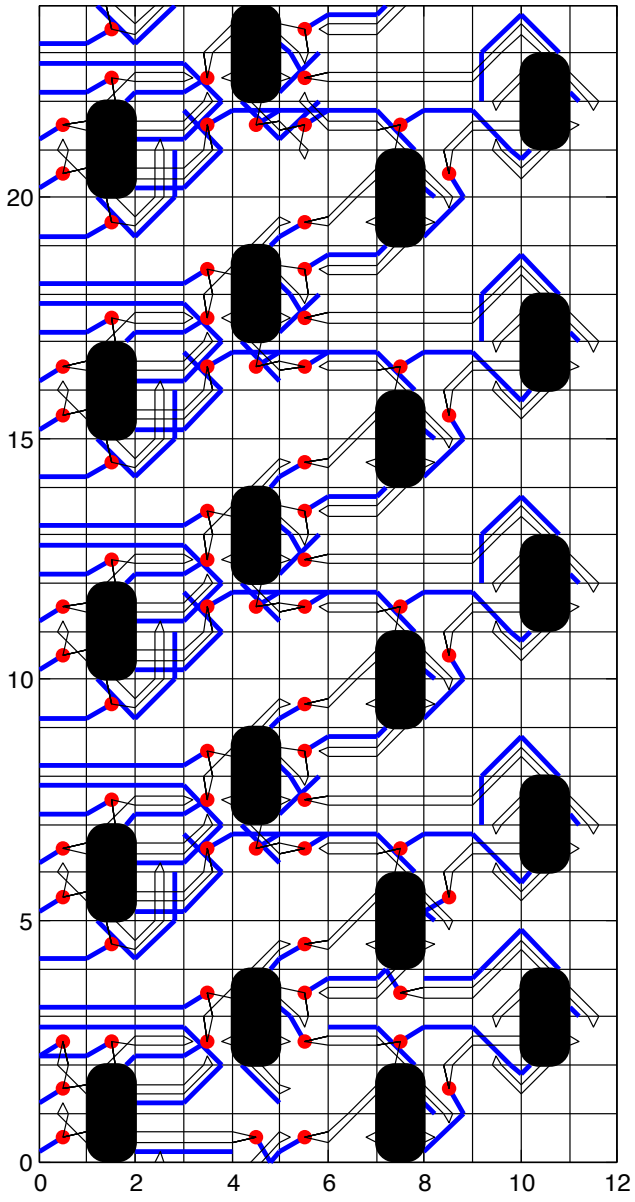


Figure 6.14: A visualisation of a section of one of the neural microcircuits developed from a random genome in experiment 1. Axons and dendrites are shown as bold blue and light black lines respectively. Red dots are representing synapses.

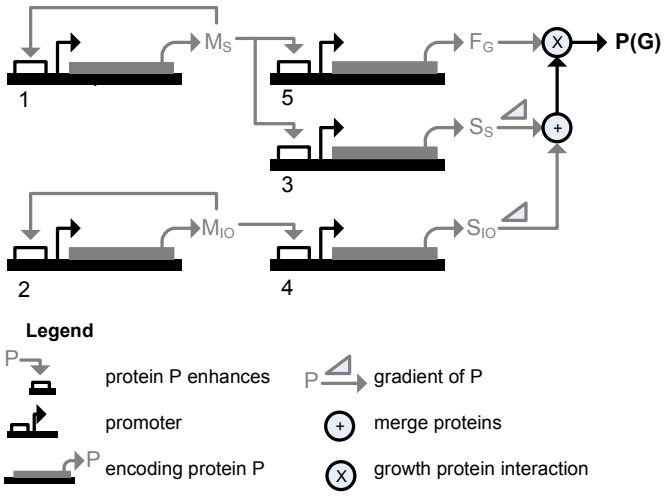


Figure 6.15: Gene regulatory network of the designed genome showing the gene-protein and protein-protein interactions.

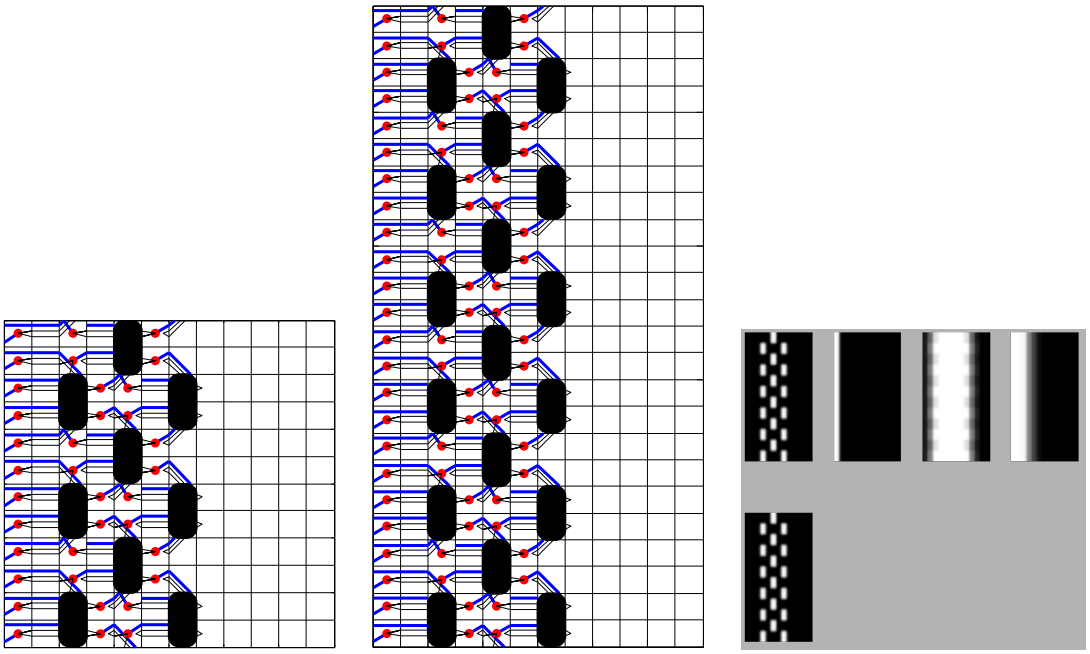
protein concentration pattern in figure 6.17(a). The neural microcircuit was developed and the routing path of the axon was recorded. In the second step, a glial cell in the axon routing path was randomly selected and tagged as "faulty" in order to simulate the effect of a fabrication fault. It was assumed that "faulty" cells are detected either dynamically by a health or activity signal, an error detection mechanism, or by using a post-fabrication test prior to starting the developmental process. In this experiment, a "faulty" cell simply does not involve in the protein diffusion process (setting all protein concentrations in the cell to zero). Therefore, the likelihood of neurite growth into that cell will be always non-positive. It was anticipated that the axon should deviate from its original path and bypass the "faulty" glial cell.

Results

Figure 6.17(a) shows the single axon grown from one neuron to the other along with the diffusion pattern of six proteins in the cortex after normal development. The glial cell, which is selected to be the "faulty" cell in the second step is labelled with a square in figure 6.17. In the second step (figure 6.17(b)), the glial cell was turned off as "faulty" and development process was rerun. Concentration levels of all proteins in the "faulty" cell were equal to zero. The effect of the "faulty" cell in the protein diffusion pattern is notable in the diffusion pattern of the third protein (figure 6.17(b) top-right). Consequently, the axon avoided the "faulty" glial cell, and connected to the target neuron through another path. Similar behaviour was observed in case of a few other randomly selected glial cells in the default path of the axon. This demonstrated that with a minimal and simple genome it is possible to produce the very fundamental mechanisms necessary for fault-tolerance and regeneration with such a bio-plausible neurodevelopmental model.

6.4.5 Evolutionary Model

The general evolutionary model and its features are explained briefly here as evolutionary model that was implemented in the software is not the focus of this study and it was only implemented as a supporting process for testing the neurodevelopmental model. The evolutionary model was implemented in software running along with the neurodevelopmental model on the same PC.



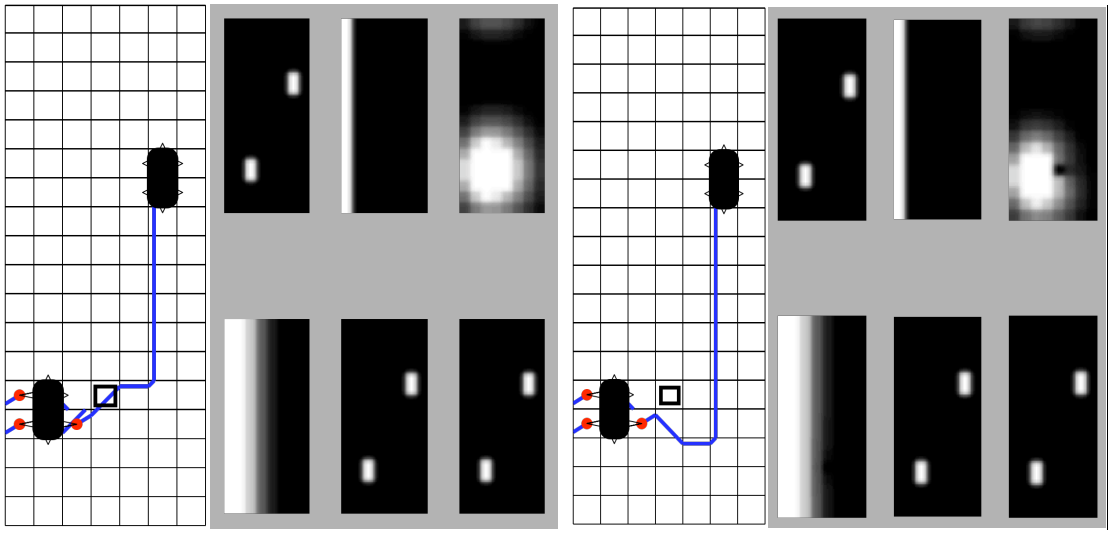
(a) (b) (c)

Figure 6.16: (a) The developed microcircuit using the designed genome, in a 12×12 cortex. (b) The developed microcircuit using the same genome in a 12×24 cortex. (c) The diffusion patterns of the five proteins in the 12×24 cortex at the end of the development.

The evolutionary algorithm used here is a flexible and quite generic algorithm adopted from [24, 25] that allows quick exploration of different evolutionary algorithms and settings. A population of adults is maintained, with n fittest individuals being used as parents for reproducing offspring. The population size is usually set to $1.25n$. In every generation, m offspring are produced and evaluated. Any of the offspring individuals that is fitter than the least fit individual in the population, will be added to the population and the least fit individual will be removed. The population is always maintained sorted based on fitness. Positive selection pressure can be achieved by smaller n values and negative selection pressure can be adjusted by changing the $\frac{m}{n}$ ratio. These parameters enable the user to implement a wide range of different evolutionary algorithms including a canonical GA algorithm (with $m = n$), and a steady-state GA (with small $\frac{m}{n}$ ratios) [24].

The initial population is generated randomly with random bit streams for gene type loci and random real numbers in $[0, 1]$ range for all other loci. Each adult has an age (set to zero when born) that is incremented by one in each generation and after reaching to a certain age (a specific number of generations) that adult will be removed from the population. This ensures that a fitter genome with a non-inheritable advantage does not dominate the population. The population size being slightly larger than n allows other adult individuals to replace dead ones when their lifespan is passed.

Every pair of parents are recombined using gene matching. Every single gene in one of the parents' chromosome(s) is matched with the most similar gene in the other parent. A uniform crossover is then used to recombine the two matched genes taking bits (for gene type locus) and real values (for all other



(a) (b)

Figure 6.17: (a) The single axon routed from one neuron to the other along with the diffusion pattern of six proteins in the cortex. (b) The axon diverted to bypass the "faulty" glial cell (marked with a black square) along with the affected protein concentration pattern.

loci) randomly from one of the genes. Both genes are then ticked off as used and the next gene in the first parent is processed in the same manner until all the genes in the second parent are used. The rest of the unused genes in the first parent are then appended at the end. The similarity measure for the genes is based on the sum of differences of the real-valued alleles of two genes. Moreover, if there are no common set bits in the gene type loci of the two genes, their similarity is equal to zero. This gene matching method allowed effective crossover of variable length chromosomes.

Four different mutation methods are used: Creep mutation, Gene duplication, Gene addition, and Gene deletion. Creep mutation adds a uniform random real value between -0.5 and 0.5 to a real-valued allele and then crops the result back to the $[0, 1]$ range. For gene type alleles, one randomly selected bit is flipped. This mutation allows small changes in the genes that may reflect as changes in the shape of the proteins, their diffusion and stability coefficients, promoter shapes, affinity or concentration thresholds, or the gene types.

Gene duplication mutation, selects two genes in the offspring chromosome by chance, and copies promoter or coding region of one of the genes to promoter or coding region of the other gene. This mutation method allows genes to produce the same protein, be triggered by the same group of proteins, or one being triggered by the other, which promotes the modularity of the gene-regulatory network and its evolvability.

Gene addition mutation adds a copy of a randomly selected gene from the offspring's chromosome to the end of the chromosome if it is shorter than maximum length allowed for the chromosomes. Gene deletion mutation, deletes a randomly selected gene from the offspring's chromosome. These mutations allow for growth and shortening of the chromosomes required in a variable-length chromosome.

Here, each parent is selected with equal probability from the top n fitter individuals of the popu-

lation. However, it is easily possible to use any other fitness-proportionate selection method (Roulette Wheel) or other methods since the population is always maintained sorted based on fitness values. It is also possible to use genetic similarity measures and crowding techniques similar to what is used in NEAT or [243] for promoting speciation and maintenance of the population diversity.

Implementation, Verification and Testing

The complexity and robustness of the bio-plausible models of development and evolution makes it very difficult to perform black-box or end-to-end integration tests. For example, at some point during testing the evolutionary model, it was revealed that the parents were being selected from the less-fitter end of the population sorted list. However, since the population was a selected group of adult and offspring individuals with higher fitness values, the maximum and mean fitness of the population was still increasing consistently during evolution. It was only after detailed debugging and inspection that such a programming mistake was revealed. In such cases, a bio-plausible model is so rich and robust to faults, errors, and even programming mistakes, that it still works with a lower performance and it is very difficult to detect the problem. Therefore, due care and proper module testing are required in the testing phase of such bio-plausible systems.

Different functions of the evolutionary algorithm were verified and tested using a debugger with a small population and very short initial chromosomes. Again as in verification and testing of the developmental model, white-box unit testing and integration testing was performed on each module of the system. To test the selection and recombination, all the mutations were disabled and the result of each recombination and selection was monitored and verified by hand using the debugger. Then mutation methods were enabled and their functionality were verified one by one.

After careful testing and verification of all the modules in the evolutionary model, a number of preliminary experiments were carried out to find useful ranges for parameters (*e.g.* population, selection and generation sizes, mutation probabilities, etc.). A few bugs and implementation errors were found and fixed during verification, testing, and parameter tuning process. The evolutionary model was finally successfully verified, tested, and ready for the following experiments.

Experiments

To test the integration of the developmental and evolutionary models an experiment was designed and carried out. The goal of the experiment was to verify if the developmental representation was evolvable and evolutionary process was able to evolve the connectivity of the neural microcircuits towards networks with higher clustering coefficients and shorter characteristic path lengths, similar to biological nervous system networks. In this experiment the fitness of the individuals were calculated using the statistical analysis of the neural microcircuits rather than results of the neural processes. The ratio of the clustering coefficient to characteristic path length of the networks were used as the fitness of the individuals. Table 6.3 reports the parameters and settings used in this experiment. The best and average fitnesses, and chromosome length of the best fit individual and average chromosome length of the population and total number of evaluations were recorded at each generation. A visual representation of the best fit phenotype in the cortex was produced every 10 generation. The experiment was repeated 32 times.

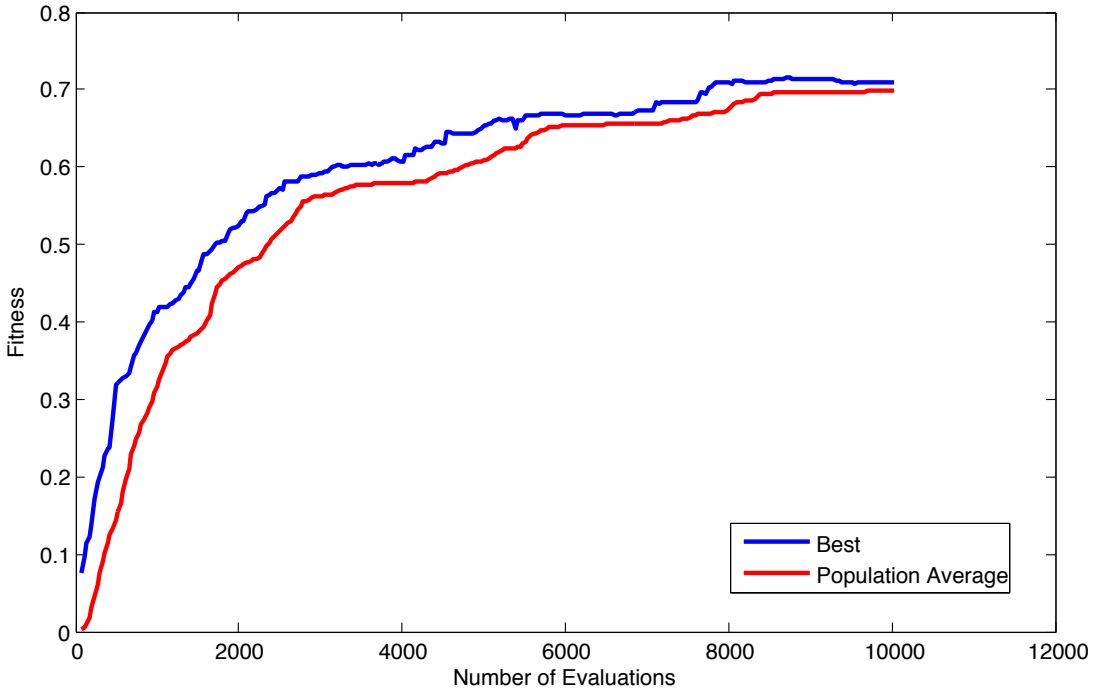


Figure 6.18: Best and average fitness of the population during 312 generations against the number of evaluations (averaged over 32 runs).

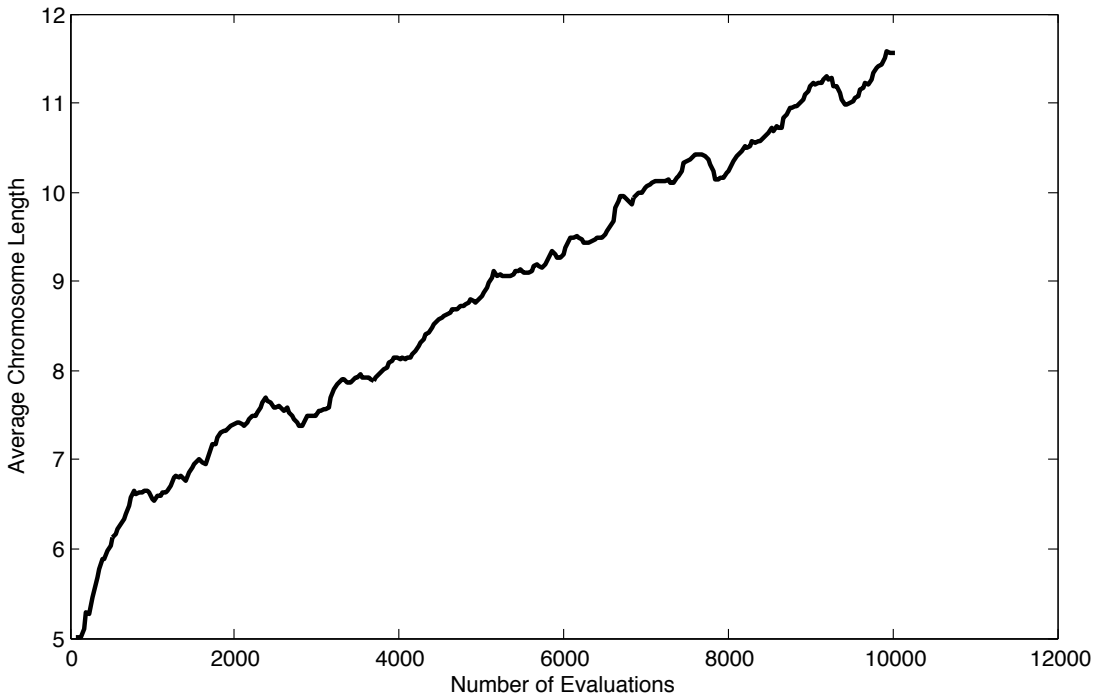


Figure 6.19: Average chromosome length of the population during 312 generations against the number of evaluations (averaged over 32 runs).

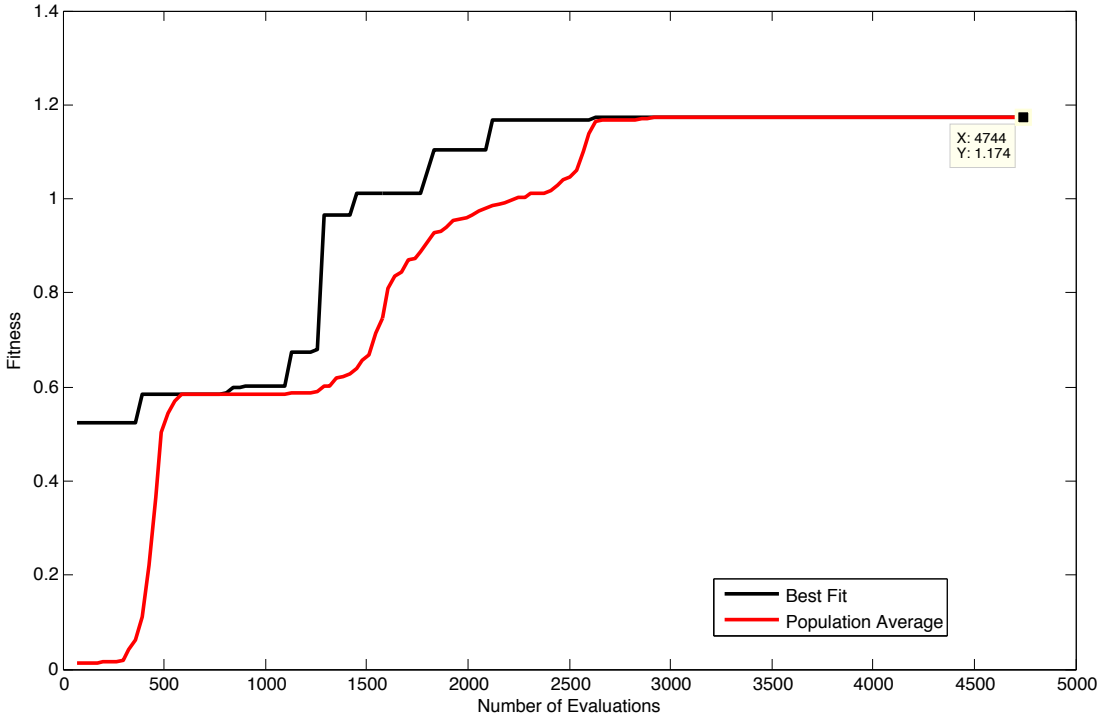


Figure 6.20: Best and average fitness of the population during an example run displaying a convergence and plateau at the end suggesting a stagnation.

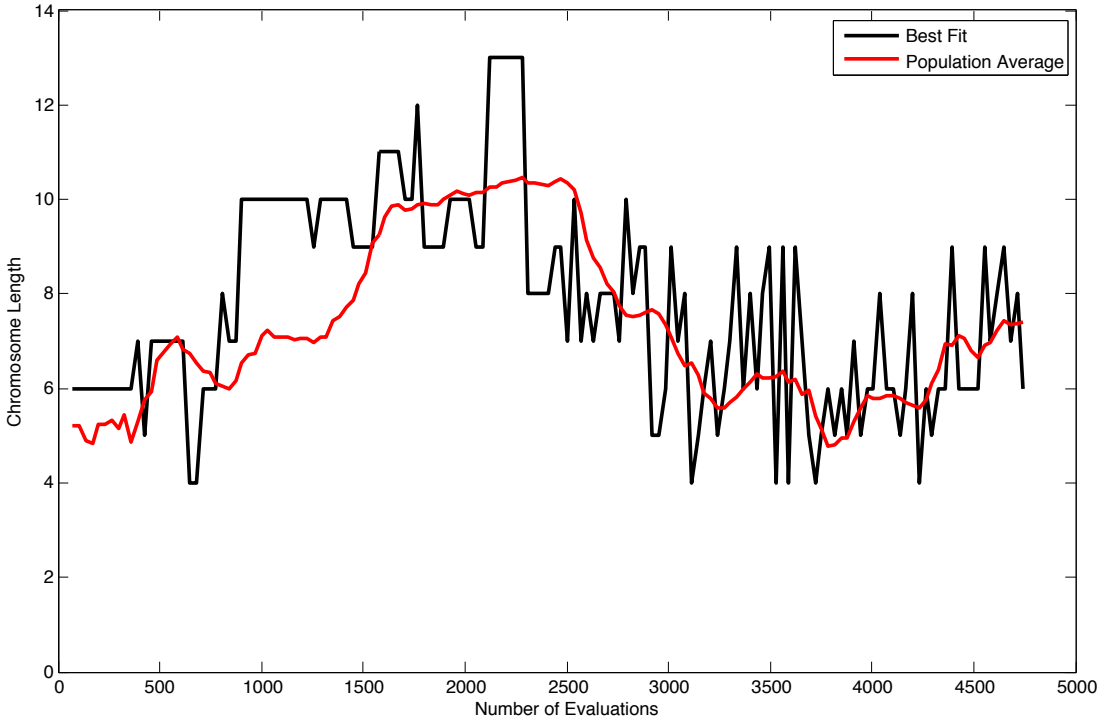
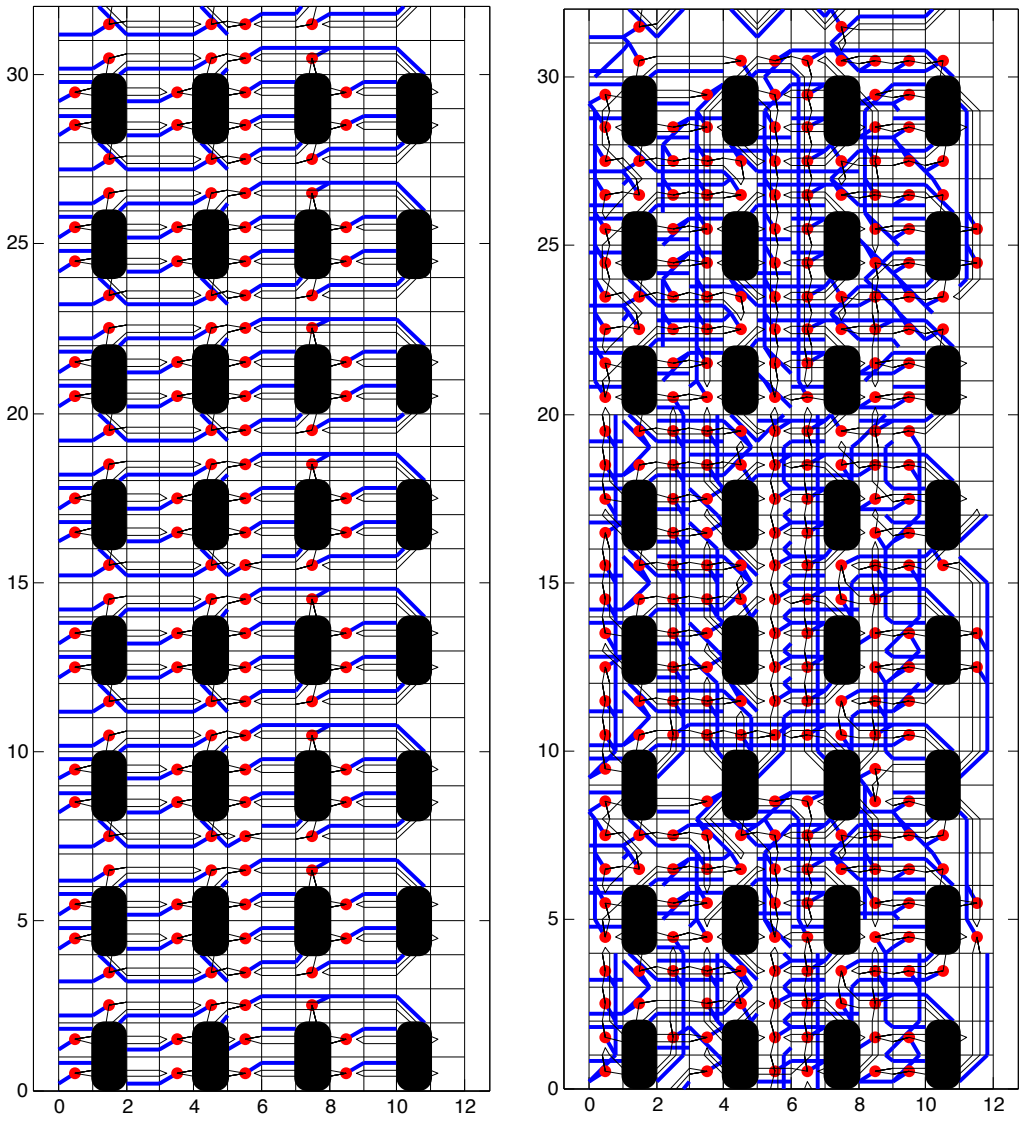


Figure 6.21: Chromosome length of the best fit and population average during 312 generations in the example run showing significant changes during the fitness plateau.



(a) (b)

Figure 6.22: Two example best fit microcircuit phenotypes after the fitness convergence of the example run (at about 4000 evaluations). (a) The fittest microcircuit of the population at generation 131 with eight clusters of four neurons. (b) The fittest microcircuit of the population at generation 141 with larger and more complex clusters but with the same fitness (mean clustering coefficient to characteristic path length) ratio as (a) equal to 1.17.

Table 6.3: Parameters and settings used in the evolutionary model experiment.

Parameter or setting	Value	Unit
Cortex size	12x32	Grid cells
Number of neurons	32	Neurons
Neuron placements configuration	II (in figure 6.13)	-
Development length	30	Cycles
Protein length (L)	50	-
Initial chromosome length	5	Genes
Maximum chromosome length	50	Genes
Population size	40	Individuals
Selection size (n)	16	Individuals
Generation size (m)	32	Offspring per generation
Life span	25	Generations
Crossover probability	1.0 (always)	Per pair of parents
Creep mutation probability	0.05	Per locus
Gene addition mutation probability	0.5	Per chromosome
Gene deletion mutation probability	0.5	Per chromosome
Gene duplication mutation probability	0.1	Per chromosome
Number of generations	312	-
Number of runs	32	-

Results

Figure 6.18 shows the best fitness and average fitness of the population during 312 generations averaged over the 32 different runs. Figure 6.19 shows how the average chromosome length of the population was changing on average during these 32 runs. They exhibit typical fitness curves of the evolutionary algorithms. The population average chromosome length shows a consistent increase during evolution that can be a sign of complexification or bloating.

Looking at a relatively successful example run (figures 6.20 and 6.21) allows us to examine the situation more closely. The average chromosome length appears to be increasing when the average fitness is increasing, suggesting complexification and that useful inheritable genetic material is being generated. When the average fitness is not increasing, initially, the chromosome length decreases, which suggests that the evo-devo model is not necessarily bloating and it might be able to generate compact and modular representations for the neural microcircuit. After the initial compacting phase the evolutionary process appears to explore the search space through neutral mutations. This can be a sign of what is already shown previously in FGRNs as evolution of robust and efficient GRNs [24]. During this phase evolutionary process can explore the search space for more robust and fault-tolerant genomes that develop into

a phenotype with the same fitness value. Although the fitness plateaued after 3000 evaluations and the evolutionary process appeared stagnated, looking at the developed neural microcircuits revealed that the neutral mutations were at work to keep the population both genotypically and phenotypically diverse. Figures 6.22(a) and 6.22(b) show the phenotypes of the best fit individuals from two populations a few generations apart at about 4000 evaluations. As it is clear in figure 6.22(a), evolution has reached a completely partitioned network of eight separate but internally highly-connected clusters with a fitness (clustering coefficient to characteristic path length ratio) of 1.17. However, a few generations later, the best fit individual has a completely different phenotype with the same fitness but with a single-partition network of dense and highly clustered modules. These two phenotypes have the same fitness and the second phenotype was transiently the best solution for a few generations before the first phenotype reappears as the fittest individual in the population again. This was due to the fact that the fitness function did not care about the number of partitions and other factors of the neural microcircuit and the evo-devo model was only satisfying the given fitness criteria. However, the neutral mutations allowed the evo-devo model not to stagnate and explore different network architectures.

6.5 Practical Considerations

In this case study, no feedback is provided from the Cortex to the developmental processes. One type of feedback data is activity related that requires simulation of the neural model to generate data about health and activity of the soma and glial cells. However, a second type of feedback that can be used by the developmental processes is feedback about available routing resources during development. This type of feedback is very important for evolution of robust and efficient routing since it provides local information to each cell about available routing resources around it. Without such clues, development has to route an axon or dendrite all the way to an obstacle when it hits the obstacle and since no routing resources are available in that direction, the next promising available direction will be used for growth. This results in wasting a lot of resources that can be otherwise used for other signals. This condition can be seen in some example phenotypes (see crooked axons in figure 6.22(b) for example). To resolve this issue, the diffusion process can be modified so that some intercellular signal proteins can be only diffused when a routing resource (for dendrite or axon) is available. This way the Cortex model can provide feedback data to the the evo-devo model about available routing resources to evolve routing strategies along with the neural connectivity patterns. This is simple to add (both in hardware and software) by setting the diffusion of the some specific type of signal proteins, to zero in the direction of the routed signals (unavailable routing resources) in each cell during the protein diffusion phase of the algorithm. This way these signal proteins will only diffuse against the direction of available routing resources giving directional clues to the growth cones.

Another issue in the phenotypes is allocation of many synapses and routing resources for repeated connections between two neurons. This can be partly resolved by controlling the synapse formation with a behavioural protein that can be associated with the proteins in the pre and post-synaptic neurons as already discussed in the previous sections. Another bio-plausible mechanism is what is known as lateral inhibition that prevents repeated connections between two neurons after the first is connection

established or causes elimination and retraction of the redundant synapses and neurites [404]. This requires the formation of a synapse to also produce a feedback to the neurodevelopmental processes both at the site of the synapse (glial cell) and in the pre and post-synaptic soma cells. Then evolution can produce genes that will be suppressed in a soma cell when its connection with a specific other neuron is established and it will prevent more growth toward that cell or further synapse formation with that cell. The evolutionary model can also produce genes that will synthesise proteins to eliminate other synapses and retract their neurites.

A feature that can help with the efficient use of the routing resources is regulating the growth cone formation by developmental processes. In the case study model, each soma cell has 6 axonal and 6 dendritic growth cones. These growth cones, rather than being regulated by developmental processes, are already formed prior to the first development cycle. One solution is to form growth cones only when the gradient of a dedicated behavioural protein is higher than a threshold. Also branching neurites and retracting them, apart from being bio-plausible, might improve the efficiency of the system as well. This can be implemented using a threshold or more complex protein-protein interactions involved in the process of neurite growth as discussed in the previous sections.

Another improvement is to code the constants related to protein diffusion in the gene using the protein folding mechanism instead of storing them in the gene directly. This allows the evolution to use more robust or very volatile representations for these values that make them more sensitive or resilient to mutations. Similarly affinity and concentration threshold values can be also coded using protein folding instead of direct encoding in the genes. Devising a similar method for coding the gene type might also prove useful in the same way.

Setting the soma parameters and synaptic weights (and their signs, for excitatory and inhibitory synapses) using behavioural proteins can also significantly improve the efficiency and accuracy of the developmental model in generating useful neural microcircuits that can actually work. Otherwise, evolution has to use the number of synapses between two neurons to adjust the synaptic weight between them.

To increase the overall performance of the evo-devo model, it is also possible to adopt some of the techniques that are used in the abstract models. Abstract models use Cartesian positional information as two static inputs to the dynamical system, which allows the developmental mechanism quickly use that information to produce morphogens. The same trick will help a multicellular model to speed up the evolution. The simplest way is to add two static maternal factors that have gradients in the x and y directions of the Cortex. The GRN will be free to quickly build upon that fixed positional information but this will not be responsive to the dynamic changes in the substrate such as faults as discussed in section 6.2.1. A more robust option would be to have initial gradients assigned to the concentration of these two maternal factors but allow them to dynamically reflect the condition of the Cortex (such as faults, etc.). This will slightly add to the computational cost of the system as these maternal factors need to be processed for diffusion and synthesis with the other proteins while in the former method they would be exempt from these processes. An even more bio-plausible, flexible and evolvable option would

be to only initialise these maternal factors in the edges of the Cortex and allow diffusion process to take care of producing the gradient. It is also possible to add handcrafted genes to the initial population (seeded population), which maintain these maternal factors. Similarly, handcrafted genes that already differentiate neurons in different positions in the Cortex may help to speed up the evolution. A more bio-plausible option would be to evolve robust GRNs that do such differentiation and use the evolved population (or other useful genetic material) as the seed population for other complex problems. These tricks all allow us to save some computation that will be needed to create these basic genetic material at the beginning of an evolutionary run. This is something that was successfully applied to FGRNs in [25].

In practice, the trade-off between bio-plausibility and simplicity appeared to be very significant in verification and testing. Comparing and benchmarking of the performance, scalability, evolvability, fault-tolerance and other features of such complex bio-plausible models with slightly different designs or parameter settings also appeared to require much more effort than what is needed for less bio-plausible models.

6.6 Summary

Figure 6.23 shows a graphical representation of the investigations carried out in this chapter. First, the general impact of the evo-devo model design on the bio-plausibility and feasibility of the whole system and its significance were discussed and highlighted. In section 6.1, general definition of bio-plausibility and feasibility measures from chapter 2 were translated into a set of tangible general design factors and constraints in the specific context of the evo-devo model. Using those general factors, different general design options and approaches and their trade-offs in different aspects of the evo-devo model design were investigated in section 6.2. Different major functions of the evo-devo model were categorised as a dynamical system for gene-regulatory network, a genetic representation and mapping, genetic operators evolving the genome, and selection mechanisms. Abstract models of the dynamical system, their bio-plausibility, and feasibility were compared with the multi-cellular (cell-chemistry) models with local interactions. Focusing on more bio-plausible multi-cellular models, a few representatives from the literature of the genetic representations were examined and their bio-plausibilities and feasibilities were compared. As the implementation of the evolutionary part of the model in FPGA proved to be inefficient (compared to implementation in software) and thus out of the scope of this study, it was briefly discussed with a quick review of a few bio-plausible genetic operations and selection methods. Different implementation methods of different parts of the model in software and hardware were investigated and compared in terms of feasibility and bio-plausibility. In the summary of the challenges and trade-offs of different design options and approaches, it was confirmed that the general trend of the bio-plausibility-efficiency and bio-plausibility-simplicity trade-offs exist in the design of the evo-devo model. However, unlike the general trend, the scalability and reliability measures of feasibility appeared to increase with the bio-plausibility of the evo-devo model.

Based on the insight gained from this analysis, a neural evo-devo model was designed as a case study. The trade-offs highlighted in this analysis showed that, with the time and hardware resource constraints of this project, it would be more beneficial to design and implement a bio-plausible model

in software (having hardware compatibility in mind) than implement a much less bio-plausible model in hardware. A multi-cellular version of the FGRN was designed as the case study neurodevelopmental model and implemented, verified, and tested to demonstrate its fundamental bio-plausible network characteristics, scalability, modularity, and fault-tolerance capabilities using randomly-generated and hand-crafted genomes. A generic evolutionary model that was already used with FGRN was adopted as the case study evolutionary model, and was implemented, verified and tested to demonstrate that the whole integrated evo-devo model is able to evolve neural microcircuits towards some bio-plausible network characteristics. Practical challenges and possibilities in the design, implementation, and testing of the model was discussed at the end. Verification and testing of bio-plausible evo-devo models, and comparing and benchmarking different designs and parameter settings were highlighted as one of the major practical challenges in the design and implementation of such models.

Now that all the three main models of the whole evo-devo neural microcircuit system are investigated and case study models are designed, implemented, and tested, it is time to investigate the integration of these models. The next chapter is dedicated to investigation of the challenges and trade-offs in integration of the neuron, cortex, and evo-devo models into a single evo-devo system that can solve a problem.

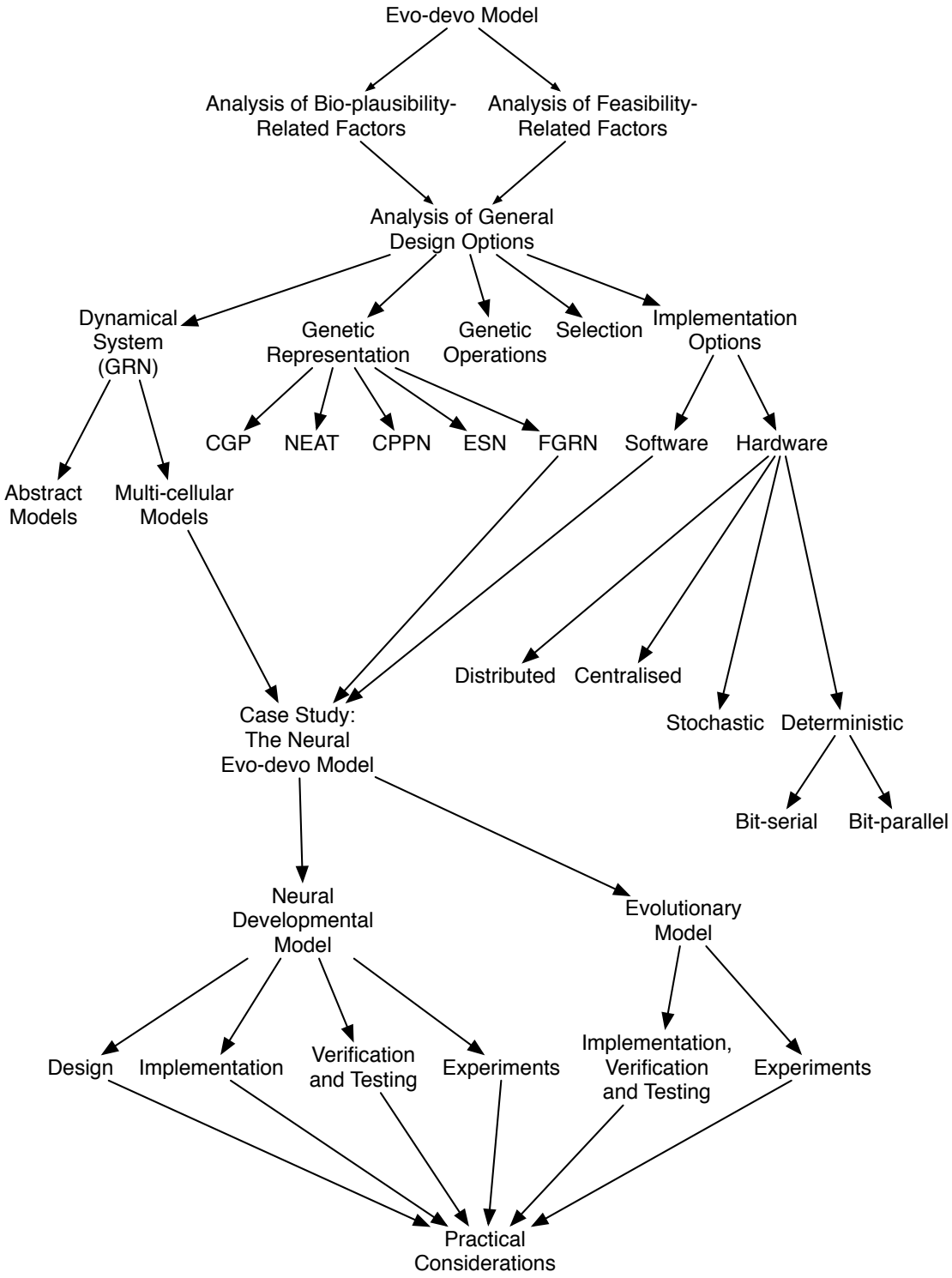


Figure 6.23: A graph of the investigations carried out in chapter 6 regarding the evo-devo model.

Chapter 7

System Integration

In the previous chapters, first, challenges and trade-offs in the selection of a hardware platform, and then challenges and trade-offs in the design, implementation, and testing of neuron, cortex, and evo-devo models were investigated one by one. Example designs for each one of these models were carried out, implemented, and tested separately. Here, the challenges and trade-offs in the integration of such models into a bio-plausible system of evo-devo neural microcircuits in FPGA are investigated and discussed.

In the integration step, a designer needs to decide about processes and submodules of the system that glue bio-plausible models together; decisions such as how a fitness value must be generated from the neuron model activity and passed to the evolutionary model, how the reconfiguration commands or activity data must be passed between developmental model and the cortex model, or how to distribute different processes of different models over available computing resources (CPUs, GPUs, FPGA). Some of these decisions mainly depend on the application and each specific scenario in practice. However, some of the design options have significant impacts on the feasibility measures of the whole system (such as performance or scalability). In some cases these options may also affect the bio-plausibility of the system.

In the following section, the general feasibility and bio-plausibility measures defined in chapter 2 are translated into tangible design factors in the context of the system integration design. Then, in section 7.2, different options and approaches in the integration of the system are investigated and discussed and their general trends, trade-offs, and constraints are highlighted. Based on that analysis and constraints of this project, integration of the models that were designed and implemented in previous chapters is presented here as a case study in section 7.3. A final experiment is reported and the practical considerations are discussed at the end.

7.1 General Design Factors

Most of the design factors are investigated and covered in the previous chapters in the context of each one of the neuron, cortex, and evo-devo models. The only design factors that are left to be discussed in the context of the system integration are related to where these models interact with each other, or those factors that depend on the system application. For example the fitness function selection, fitness evaluation process and its partitioning between software and hardware are questions that arise in the

system integration step. Figure 7.1 illustrates an abstract data flow diagram of the system, showing the interactions between different models and how both the developmental model and fitness function evaluation module can be implemented in any mixture of hardware and software. The integration of the cortex and neuron models, which are both implemented in hardware, is already covered in chapter 5. Similarly, the evolutionary and developmental models were investigated as an integrated evo-devo model in chapter 6. All the rest of the factors in the design and implementation of the fitness evaluation and interaction of the models that are partitioned between hardware and software can impact both the bio-plausibility and feasibility of the whole system. In the following, both bio-plausibility and feasibility related factors and constraints are highlighted in turn.

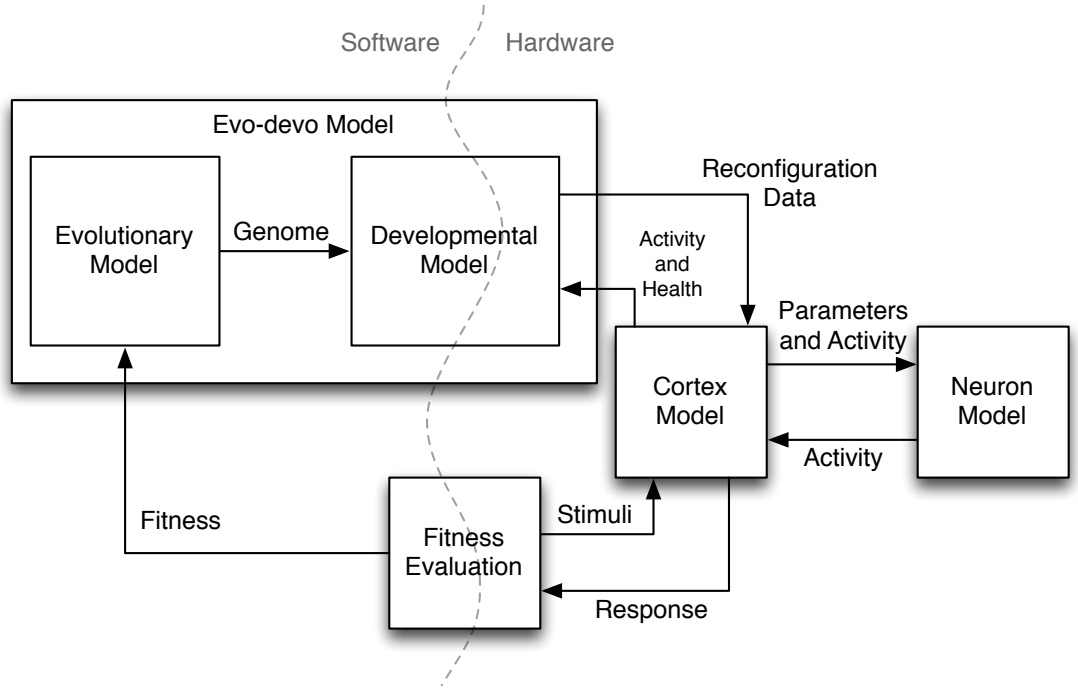


Figure 7.1: An abstract data flow diagram showing the interactions between different models in the system and demonstrating the hardware-software partitioning problem.

7.1.1 Bio-plausibility Related Design Factors

Both the bio-plausibility of the means of interactions between system models and modules and the bio-plausibility of the fitness evaluation process can affect the bio-plausibility of the whole system. In general, it is desirable to imitate the same structure of the biological systems in the interactions of the models. However, this can lead to a structurally very complex system in case of fitness evaluation module.

The fitness is not an explicit measure in biology. Many processes are involved in the interactions of an organism with the environment and other organisms that affect its chance of reproducing offspring that can transfer its genetic heritage to the future generations. The structural accuracy of the fitness evaluation mechanism and its compatibility with the biological models available [165] are desired. However the focus here is on the factors in the design of that part of the fitness evaluation processes that must be

implemented in hardware. The other factors lie in the domain of an evolutionary model that is not constrained by the limits of hardware implementation, which has been the subject of research for a long time with a plethora of literature already available [165, 180, 302].

7.1.2 Feasibility Related Design Factors

Factors Affecting the Performance

The general performance of the whole system depends on the evolvability of the model and how many evaluations are needed to find an acceptable solution for the application problem. But the time that each fitness evaluation takes apart from the development and simulation times, also depends on the time for communication between different models. Data transfer delays, overheads, and bottlenecks are major factors in the total performance of the system as evo-devo systems usually need many many evaluations to evolve something useful.

Factors Affecting the Hardware Cost

The interfaces between different models and modules of the system and also the hardware needed for implementing the fitness evaluation module can all add to the hardware resources needed on the FPGA. Hardware resources are limited and it is always desired to minimise the total hardware cost and specifically minimise the FPGA hardware resources need for the interfacing and fitness evaluation.

Factors Affecting the Reliability

Single points of failure, inaccuracies and errors in data communication can impact the reliability, robustness and fault-tolerance of the whole system. Moreover, all different aspects of the reliability must be considered in the design and implementation of the fitness evaluation module so that it not only impair the reliability of the system, but also adds to its robustness. Insensitivity of the system to the noise in the training data used for the fitness evaluation is a good example of the reliability factors related to the system integration.

Factors Affecting the Complexity

The integration and the design of the fitness evaluation module need to follow a modular design approach that produces a manageable and testable system. As already shown in previous chapters, testing bio-plausible systems turns into a major issue as their complexity grows. A modular design allows us not only to test and verify the modules separately, but also perform integration tests and verifications for any two of the integrated modules separately.

Factors Affecting the Scalability

Anything in the integration of the system that causes the performance, hardware cost, complexity, or reliability of the system to be significantly impacted by growing the size of the cortex or the complexity of the problem must be avoided.

7.2 General Design Options

Figure7.2 shows a flowchart of the system processes in its abstract and general form. Although these steps are not necessarily executed sequentially as shown in this flowchart, it helps to analyse how the

whole system can work. Each one of the main processes in this flowchart and some very general options are briefly discussed before focusing on the details.

Initial Configuration

At the system start-up, the FPGA chip needs to be configured using the initial configuration bitstream. This bitstream can be stored in the PC storage or in a non-volatile (*e.g.* EPROM or flash memory on the FPGA board or on the FPGA chip. In case of using a host PC for initial configuration, it can be carried out simply by running a standard console application (usually provided by FPGA manufacturer) on the PC. During this process, the FPGA will be programmed to create the hardware of the cortex and any embedded soft processors or IO circuitry needed in the FPGA. After this stage, the FPGA will be ready for evolution. The speed of this initial configuration is not critical as this is only performed once in an evolutionary run.

Evolutionary Process

The evolutionary algorithm starts either with a population of random genomes or is seeded using a set of pre-evolved (or hand-crafted) genomes. Each individual solution is initially developed for a number of development cycles (see Initial Developmental Process below). The resulting developed neural microcircuit will be translated into reconfiguration data (see Reconfiguration Process below) sent to the FPGA chip to route the neural microcircuit and set the cell parameters. Then, the neural microcircuit that is configured in the cortex is simulated inside the FPGA. During simulation phase the stimuli (input vectors to its inputs) is fed into the cortex and responses (output vectors) and activity data are received from it (see Neural Simulation Process below). Optionally, the activity data can be used in the next activity-dependent development (see Activity-dependent Development Process below) cycle followed by another cycle of reconfiguration and neural simulation. These cycles of development, reconfiguration and simulation can repeat for a number of times. Finally, the cortex responses are used to calculate the fitness of the microcircuit (see Fitness Evaluation Process below). Each new individual in the population will be developed, reconfigured, simulated and evaluated in this way, and the next generation of microcircuits is reproduced based on the fittest individuals as explained in the previous chapter.

Initial Developmental Process

Each individual can be developed initially for a number of development cycles. This is similar to initial development in biology before any activity-dependent development and plasticity takes place in the nervous system. The number of initial development cycles can be fixed or evolved (for example using the global concentration of a specific protein, or directly encoded into the genome). Developmental process generates cortex reconfiguration data that set the cell parameters, and routing resources in the cortex.

Reconfiguration Process

The configuration data from the developmental process is sent to the reconfiguration process. This process comprises putting the FPGA in reconfiguration mode, translating the cortex reconfiguration data from developmental process into FPGA reconfiguration commands, and reconfiguring the FPGA.

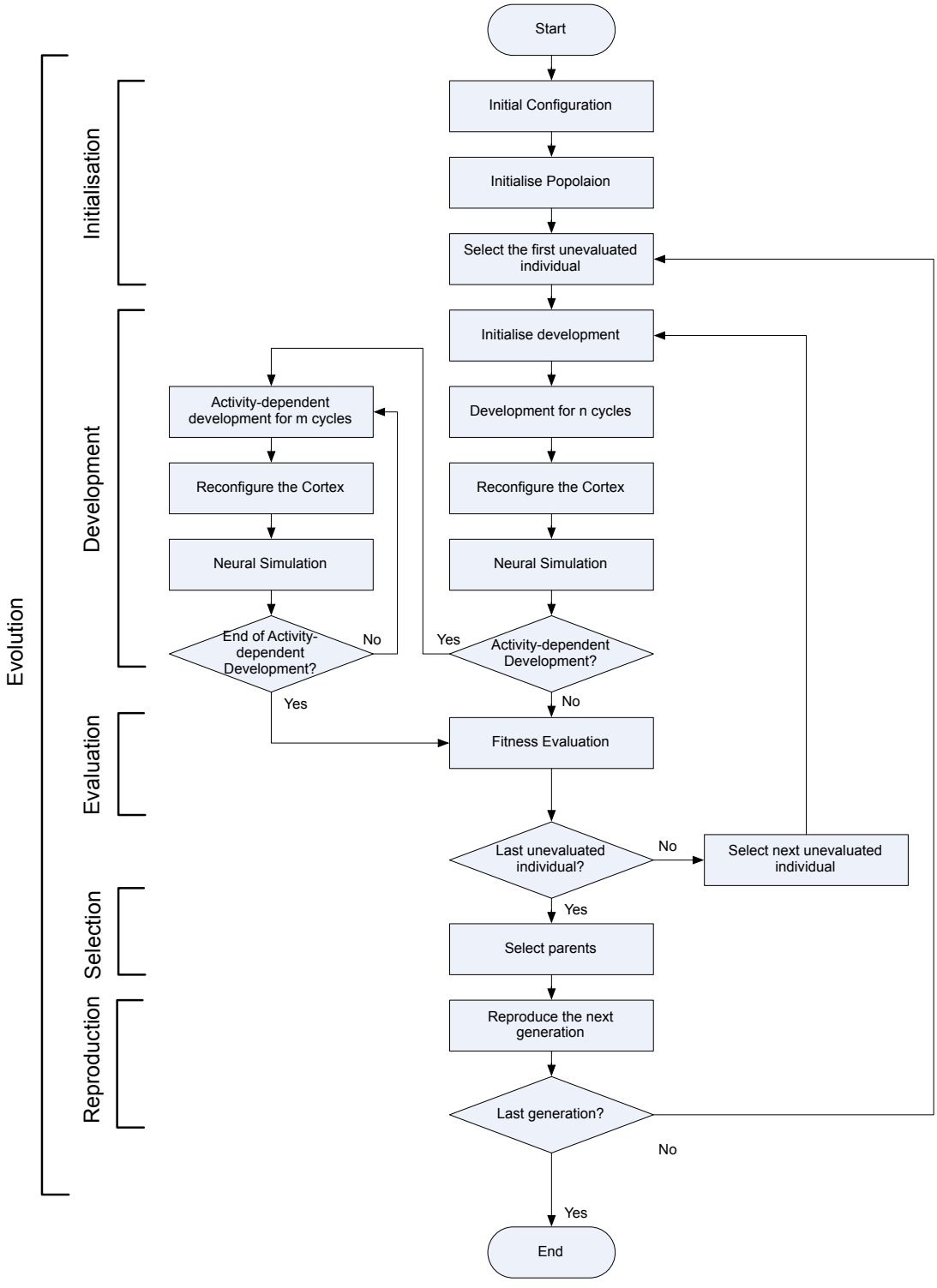


Figure 7.2: Flowchart of the evo-devo neural microcircuits system

Neural Simulation Process

The neural simulation process comprises initialising (for example resetting if needed) the cortex and putting it in neural simulation mode and feeding the stimuli to the cortex and reading back the responses and activity data. The stimuli can be generated dynamically or be stored in a memory (on a PC, the FPGA board, or the FPGA chip itself). The response can be similarly stored in a memory or directly sent to the fitness evaluation process. Depending on the neuron model design that may include an unsupervised learning processes (such as Hebbian) in the synapse model, this simulation process can perform unsupervised learning concurrent with the neural simulation. However, one type of activity data that can be gathered from each synapse and sent to the developmental model during simulation is statistics of the relative timing of the pre and postsynaptic action potentials. This data can be used by an activity-dependent evo-devo model to evolve effective unsupervised learning strategies for adapting the synaptic weights and network architectures.

Activity-dependent Developmental Process

Very similar to the initial development process, the activity-dependent developmental process can also use the activity data from the cortex to control the local concentrations of few proteins during a number of developing cycles. This process can be optionally used in the system where simulation and development processes are concurrently (or alternatively) executed. This is similar to the activity-dependent development in biological organism reported as in phenomena such as ocular dominance plasticity and spontaneous neural activities in mammalian visual cortex [187]. This may be particularly effective in efficient use of the limited resources in the cortex, for instance where only some of the IO cells are carrying information in a specific application.

Fitness Evaluation Process

This process comprises analysing the cortex response to calculate a fitness value that based on the system application shows how well the evaluated microcircuit can tackle the application problem. This process might also use a supervised learning method for transforming the cortex response to a set of outputs required for solving a classification or regression problem. The fitness can be also affected by the simulation time or development time if needed. This process is discussed in detail in section 7.2.2.

Most of the above processes are discussed separately in previous chapters when each one of the system models are investigated. Figure 7.1 shows an abstract data flow diagram of the models in the system and how they need to interact with each other. To integrate the system, apart from designing and implementing these means of interactions, the fitness evaluation module must be also designed and implemented. As it is shown in figure 7.1, interactions between the fitness evaluation module, evolutionary model, and developmental model that is already covered in the previous chapter and are implemented in software is not the focus here. Similarly, the interactions between the cortex and neuron models, which are both implemented in hardware, are already discussed in chapter 5. The following two functions are left to be investigated and different approaches towards design and implementation of each one are explored in detail in the following sections:

1. a. Interactions between two partitions of the developmental model that can be implemented in hardware and software, and

   b. The interactions between the developmental model and the cortex model including sending the reconfiguration data to the cortex and receiving the activity and health information from the cortex.

2. Fitness evaluation module and its interactions with the cortex that include sending stimuli to the cortex and receiving the response from it.

7.2.1 Developmental Model Partitioning

As discussed in section 6.2.4, the evolutionary model should be implemented in software. However, depending on many factors discussed in that chapter, some of the processes in the developmental model might be better implemented in the same FPGA as the cortex model. The developmental model receives a single genome for each individual. This genome will be then translated (mapped) into a dynamical system (GRN) that iteratively changes the state of different cells, producing different cortex configurations. At one or many stages, the cortex model needs to be reconfigured according to these configurations. Two different parts of the developmental model (mapping and GRN) can be both implemented in hardware, both implemented in software, or only the mapping implemented in software. Integration of the developmental model internal modules is discussed here in every one of these situations.

Hardware-based Developmental Model

In this case, both mapping and GRN modules of the developmental model are implemented in hardware. The developmental model residing in the FPGA needs to receive a genome from the evolutionary model, which is implemented in software. A genome is a relatively short string of numbers in its very compact form. Bandwidth between the software and hardware in this case is not a critical factor in the performance of the whole system and therefore it does not require a very high-speed data link between the software running on the PC and the hardware in the FPGA. However, as discussed in the previous chapter, depending on the complexity of the mapping algorithm, implementing the mapping in hardware may significantly add to the hardware cost on the FPGA while it may have marginal or no performance benefits for the whole system.

When both modules of the developmental model are implemented in hardware, the interactions between the developmental and cortex models also take place in hardware. This will be most beneficial when the dynamical system (GRN) module is distributed and replicated for each cortex cell and the reconfiguration and activity data can be locally communicated between developmental and cortical parts of each cell using a virtual FPGA method as discussed in chapters 5 and 6. However, when the GRN is implemented in a single or in a few PEs, they may use a single reconfiguration port using a dynamic partial reconfiguration method or a few separate custom reconfiguration ports using a virtual FPGA method. In either case, the number and bandwidth of the reconfiguration ports can limit the reconfiguration speed. When using the dynamic partial reconfiguration method, employing lookup tables and precompiled pieces of the reconfiguration bitstream stored in a small memory can help to speed up the translation of the cortex reconfiguration data into FPGA reconfiguration data.

Software-based Developmental Model

When all parts of the developmental model are implemented in software, the evolutionary model can simply communicate with the developmental model inside the software partition. The only issue will be the interactions between the developmental model, which resides in the software partition, and the cortex model in the hardware partition. One side of this interaction is sending the reconfiguration data, which also includes translation of the cortex reconfiguration data into FPGA reconfiguration data. Depending on the partitioning and mapping if this translation process, there are a few options:

One option is that the translation is performed in software running on the same PC as the rest of the developmental model is running. In this case the efficient general-purpose processor of the PC can quickly translated the cortex reconfiguration data into an FPGA reconfiguration bitstream and send it directly to one of the FPGA external reconfiguration ports. This limits the reconfiguration speed by the external reconfiguration port bandwidth and the speed of the datalink between the PC and the reconfiguration port. This can significantly impact the performance of the system as reconfiguration bitstreams can be quite large and apart from the FPGA reconfiguration port limitations, a moderate datalink such as a USB (Universal Serial Bus) can result in significant transmission time overheads. This can be aggravated when FPGA configuration memory frames are needed to be read, modified, and written back depending on the FPGA architecture and its reconfiguration bitstream format.

The other option is to implement the translation process in software, but map it to an embedded processor in the FPGA. This processor can have direct access to the internal FPGA reconfiguration ports on the chip that usually offer higher bandwidth than external off-chip reconfiguration ports. Although an embedded processor (either a hard or soft IP core processor) is usually not faster than the high-end PC processors in translating the data, since the cortex reconfiguration data is a much more compact representation of the FPGA configuration than FPGA bitstream, less transmission overhead and latency is expected. However, the actual trade-off between the translation and transmission times may vary depending on different available FPGAs, embedded processors, and data links.

The third option is to translate the reconfiguration data in FPGA hardware. This is probably the fastest method as only compact cortex reconfiguration data is being transferred and the translation is performed by custom hardware in FPGA with direct access to the FPGA fabric and internal reconfiguration ports. However, this requires more hardware resources than former methods due to the complexity of the FPGA configuration bitstream formats. Again, employing lookup tables and precompiled pieces of the reconfiguration bitstream stored in a small memory can speed up the translation processes in FPGA.

The health and activity data from the cortex model can also be transferred from FPGA to the PC running the developmental model using the same datalink that is used for sending reconfiguration data. For both FPGA reconfiguration and activity data, the size of the data that needs to be communicated over the link between the PC and the FPGA is proportional to the area of the cortex and the solution will not scale very well as the datalink bandwidth becomes a bottleneck when cortex size and number of FPGAs are increased.

Partitioned Developmental Model

A partitioned developmental model can allow a balance between scalability and efficiency. For example the genome to GRN mapping can be implemented in software running on a PC while the GRN can be implemented in hardware on FPGA. This can be a much faster approach as discussed in the previous chapter. This, however, requires that only dynamical system (GRN) description to be transferred from software partition to the FPGA. The interactions between the dynamical system and the cortex model can be very efficient and local at the cell level inside the FPGA, particularly if a distributed dynamical system is implemented with local reconfigurations using a virtual FPGA method. Partitioning the developmental model at a point closer to the genome adds to the complexity and hardware cost of the developmental model but allows more processes to be implemented in a distributed and parallel architecture that improves scalability and performance as discussed in previous chapter. On the other hand, partitioning it closer to the cortex model decreases the hardware cost by migrating more processes to software up to the point that even the translation of the cortex reconfiguration data to FPGA reconfiguration data can be done in software, although this could be on a PC or an embedded processor on the FPGA chip.

If the developmental model is partitioned so that the dynamical system is running in FPGA, the size of the data communicated between the PC and the FPGA is fixed as only one copy of the dynamical system (GRN) description needs to be send to the FPGA and used for all the cells and no activity or health data is needed to be feedback to the PC since dynamical system use that data on FPGA itself. This allows much more scalable solutions compared to a software-based developmental model or partitioning it at the boundary of the dynamical system and reconfiguration data translation. Figure 7.3 depicts the trade-offs in the partitioning of the developmental model and how different factors grow with the selection of the partitioning boundary.

7.2.2 Fitness Evaluation Module

This module is responsible for generation and feeding the stimuli to the cortex model during simulation of the neural processes and also receiving and analysing the responses of the cortex to the stimuli. The fitness evaluation module produces a fitness value (or a set of values in case that the evolutionary model supports constraints or multiple objectives). If activity-dependent development is used, this module must also generate the stimuli during the simulation sessions needed for the activity-dependent development iterations. The cortex response during these simulation sessions may or may not be used towards calculation of the fitness value. Fitness value calculation and its processes are highly dependent on the specific application of the system. Here the focus is on those parts of the fitness evaluation module that may be implemented in hardware. Therefore it is best to tackle the partitioning problem of the module first.

Hardware-software Partitioning

Two parts of the fitness evaluation that need to keep up with the neural simulation process are: sending stimuli to the cortex, and receiving the responses from it. Although generation of input spikes and detection of the cortex output spikes are normally carried out at a lower resolution than low-level neural processes (internal soma and synapse computations), large number of input and output signals can significantly increase the required bandwidth. For example, a cortex that simulates equivalent of each

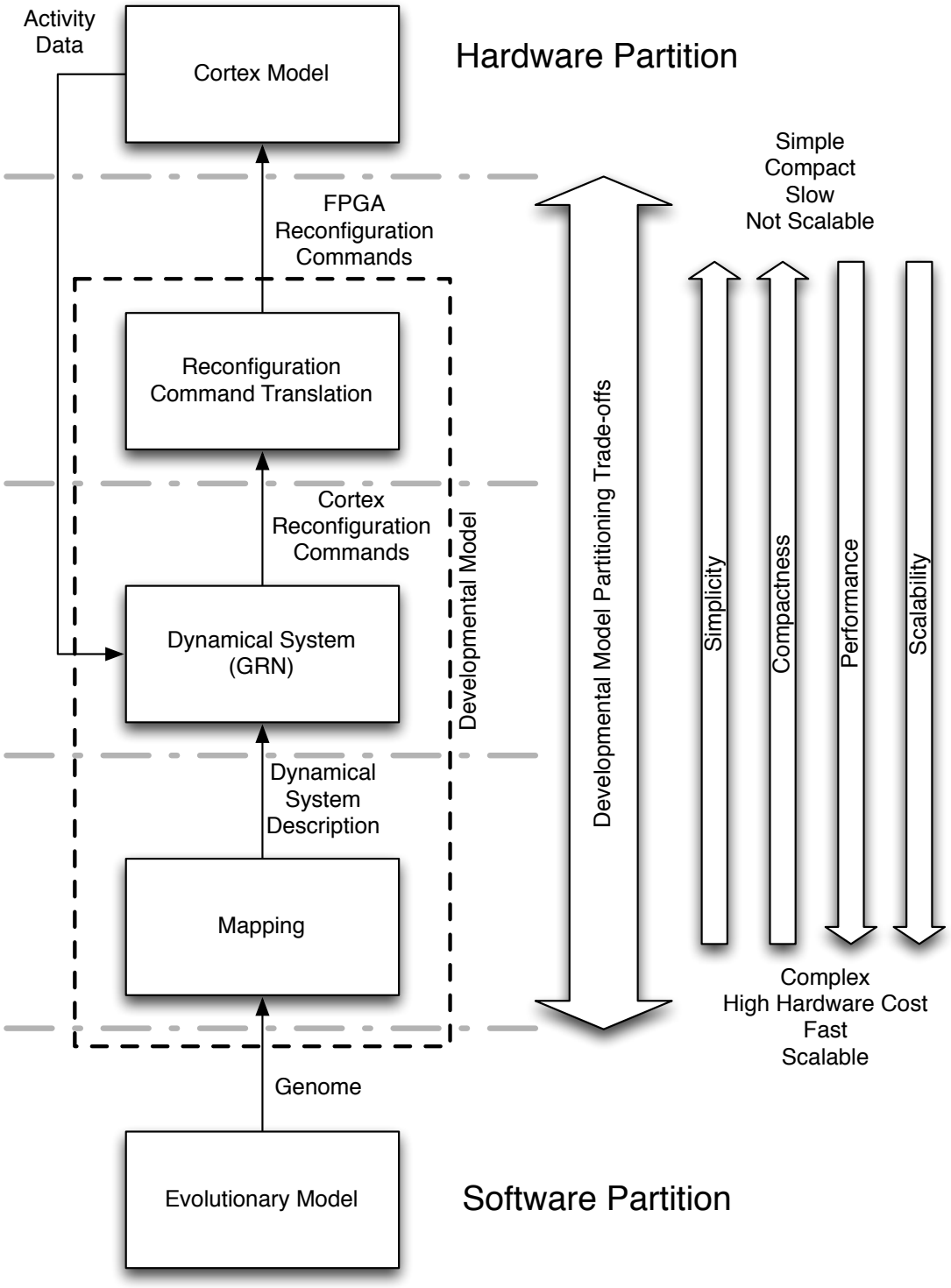


Figure 7.3: Developmental model partitioning trade-offs depicting the growth of different factors based on the partitioning boundary between hardware and software components of the system. The stroke dash lines represent some of the different options for the partitioning of the developmental model between hardware and software components.

millisecond of biological neuron activity in one microsecond (1000 times real-time speed), with 100 input and 100 output signals with resolution of one sample per millisecond of activity, needs a bandwidth of at least 100 mega bits per second in each direction. This bandwidth is proportional to the number of signals and any bottleneck in these communications can impact the performance of the system. Therefore, custom hardware encoder and decoder units (similar to IO cells in case study of chapter 5) that can translate sparse spike trains into more compact representations (*e.g.* real or integer numbers) must be used that scale with the number of cortex input-output signals. Such encoder/decoder units can significantly reduce the require bandwidth and allow a general-purpose processor to generate and analyse them. The stimuli can be generated dynamically at simulation time. It is also possible to store static stimuli (such as input vectors from a dataset) in the memory and feed them to the cortex. This can be done using hardware-based methods, such as DMA (Direct Memory Access), or using software running on a processor on a PC, on the FPGA board, or on the FPGA chip itself. The cortex response can be also analysed dynamically or simply stored in a memory and analysed at a later time or in a pipeline of processes. In either way the bandwidth needed for communicating the data between the hardware partition and the software that analyses the data is a major factor that can impact the scalability, performance, and hardware cost of the system. However, high-speed data transmission between digital devices is a well-studied subject. Moreover, since FPGAs are heavily used in the telecommunication industry, they usually feature the state-of-the-art high-speed interfaces as on-chip hard IP cores. In either way using those interfaces add to the hardware cost. It is therefore preferable to analyse the cortex responses with minimum communication between devices when possible. This depends on the complexity of the fitness function that is highly dependent on the specific application. In the following, a few possible scenarios and options are investigated focusing on the partitioning of this module.

The evo-devo neural microcircuit system can be applied to intelligent control, classification, or regression problems, using a supervised, semi-supervised, or reinforcement learning method. In a supervised learning approach, the evo-devo system can be responsible for evolving the initial weights but then a traditional supervised RNN method be used for adjusting the weights during the simulation. This requires a training dataset for learning and a testing dataset for evaluating the error rate. The testing error can be used as a measure of fitness.

Another option is to take a Reservoir Computing approach. In this method, the evo-devo system evolves the microcircuit and all its internal synaptic weights but a readout map (see section 2.4.4) is then trained using a training set to obtain the desired outputs for a classification or regression problem. A linear readout map of the form:

$$y = X \cdot w \tag{7.1}$$

(for regression or classification, based on the application) can be trained for every system output using the training set to minimise the training error ($||y - X \cdot w||_2$). Here, X is a matrix constructed from the cortex output vectors for different input vectors in the training set, y is the vector of labels (or desired readout map output, in case of regression) and w is the vector of readout map weights. This is a linear least-squares problem that can be solved by obtaining the pseudo-inverse of the matrix (X^+) and

calculating the right side of the equation:

$$w = X^{+}y. \tag{7.2}$$

This process can be deployed on a PC. However, as this solution must be repeated for all readout map outputs and involves similar operations on potentially very large matrices, a GPU can also be used to speed up the process. The testing dataset can be fed into the cortex and its output vectors processed using this trained readout map. The output of the readout map can be then compared to the desired output vectors of the testing dataset using the same least square error measure to calculate total testing error. Then the fitness of the individual can be evaluated base of this testing error.

Both aforementioned methods require simulation of the microcircuit on both training and a testing datasets. An alternative approach is to measure the separability of the cortex outputs on a single testing set without any need for training the readout map. Since readout maps in RC approach are linear, a linear separability measure can be used as the fitness measure. In a semi-supervised learning setting, optional unsupervised learning processes (such as Hebbian) in the neural model (see chapter 4) can be used to adjust some or all of the synaptic weights when the neural microcircuit is simulated on an unlabelled training dataset and then evaluate its separability measure on a labelled testing dataset. In reinforcement learning setting, there is no label for each input vector but responses of the cortex on a specify task can be evaluated by award or punishment signals that can contribute to the fitness of the individual. This setting may require the system to be embodied in the problem environment or the environment to be simulated in sync with the cortex, which can be a problem given the high speed of the cortex. However, the later method appears to be the most bio-plausible option as its structure is highly related to the interactions of the individual organisms with their environment in biological systems. Again optional unsupervised learning processes inside the neural model can be similarly used in the reinforcement learning approach. In all the above methods and approaches, the spike encoding and decoding modules are better to be implemented in the hardware and the matrix operations are better to be implemented in software deployed to a PC with a suitable GPU.

7.2.3 Bio-plausible Methods to Reduce Evaluation Time

A bio-plausible method to reduce the evaluation time, is to have a variable number of development cycles that can be adapted by the evo-devo system. The number of the development cycles or the actual development processing time can contribute to the fitness of the individuals as a negative factor. This allows the system to quickly evolve simple individuals in the beginning of the evolutionary run and then increase the number of development cycles as the system needs to complexify the phenotypes to achieve higher total fitness values. The development time can be encoded directly in the genome or be controlled by developmental processes using a specific protein concentration or implicitly using energy methods as in [80]. This method, apart from its performance benefits and being highly bio-plausible, is shown to be able to improve the robustness and scalability of the individuals and their capacity to regenerate and self-repair [80].

Similarly, the simulation time and size of the datasets used to evaluate individuals can be variable

and increased using a progressive fitness function as in [337]. This method, apart from its bio-plausibility and performance benefits, is also shown to be effective in improving the evolvability of evolutionary methods for inference of Finite State Machines [337, 161] and neuro-controllers [104]. This method can be particularly effective if it is possible to sort the testing dataset entries (or challenges in a reinforcement learning approach) based on their difficulty. Most of these approaches and options, can have significant impacts on the performance, and bio-plausibility of the system. However, they mostly belong to the domain of evolutionary model and are not bounded by the limits of the hardware implementation in FPGAs that is the focus of this thesis.

7.3 Case Study: System Integration

Given the results of the general investigation of the design factors, options, and approaches in the previous sections, here, a simple example of integrating the neuron, cortex, and evo-devo models (from previous chapters) is presented as a case study for further investigation of the practical challenges, trade-offs, and constraints of the integration of such bio-plausible systems.

7.3.1 Application Problem

A very simple problem is selected for this case study to reduce the time needed for design, verification, and testing of the integrated system. The application chosen for this study is the temporal version of the XOR problem previously used in studies on spiking neural networks [41]. The XOR problem is a linearly-non-separable problem that requires hidden units to produce the correct outputs. Three inputs to the Cortex are used in this problem. One input is used as the reference spike for measuring the timing of the other spikes. It always fires once at time $t_r = 0$. The firing time of the two other inputs (t_a and t_b) can vary between 0 to 6ms. Only one of the Cortex outputs are used. The first firing time of this output t_o is desired to be between 10 to 16ms governed by the following equation [41]:

$$t_o(t_a, t_b) = 10 + \frac{6}{1 + e^{-2(|t_a - t_b| - 3)}} \tag{7.3}$$

Figure 7.4 shows a 3D plot of this function. This is an interpolated version of the XOR function sampled at 1ms intervals of t_a and t_b. This interpolated XOR function is more complicated to evolve than the binary version that only allows two different timings (*e.g.* $t = 0$ and $t = 6$) for each input spike. Bohte in [41] trained a network of six neurons to generate this function.

7.3.2 System Overview

Figure 7.5 depicts the block diagram of the whole integrated system of this case study. The whole system consists of two major hardware components: A PC and a Xilinx ML505 FPGA prototyping board. Selection of this prototyping board and FPGA is already covered in chapter 3. These two main hardware components are connected using a USB to JTAG adapter (Xilinx USB Platform Cable) that allows the PC to configure the FPGA. The PC is also connected to the FPGA using a serial link that allows PC to communicate with an embedded processor (MicroBlaze™) on the FPGA chip for consecutive reconfigurations and response data transfer. Using a PC for the reconfiguration of the FPGA and running the

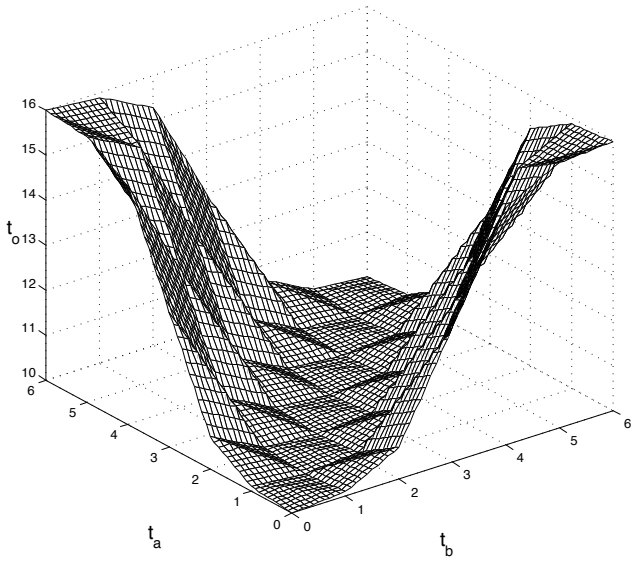


Figure 7.4: A 3D plot of the desired output timing (t_o) as an interpolated XOR function of two input spike timings (t_a and t_b).

evo-devo model processes is already discussed in chapters 5 and 6. In the following, different processes of the system are explained step by step.

Overview of an Evolutionary Run

At the beginning of the system start-up, the FPGA chip needs to be configured using the initial configuration bitstream stored in the PC storage. During this process, the FPGA is programmed to create the hardware of the Cortex, an embedded soft processor (MicroBlaze ™), and the needed IO circuitry (*e.g.* UART- Universal Asynchronous Receiver/Transmitter) in the FPGA. After this stage, the FPGA starts to work. The embedded system starts running its own program, which after performing self tests for its peripheral devices, reports back to the PC using the serial link, signalling that it is ready for the Cortex reconfiguration data.

After initialisation of the FPGA, the evolutionary algorithm starts. The details of the evolutionary and developmental models are reported in the case study of chapter 6. The evolutionary process, which was entirely implemented in software running on the PC, starts with an initial population of random chromosomes. To evaluate the fitness of each individual solution, first it is developed for a fixed number of development cycles with no activity-dependent development for simplicity. The resulting developed neural microcircuit is then translated into a set of Cortex reconfiguration commands, ending with a simulation command, sent to the embedded processor on the FPGA through the serial link. These commands determine the routing information, synaptic weights for glial cells, and parameters for soma cells in a compact format.

The embedded processor receives the Cortex reconfiguration commands, translates them into FPGA reconfiguration commands, resets the cortex, and reconfigures the Cortex through ICAP using the dynamic partial reconfiguration method. Then the embedded processor enables the Cortex clock signal

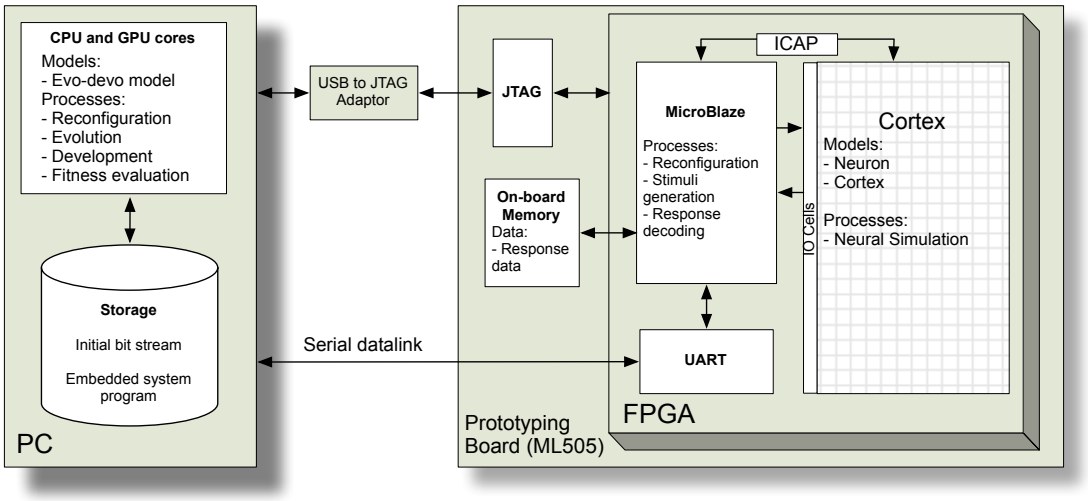


Figure 7.5: A block diagram of the integrated system of the case study showing the connectivity of the main components (PC and FPGA) and mapping of the modules and processes to these components.

and starts to generate input spikes (stimuli) as the neural simulation process is running on the Cortex. The embedded processor generates all the possible combinations of the t_a and t_b timings from 0 to 6ms with 1ms steps (49 different combinations). The time-reference input and two inputs for XOR function are fed into the Cortex through more than three inputs giving a wider range of neurons in the Cortex access to these input signals. These three input signals were fed into IO cell numbers (2,3,4), (6,7,8), (10,11,12), (14,15,16), (18,19,20), (22,23,24), (26,27,28), (30,31,32). The output was received from the IO cell number 4. To help the evo-devo model to locate these input and output cells, a maternal factor protein was added to the developmental model and its concentration was set to 1.0 in IO cell number 4 at the beginning of the development. This allows the developmental process to generate non-symmetric positional morphogens in the vertical direction of the Cortex. All IO cells have their own maternal factors as explained in chapter 6. The embedded system feeds the input spikes to the IO cells and reads the spike counter register of the 4th output every 40 clock cycles, which is assumed to be equal to 1ms of a biological neurons activity. The embedded processor records the timing of the first spike in the output (with 1ms resolution) during 255×40 clock cycles for every combination of input spike timings (49 combinations). These 49 timing values are sent back to the PC through the serial link. If a microcircuit does not generate any spikes in the output for a timing combination, the maximum output timing (255) is assumed. The rest of the fitness evaluation module, implemented in the software running on the PC, receives these timings from the Cortex (t_c) and compares them with the desired timings (t_o) of equation 7.3 using a root mean squared error of the form:

$$Error = \sqrt{\frac{\sum\limits_{t_a,t_b \in \{0..6\}} \left(t_c - t_o(t_a,t_b)\right)^2}{49}} \tag{7.4}$$

The individual's fitness is then calculated as:

$$Fitness = 255 - Error \tag{7.5}$$

This fitness value is then used by the evolutionary algorithm for that individual. This evaluation process is repeated for fitness evaluation of each individual in the evolutionary algorithm. The same evolutionary algorithm of previous chapter is used here.

7.3.3 Detailed Design

Detailed design steps of different parts and processes of the system are reported here. First the details of the initial reconfiguration of the FPGA and the serial link between the PC and FPGA are discussed. The minimum required changes in the developmental model to tackle the case study application problem and its partitioning is explained next. The Cortex reconfiguration process, neural simulation and fitness evaluation processes are covered at the end before discussing the details of the implementation in the next section.

Initial FPGA Configuration

This initial configuration is simply carried out by running the standard console application (IMPACT - provided by Xilinx as part of their ISE FPGA Design Suite) on the PC. The communication is through the USB-to-JTAG adaptor (USB Platform Cable - provided by Xilinx). The program for the embedded processor is written to the block RAMs used by the embedded system available in the FPGA fabric. This is done by adding the program binary to the bitstream file using Xilinx tools before reconfiguration. The USB-to-JTAG adapter provided by Xilinx, is capable of programming FPGAs through a JTAG interface with a maximum clock frequency of 24MHz, resulting roughly a band width of 24Mbps. However, the JTAG boundary scan chain (a series of devices connected serially to be programmed using a single JTAG port) in the ML505 board does not support speeds higher than 6MHz. This appears to be due to the existence of a peripheral chip with that limitation in the chain. Although this effectively degrades the reconfiguration speed through PC by a factor of four, since this initial configuration is performed once at the beginning, it is not a critical performance factor. This JTAG link is also used for debugging the software and hardware of the embedded system using Xilinx tools. It is also possible to program the flash memory available on the FPGA prototyping board once and then every time that the FPGA is powered up, it will automatically reconfigures itself. However, as the embedded system program needs to be modified and debugged repeatedly during the study, this option is not used.

Serial Link

Apart from the USB-to-JTAG connection between the PC and the FPGA board, there is also a data link needed for communicating the Cortex reconfiguration and simulation commands and Cortex activity and response data. Although the ML505 sports a number of very high speed interfaces such as PCI Express and 1Gbps Ethernet, for simplicity in the design and saving in implementation and testing time, a UART (Universal Asynchronous Receiver/Transmitter) is used. This IO module is already available as an optional peripheral for the MicroBlaze^{TM}embedded system in the FPGA. It can work at up to 9600000bps. Design and implementation of a high-speed PC-FPGA link is a usual engineering task that is not the subject of this study. As the system is supposed to be robust to a small amount of communication noises and bad timings, a simple hand-shaking and checksum error detection protocol is used for communication

between two devices.

Developmental Process

Each individual is only initially developed for a fixed number of development cycles to obtain active microcircuits and no activity-dependent development is used to keep the system simpler and faster. Moreover, the Cortex and Digital Neuron designed in the previous chapters lack any activity feedback mechanism that could send data to the developmental model. The number of initial development cycles are fixed for simplicity. The developmental model is same as the one explained in chapter 6, implemented in software, running on the PC.

The developmental model of the previous chapter lacks a mechanism for developing the synaptic weights. Fixed values are used in that model. This feature is included in the developmental model by adding a pair of synapse weight factor protein types (pre and post synaptic weight factors) to the developmental model. The synaptic weight W_{jkl} in each glial cell l between axon j and dendrite k is calculated using the equation:

$$W_{jkl} = \tanh\left(100 \frac{\sum_{i=1}^{L} V_i^{aj} \cdot V_i^{dk} \cdot V_i^{ml}}{L}\right) \tag{7.6}$$

where V_i^{aj}, V_i^{dk}, and V_i^{ml} are representing the ith value in the pre and post synaptic weight compound proteins (of the axon and dendrite mother cells) and compound protein (in the glial cell) shapes. This is very similar to the way that synapse formation probability is calculated but the coefficient and constants are selected in a way that the weight is in range $[-1, 1]$ instead of the range $[0, 1]$. These synaptic weight values are then multiplied by 32767 to expand them to the full range of the possible synaptic weights in the Cortex. The hyperbolic tangent function allows the weight values to be distributed suitably while also smoothly cropped over the required range.

Also, for simplicity, the soma cell parameters are not evolved and set to fixed values for all the neurons in the cortex. However, a group of protein types and protein interactions similar to the mechanism used for synaptic weights can be added to the developmental model to allow these parameters to be controlled by the evo-devo model.

At the end of the developmental process, the results are translated into simple Cortex reconfiguration commands. These commands are formatted based on Cortex columns so that one single command is followed with the routing data, synaptic weights, and soma parameters for all the cells in a single column of the Cortex. This allows to reduce the data transfer overhead by using large packets of data. Also, instead of translating the Cortex reconfiguration commands to FPGA reconfiguration commands on the PC, it is performed on the embedded system due to the relatively low speed of the serial link.

Reconfiguration Process

The Cortex reconfiguration (and simulation) commands from the PC are received by the embedded MicroBlaze™processor in the FPGA. This process is responsible for disabling the Cortex clock and reseting the Cortex cells (as explained in chapter 5) and putting the FPGA in partial reconfiguration mode, translating the commands into configuration memory frames (using lookup tables) and partial

reconfiguration of the FPGA through the ICAP (Internal Configuration Access Port) available in the FPGA. To speed up this process all the Cortex cells in a column are reconfigured by one command with all the data included in a single packet. This allows the embedded system to reconfigure FPGA frames in an efficient manner (as explained in chapter 5).

Neural Simulation and Fitness Evaluation Process

A simulation command can follow a Cortex reconfiguration command. It initiates the simulation process that consists of enabling the Cortex clock, which effectively starts the neural simulation. Then after a short time the embedded processor starts writing the stimuli spikes to the spike generator registers and reading from spike counter of the output (once every 40 clock cycles). The timing of the program can be adjusted by adding NOP commands to the loop in the embedded system program. However, no NOP instructions were necessary for a timing of 40 clock cycles. For every one of the 49 combinations of the input timings the time of the first output spike is stored in the embedded processor memory. If the microcircuit does not generate any spike in the output, a maximum delay (255ms) is assumed. These 49 timings are then sent to the PC in a single packet in response to the simulation command. The neural simulation process includes realtime stimuli generation, and encoding and decoding of the spikes to/from spike timing that reduces the amount of transferred data between the FPGA and PC as the serial link is fairly slow. With some small changes in the design of the spike counters and spike generators in the IO cells of the Cortex it would be possible to perform the spike time decoding and encoding in the hardware. However, in this case, the embedded processor is able to keep up with the Cortex simulation speed.

The rest of the fitness evaluation process is performed on the PC in software. The output timings of the neural microcircuit, received by the PC are used to calculate the fitness of each individual using equations 7.4 and 7.5.

7.3.4 Implementation

The hardware is mostly the same hardware implemented, tested and used in the previous chapter comprising of the following IP cores from Xilinx:

- MicroBlaze processor core
- Processor Local Bus (PLB) 4.6
- two Local Memory Buses (LMB) 1.0
- Block RAM (BRAM) memory unit
- two LMB BRAM Controllers
- XPS UART (Lite)
- Clock Generator
- MicroBlaze Debug Module (MDM)
- Processor System Reset Module

- FPGA Internal Configuration Access Port

plus the Cortex IP core that was implemented and tested in chapter 5. Using Block RAMs for the data and program storage of the embedded processor allows the MicroBlaze to run at high speed without using a cache. The spike generators and counters of the Cortex IO cells are connected to the processor using the processor local bus that allows maximum speed in writing and reading to/from the Cortex input/outputs. Eight of the cortex input/output spike signals, clock, reset, and eight of the dendritic signals at the left edge are also connected to the ML505 edge connector for debugging purposes.

The majority of the software is already implemented in case studies of the previous chapters for testing and verification of the Digital Neuron, Cortex, and the Evo-devo model. The major addition is the fitness evaluation process that is added as a small routine running on the embedded processor in a loop that receives the reconfiguration and simulation commands and a small function in the PC software that calculates the fitness and assigns it to each individual in the evolutionary algorithm. As explained in the previous sections, some limitations in the simple developmental model of chapter 6 called for small modifications in the proteins such as adding an extra maternal factor for vertical asymmetry in the Cortex, and adding a synaptic weight protein and its interaction with other proteins. The program for the embedded system and for the PC are written in C and C++ respectively. The C++ program is also linked with a Matlab engine that allowed for easy visualisation of the results and statistical analysis during design, verification and experiments. The visualisation functions are able to visual a microcircuit in the Cortex, the synaptic weights of the different glial cells, protein concentrations, the progress of the population fitness during evolution and the output timing of the evolved microcircuit agains its input timings among other model variables. Every time that a better solution is found or every few generations the best neural microcircuit found so far and its evolved XOR function are visualised. To be able to measure the performance of each process in the software, it is implemented in a sequential fashion on the CPU rather than using multi-treaded or GPU implementations.

7.3.5 Verification and Testing

Although most of the system modules were already verified and tested they needed to be tested after integration. First the neural simulation process and fitness evaluation were tested using a single neuron reconfigured with parameters with known behaviours. This single neuron was supposed to deterministically and consistently behave as a regular excitatory neuron in consecutive simulation runs. It was revealed that after the first simulation run the result of the successive runs were depending on the timing of the simulation commands and sensitive to the workload of the PC. This could lead to a large amount of randomness in the fitness evaluations. To avoid this problem after a reconfiguration command that resets the Cortex to its initial state and receiving a simulation command, the embedded system were running through a complete set of simulation runs (49 timing combinations) in a deterministic manner and independent of the PC. The consistent timing responses and fitness values in repetitive reconfiguration and fitness evaluations of the same phenotype was tested and verified at different stages on a single neuron and completely developed and evolved microcircuits.

There were also confusions about the Cortex input/output signals numbering and their memory

mapping in the embedded system causing inconstant timing and fitness results that were resolved and verified with low level signal monitoring and crosschecking the results on the PC and hardware. The expected output timing of the simple phenotypes, which directly connected an input to the output was also verified to insure the correct input and outputs of the Cortex are used in the evaluation process.

The use of PC to FPGA connection for configuration of the Cortex was already tested and verified in chapter 5 through comprehensive test cases. A set of self tests for the embedded system, the UART, the Cortex configuration and fitness evaluation modules were added to the programs on the embedded system and the PC to make sure everything was working as expected at the beginning of each experiment.

It was again noticed that the richness and complexity of the system allows it to appear to work although it may need parameter tuning and corrections. Therefore each model, module and process was separately examined and checked using debugging and visualisation tools available in the Visual Studio IDE and Matlab engine. For example, although in the preliminary tests the system appeared to evolve towards desired microcircuits, further investigation of the synaptic weights showed that the system is barely able to produce any large positive synaptic weights. This was then corrected by changing the constants to their current values in equation 7.6 and visualising the distribution of the synaptic weights given randomly generated compound protein shapes (V_i^{aj}, V_i^{dk}, and V_i^{ml}). The functionality of the integrated system was verified and tested as far as the time constraint of the project allowed before starting the final experiments.

Preliminary experiments showed that, in the beginning of the evolutionary run, the randomness in the developmental process was high. This led to very fit individuals with unfit offspring. This was confirmed by developing the fittest individual of the population a few times and observing highly different fitnesses. This was due to the stochastic nature of the gene expression in the developmental model that can be regulated by evolutionary model when robust pathways are needed. However, a population dominated by such highly fit but unstable individuals can stagnate the evolution as they can not produce any really fit and stable offspring to replace them. The evolutionary algorithm used here (and in previous chapter) has a lifespan for each individual to deal with such a randomness in fitness evaluation (including randomness in development). To deal with this randomness, the lifespan parameter of the evolutionary model was reduced to 5 generations to make the evolutionary algorithm less elitist. The positive effect of this change on the speed of evolution was quickly observed in the preliminary experiments. However, it was noted that the parameter tuning of such a system requires a long and thorough statistical analysis.

Also, it was revealed during the preliminary experiments that the evolution tended to quickly connect one of the inputs to the output and achieve a fairly high (but not perfect) fitness value that was independent of developing any neural microcircuit, effectively decoupling the fitness value from the properties of the developed microcircuit. This caused a very rugged fitness landscape and crippled the evolution. This was overcome by adding a constraint to the fitness function that required causality relations between the input and output spikes and avoided such shortcuts to be evaluated incorrectly. This was accomplished by setting the timing of the output spike to 255ms (maximum) every time the output fired before 10ms.

7.3.6 Experiment

The aim of this experiment is both to demonstrate that the integrated case study system is relevant as it can evolve neural microcircuits on FPGA towards a desirable functionality and to investigate some of the challenges, trade-offs, and limitations in running such a bio-plausible system in applied or experimental settings.

In this experiment the integrated system was set up to evolve an interpolated temporal XOR function as explained in section 7.3.1. All the soma parameters were fixed at a set of parameters for regular spiking neuron (the same that was used in the Cortex selftest routine). These parameters and all the other system parameters used in this experiment are reported in table 7.1.

Table 7.1: Parameters and settings used in the experiment.

Parameter or setting	**Value**	**Unit**
Cortex size	12x120	Grid cells
Number of neurons	120	Neurons
Neuron placements configuration	II (in figure 6.13)	-
Soma tap parameters	-4,-4,-3,-3,1,1,4,4	-
Soma V_{reset} parameter	-16000	-
Soma V_{start} parameter	0	-
Soma V_{bias} parameter	2030	-
Development length	20	Cycles
Protein length (L)	50	-
Initial chromosome length	10	Genes
Maximum chromosome length	50	Genes
Population size	40	Individuals
Selection size (n)	16	Individuals
Generation size (m)	32	Offspring per generation
Life span	5	Generations
Crossover probability	1.0 (always)	Per pair of parents
Creep mutation probability	0.05	Per locus
Gene addition mutation probability	0.5	Per chromosome
Gene deletion mutation probability	0.5	Per chromosome
Gene duplication mutation probability	0.1	Per chromosome
Number of generations	500	-
Number of runs	1	-

An evolutionary run of 500 generations (16008 evaluations) was carried out and best and average fitness of the population, the population average chromosome length, and the chromosome length of the

best individual were recorded in each generation. Moreover the neural microcircuit of the best individual and its output spike timing against the input spike timing were visualised.

Results

Figure 7.6(top) shows the best and average fitness of the population during 500 generations of the evolutionary run. Figure 7.6(bottom) shows how the chromosome length of the best individual and its population average were changing during the evolutionary run. The effect of the randomness in the fitness evaluation causing a highly fit but not evolvable individual in the beginning of the run and its deletion after few generations is evident in the fitness curve. The convergence of the average fitness to the best fitness shows how evolution was able to produce robust pathways for development of the microcircuits that not only are robust to the randomness of the developmental process, but also robust to the mutations in the genome. The population average chromosome length shows a consistent increase during evolution that can be a sign of complexification. It appears that when system reaches the point at which it can not evolve any fitter microcircuits, it starts to reduce the chromosome length. This reduction can also reduce the total number of mutations in a genome.

After 500 generations the input-output spike timing diagram of the the fittest individual (shown in figure 7.7), although not perfect, clearly resembles the target timing (figure 7.4). Figure 7.8 presents a visualisation of the connectivity and synaptic weights of the fittest individual after 500 generations along with a close-up of the top and bottom parts of the Cortex that are involved in the active part of the evolved neural microcircuit. Closer examination of the evolved microcircuit revealed that only six neurons are involved in the generation of the output signal (highlighted with a green rectangle). Slightly different synaptic weight patterns are evident in this area of the Cortex. It shows how, the number of synapses, their weights, and length of the dendrites and axons are used by evolution to achieve the desired effect.

Experiment results demonstrated successfully that the integrated system is able to evolve microcircuits towards the desired behaviour and that the case study system is relevant as an example of bio-plausible evo-devo neural microcircuits on FPGA for investigation of the challenges.

7.4 Practical Considerations

The processing time of different processes in the system were measured during the evolutionary run. Figure 7.9 shows the breakdown of the total evaluation time for each evo-devo neural microcircuit. Each evaluation takes between 600 to 700ms depending on the complexity of the development task using a 2.4GHz i5 Intel PC. About 222ms was spent for the Cortex reconfiguration. Only about 5ms of the evaluation time was spent for neural simulation. This is due to the fact that the application problem to be solved here is very simple and the data set is very small. This application problem can be tackled with few neurons, while the size of the Cortex used is 10 to 30 times larger than what is required. This issue can be addressed in a few different ways. One is to limit the developmental processes to the used cells of the Cortex. This can effectively reduce the development time. However, a similar change in the reconfiguration process requires modifying the Cortex reconfiguration command formats that are based on Cortex columns. A more bio-plausible approach would be to allow the size of the available cortex to

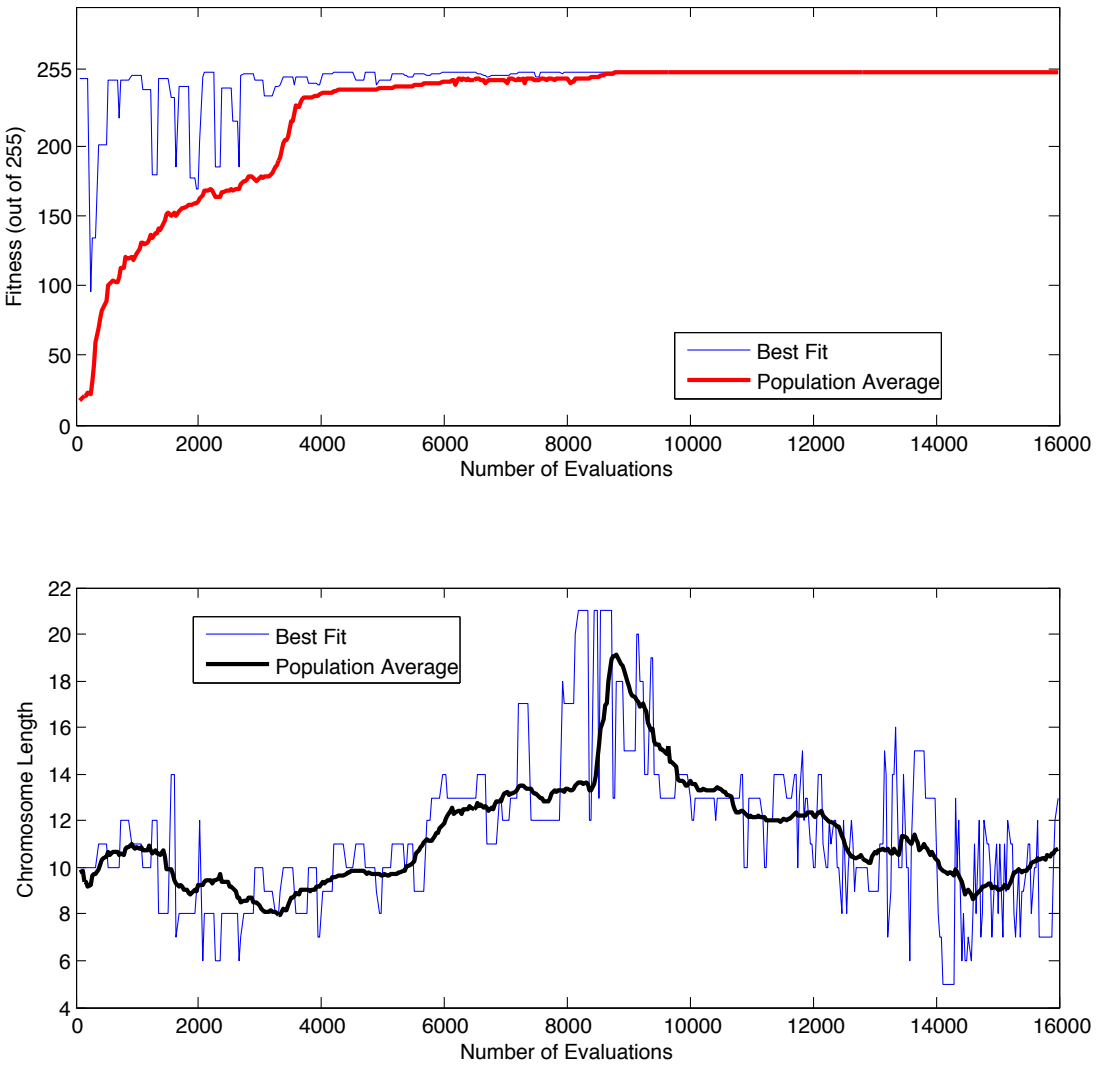


Figure 7.6: Top: Fitness of the best and population average during an evolutionary run against the number of evaluations. Bottom: Chromosome length of the best individual and population average against the number of evaluations during the same evolutionary run.

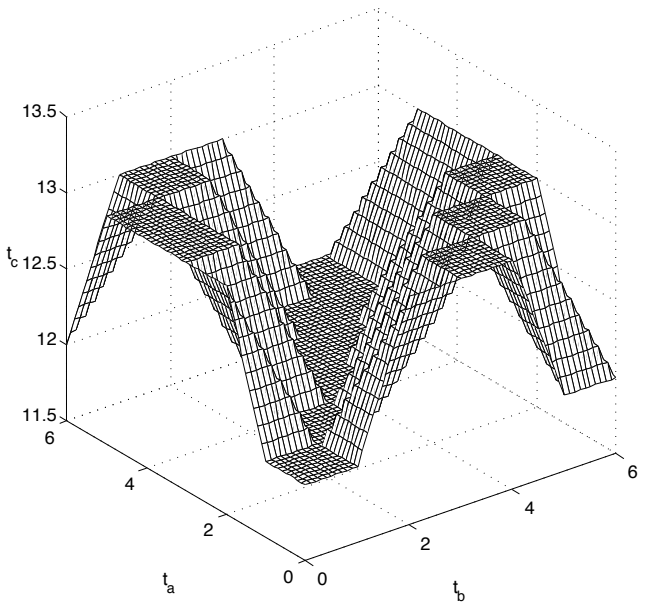


Figure 7.7: The input-output spike timing of the fittest evolved neural microcircuit, which clearly resembles the target timing (figure 7.4).

grow through evolution, controlled by a protein concentration or a separate protein interaction designed for this purpose. However, there must be an evolutionary pressure for smaller Cortex sizes.

The Reconfiguration of one individual and development of the next individual can be processed in parallel on two different PC cores. In this case study design, such a multithread programming can reduce the evaluation time significantly. The reconfiguration time can be also reduced by using a faster datalink between the PC and FPGA, a faster embedded processor, and a faster HWICAP core. About half of the reconfiguration time is spent for communication between the PC and FPGA and the rest is spent for translating the reconfiguration commands and communications with the FPGA ICAP port. Using a faster HWICAP IP core is already discussed in chapter 5. Here, possible quick changes in the system integration that can lead to faster reconfiguration are discussed briefly.

With spending more time it may be possible to use the same 6Mbps USB-to-JTAG interface to emulate a serial link for the embedded processor using the debugging tools provided by Xilinx that might improve the communication speed. The nominal speed of the USB-to-JTAG (6Mbps) could be also improved to a nominal 24Mbps by bypassing the limiting device using a short wire soldered between two pins on the board. These two modifications can effectively improve the Cortex configuration time by a factor of 20 or more. After these changes, experiments may reveal that moving the whole reconfiguration process to the PC (using the JTAG interface at 24Mbps) is faster than the current design since the embedded processor is much slower than the PC processor in translating the Cortex reconfiguration commands. This decision obviously depends on the speed of the PC, embedded processor, datalink between the PC and FPGA, and the ICAP reconfiguration port of the FPGA.

Although the MicroBlaze^{TM}embedded processor could run at a maximum speed of 250MHz it was sharing the same clock signal of the Cortex at a frequency of 104.5MHz. This frequency was selected to

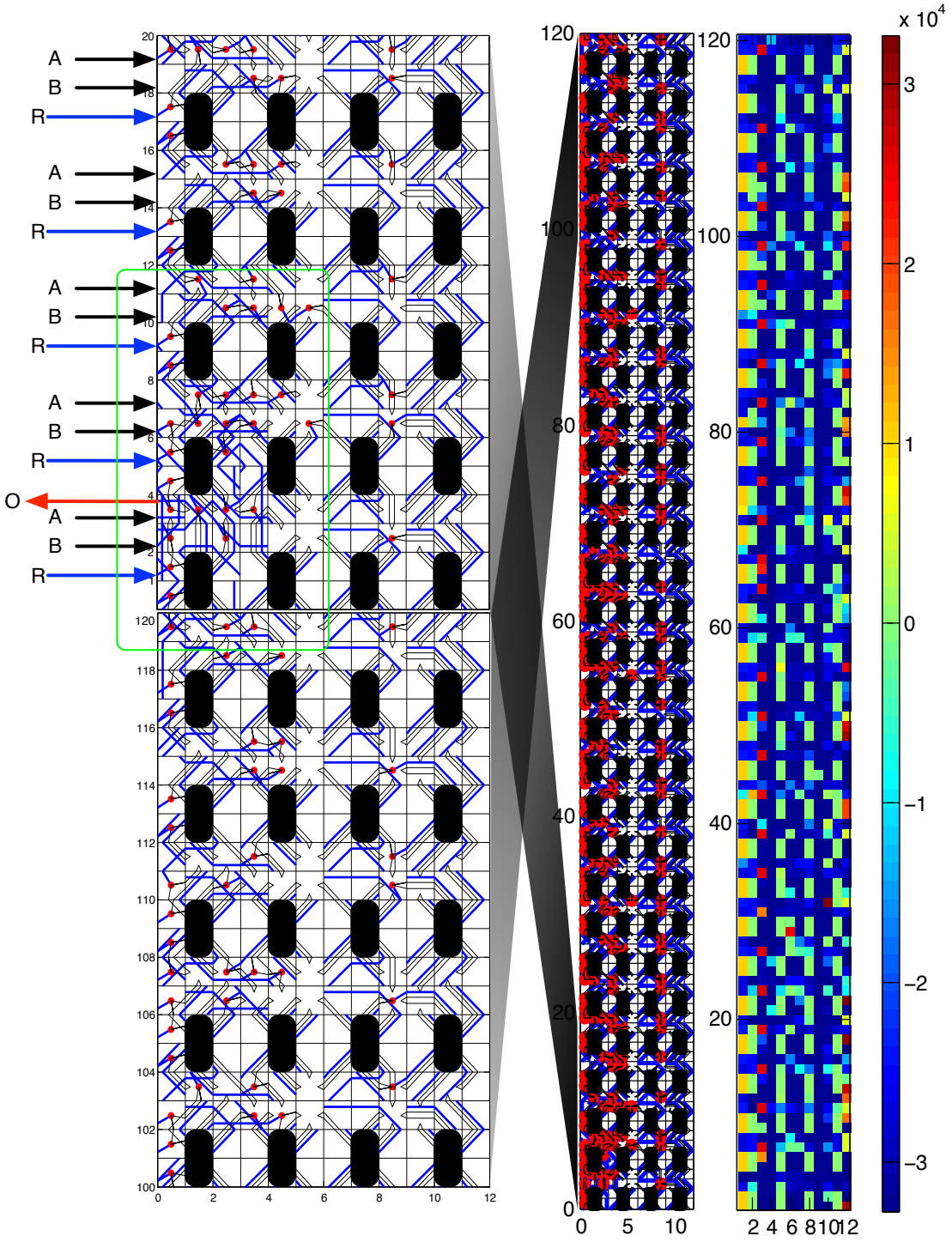


Figure 7.8: The best evolved neural microcircuit after 500 generations. The whole Cortex is shown in the middle, the synaptic weights on the right, and the details of the top and bottom part of the cortex are shown on the left. A, B, R, and O, represent the three input signals with timings t_a, t_b, and $t_r = 0$, and the output signal of the microcircuit respectively. The evolved microcircuit uses the wraparound signals that connect top and bottom of the Cortex. The top and bottom part of the Cortex are magnified and put back together to show the active parts of the microcircuit (highlighted with the green rectangle).

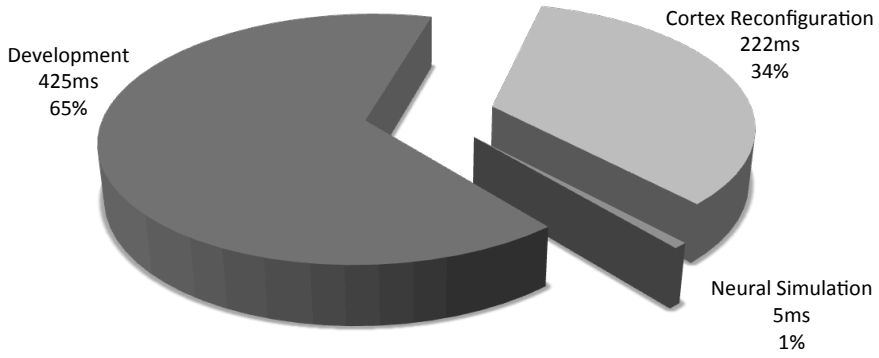


Figure 7.9: The breakdown of the average 652ms total evaluation time of the evo-devo neural microcircuits with 20 development cycles.

reduce the communication error rate of the UART. However, it would be possible to use separate clock signals for the Cortex, embedded processor, and the UART or any other communication device that is used to communicate with the PC. This allows to focus on the bottlenecks in the system and speed them up even if the other parts are limited by their own constraints.

It also appeared that a large number of generations in an evolutionary run are spent only for evolving robust, stable, and useful pathways and modules in the GRN that can support the development of the required microcircuits. There are a few ways to mitigate this problem. One is to use a seed population that already contains some useful GRN modules and pathways that can be reused in evolving microcircuits. Another way is to reduce the number of development cycles. As a shorter development time exposes development to less randomness, generally it is expected to result in a more stable development than longer development times. However, this reduces the ability of the system to develop complex phenotypes that need longer development times. A bio-plausible solution to tackle this problem is to allow the number of development cycles to be evolved and to add an evolutionary pressure towards shorter development cycles. This can be achieved by direct or indirect coding of the number of development cycles in the genome or by implicitly controlling it through the developmental process itself as in [80]. An evolutionary pressures towards shorter development and smaller cortex sizes is bio-plausible as both of them require time, energy, and materials in a physical environment. One simple and bio-plausible way to implement this pressure is to use the development processing time of each individual (on the realtime clock) as a measure to delay the introduction of the individual to the population. However, such methods are part of the evolutionary model running in the software and are not the focus of this study. Another parameter of the evolutionary system that can be left to evolution to tune is the lifespan, which was fixed to five generations in the above experiment.

To improve the ability of the system to evolve perfect solutions, one may consider removing the limitations imposed on the system by fixing the soma parameters and keeping the number of Cortex clock cycles for equivalent of 1ms neural activity at a bare minimum of 40. As explained in chapter 4 and 5, the neuron model needs 18 clock cycles plus the number of pipeline flip-flops in its dendritic loop to complete one update cycle. With the extra 10-bit shift register and six internal pipeline flip-flops of

the soma cell added in chapter 5 to insure the minimum dendritic loop length, this value increases to 38 clock cycles with no dendritic growth. Every dendrite growing one cell away from the soma adds one clock cycle to this value. Therefore, it appears that a larger number of about 70 cycles and ability to evolve soma parameters can give the required accuracy to the neurons to generate spikes at the right time and evolving perfect neural microcircuits. With a clock frequency of 104.5MHz and 70 clock cycles for simulation of 1ms of biological neuron activity, this system will be still able to achieve a speed 1500 time faster than realtime simulation of biological neurons with 1ms resolution.

One of the practical challenges in the design and testing of bio-plausible evo-devo models is the effect of the richness of the model on its response to bugs and errors. Such models are so rich and robust that making small mistakes in the design or coding does not totally cripple the system. Although the model might not perform as well as it can, it still works and it is very difficult to do end-to-end tests on them. Moreover, tuning the parameters of such systems requires long and tedious evolutionary runs and carefully designed statistical analysis. The modular design of the system allowed to test and tune some of the parameter and constants at the lower levels (*e.g.* synaptic weight behavioural protein interaction constants) but tuning higher level parameters (such as evolutionary model parameters) remained to be tuned through statistical analysis.

7.5 Summary

Figure 7.10 shows a graphical representation of the investigations carried out in this chapter. First in this chapter the general impacts of the integration of the different system models and modules on the bio-plausibility and feasibility of the whole system were highlighted. Then the general design factors and constraints that can affect the bio-plausibility, performance, hardware cost, scalability, reliability, and complexity of the system were discussed in the context of the system integration. Different modules, processes, and functions of the system and their interactions were reviewed. Partitioning of the fitness evaluation and developmental processes between hardware and software, and the mapping of the partitions to the PC and FPGA were discussed and different approaches and their challenges, trade-offs, and limitations were discussed.

As a case study, all the models designed, implemented and tested in the previous chapters, along with a simple fitness evaluation module were integrated into a complete system to solve an interpolated temporal XOR problem with three inputs and one output. Design, implementation and testing of the case study system and their challenges were reported. A final experiment was successfully carried out to validate its relevance and to further investigate the practical challenges in the integration and applying the system to an experimental problem.

Effects of the PC, embedded processor, datalink, and reconfiguration port speeds on the reconfiguration and evaluation speeds, and performance of the whole system were discussed. Limitations of the case study design and implementation were investigated and highlighted. Also some bio-plausible options to resolve the limitations and to improve the evaluation speed and total performance of the system were suggested. It was noted that the complexity of such bio-plausible systems complicates the system verification and makes parameter tuning very time consuming. The modular design of the case study sys-

tem reduced the testing complexity and allowed to find some useful parameter settings quickly in case of lower level processes. It was also suggested in the final practical considerations that bio-plausible approaches may help to reduce the need for tuning some of the evo-devo model parameters such as individual lifespan, development time, simulation time, or the cortex size.

The successful experiment on the integrated system showed its ability of evolving microcircuits towards the desired behaviour and validated its relevance for practical investigation of the challenges as an example of bio-plausible evo-devo neural microcircuits on FPGAs.

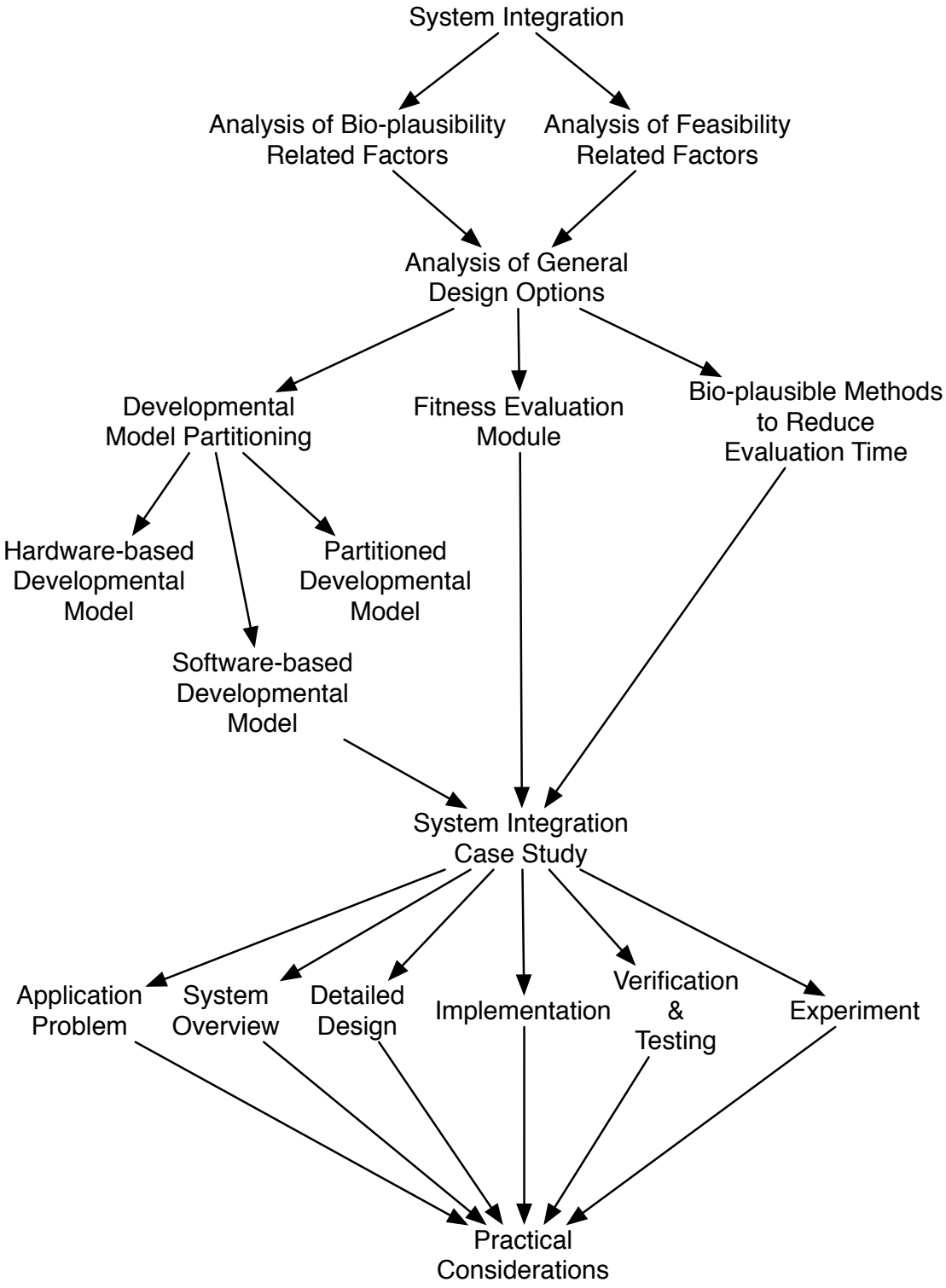


Figure 7.10: A graph of the investigations carried out in chapter 7 regarding the system integration.

Chapter 8

Conclusions

In this final chapter, the study is summarised, evaluated and concluded. First, the thesis objectives are revisited and the research outcomes towards the objectives are summarised. Then based on the outcomes, a set of suggestions, recommendations, and implications for researchers, designer and FPGA manufacturers is presented. The methods of the research are criticised and evaluated and future work for further investigations and developments are proposed. The chapter is concluded with a list of the contributions of this thesis.

8.1 Objectives Revisited

Chapter 1 starts with the importance of bio-inspired engineering in the technology and the role that technological constraints and trade-offs play in it. The potentials and challenges of bio-plausible approaches to intelligent systems and artificial brains in silicon are highlighted and that how an investigation of the challenges, trade-offs and constraints of achieving bio-plausibility in evo-devo neural microcircuits in FPGAs can pave the way and provide insight for the future designers of such systems. The research problem, aim, scope, and objectives of the research are also clarified. Chapter 1 ends with a list of publications based on this study.

At the beginning of chapter 2, bio-plausibility and feasibility, as two main themes of the study, were defined in the context of this research using the existing definitions from the artificial life and robotics literature, and the field of digital electronics **(objective 1)**. Bio-plausibility was defined and measured qualitatively throughout this study by structural accuracy of the models and their consistency with the current biological models. Feasibility was defined and measured throughout the study based on different feasibility measures of cost, performance, scalability, design and testing time and complexity, availability, and reliability. These measures of feasibility and bio-plausibility were the basis for comparison of competing models in different contexts during the study.

The state of the art in the FPGA-related technologies and methods, and bio-plausible neural networks, evolutionary, and neurodevelopmental models in the literature were reviewed and assessed, later in chapter 2 **(objective 2)**. Bio-plausible approaches to spiking neural networks such as LSM and HTM were found to be on the rise in popularity and success. However, lack of systematic approach in their design and adaptation called for new methods such as evolutionary approaches that had already shown

some promising results. Very bio-plausible, and at the same time computationally expensive neuron models were available. Computationally cheaper but bio-plausible models such as Izhikevich model were recently introduced but most of the FPGA implementations were based on simplistic models. Very few bio-plausible neuron and STDP unsupervised learning models existed for FPGA implementation. Some bio-plausible neurodevelopmental systems were found to be used for evolution of neural networks but they were usually very simple and not modelling the details of the gene-protein and protein-protein interactions. Detailed and complex systems were slow and computationally expensive. It was not quite clear which details and complexities of the bio-plausible models should be incorporated in a bio-inspired design in order to achieve desired emergent properties such as evolvability, scalability, fault-tolerance, and robustness. Only very simplistic developmental models were found to be implemented in hardware. Although general trends pointed to promising results from direct implementation of bio-plausible models in hardware, very few attempts, only with custom chips, were found, which appeared to be limited by their cost, scale, and availability. Chapter 2 was concluded with the motivation of this thesis to investigate the challenges of realising these bio-plausible models in ever-increasingly ubiquitous, large and powerful FPGAs.

Chapter 3 examines the challenges, constraints and trade-offs in the selection of an FPGA hardware platform suitable for a bio-plausible evo-devo neural microcircuit system **(objective 3)**. Based on available literature and experience, fourteen different features and factors of the FPGA platform were highlighted and analysed: cost, popularity and prevalence, performance, size and scalability, power consumption, partial reconfiguration, reconfiguration speed, communication bandwidth, interfacing, indestructibility and validity checking, embedded processing, observability, reliability, and ease of use. The general trade-offs between the FPGA cost, its popularity, and its technical specifications were highlighted. The foreseeable impacts of each one of the above features and factors on the bio-plausibility and feasibility measures of the whole system were analysed and evaluated to obtain a selection criteria. The FPGA manufacturers, their focus, market share, and products at the time were studied and assessed. The study showed that very few partially and dynamically reconfigurable and large devices were available at the time. Based on the above criteria, market status, and the available budget, the ML505 prototyping board with a Xilinx Virtex-5 FPGA was selected for use in the case studies as a representative of the state-of-the-art technology at the time of the selection.

Chapter 4 focused on the challenges, options, trade-offs, and constraints involved in the design, implementation, testing and, evaluation of a bio-plausible neuron model suitable for an evo-devo system on an FPGA **(objective 4)**. Bio-plausibility and feasibility of the neuron model and factors affecting them were discussed and examined from different aspects in section 4.1. Based on these factors the rest of the chapter was focused on a general flexible neuron model architecture with a connected network of processing elements that provided the infrastructure for bio-plausible simulation of neuron dynamics with the minimum use of routing resources of the FPGA. Stochastic and deterministic, distributed and centralised approaches to the design of the neuron model were investigated and their feasibility and bio-plausibility were assessed through analysis and simulation. Analysis of the design options revealed

performance-reliability and compactness-reliability trade-offs, and accuracy constraints. Different design options and their trade-offs were summarised in section 4.5 and table 4.2. Based on the analysis of the design options and constraints of this project, a novel digital neuron model with a new Piecewise Linear Approximation of Quadratic Integrate and Fire (PLAQIF) soma model were designed as case study for further investigation of challenges. The new neuron model also served as a simple example of possible compact bio-plausible designs and as a basis for case studies in the later chapters. Detailed design, implementation, and testing of the digital neuron model and the practical challenges and limitations were reported.

The challenges, options, trade-offs, and constraints involved in the design, implementation, testing and, evaluation of a bio-plausible reconfigurable structure on FPGA suitable for evo-devo neural microcircuits (called a cortex model) comprised the focus of chapter 5 **(objective 5)**. The factors impacting the bio-plausibility and feasibility of the cortex model were highlighted and different approaches towards inter and intra-cellular communication, reconfiguration, and feedback were examined and compared closely. The analysis revealed a set of trade-offs between bio-plausibility and performance, compactness, and efficiency of the cortex model. Based on the analysis of the design options and the constraints of the project, a new bio-plausible cortex model (called the Cortex Model) was designed, implemented and tested as a case study that can work with the digital neuron model of the chapter 4. The practical challenges, trade-off, limitations, and possible solutions were discussed at the end of chapter 5.

Chapter 6 examined the challenges, options, trade-offs, and constraints of the design, implementation, testing, and evaluation of a bio-plausible evo-devo model for growing neural microcircuits in FPGAs **(objective 6)**. Factors in the design of the evo-devo model that affects the bio-plausibility and feasibility of the system were highlighted in section 6.1 and then different options and design approaches were closely examined and compared. The analysis confirmed the general bio-plausibility-efficiency and and bio-plausibility-simplicity trade-offs, but also revealed a positive correlation of bio-plausibility of the evo-devo model with the scalability and reliability of the system. It also showed that it is more efficient to implement the evolutionary model in software. Based on the analysis and the constraints of the project, a new bio-plausible multicellular evo-devo model (called the LGRN - Logistic GRN) capable of demonstrating fundamental properties was designed, implemented and tested as a case study. The practical challenges, trade-offs, constraints and possible improvements were discussed at the end of the chapter.

The challenges, options, trade-offs, and constraints involved in the integration, end-to-end testing, and evaluation of a bio-plausible evo-devo neural microcircuit system were assessed in chapter 7 **(objective 7)**. Integration design factors affecting the bio-plausibility and feasibility of the system were highlighted in section 7.1. Different approaches towards system integration and partitioning of the developmental model and fitness evaluation module were discussed and evaluated confirming the usual performance-compactness trade-off and also revealing how hardware-software partitioning can impact the feasibility measures. All the case study models of chapters 4 to 6 were successfully completed, integrated, and tested as a final case study demonstrating the relevance of the case studies and challenges in integration and application of such bio-plausible systems.

8.2 Implications and Suggestions

The investigations carried out through this study revealed many different challenges, constraints, trade-offs, and options. Here we summarise these results as some general trends, recommendations for similar studies, and suggestions for FPGA manufacturers.

8.2.1 General Trends

Analysis of the different options in the design of the neuron, cortex and evo-devo models in chapters 4 to 6 confirmed that performance, and compactness are impacted as bio-plausibility (and thus computational complexity) grows. This is in agreement with the general expected trade-off between bio-plausibility and feasibility, predicted in the first chapter. However, bio-plausible models and FPGAs appeared to lend themselves to distributed and massively parallel architectures, which allows designers of bio-plausible systems to utilise the inherent performance-compactness trade-off to maximise the efficiency, reliability, scalability or other desired factors. Figure 8.1 depicts this relation in a conceptual way. Although the increase in bio-plausibility makes the performance-compactness tighter, it gives more flexibility to play with the trade-off. However, this flexibility of the bio-plausible models can be constrained by the limitations of the hardware platform.

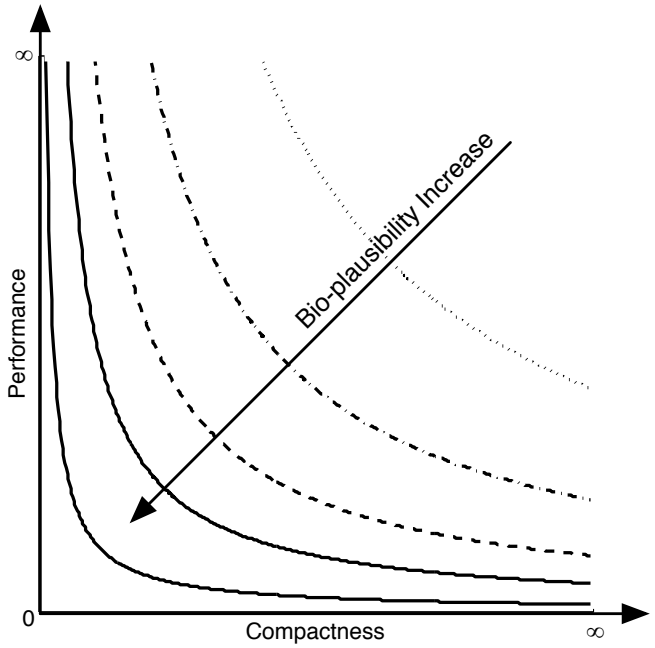


Figure 8.1: A schematic illustration of the relation between bio-plausibility and the performance-compactness trade-off. The continuity of the curves represent the increase in design flexibility to play with the trade-off.

In contrast with the general expected trade-off between bio-plausibility and feasibility, two of the feasibility measures can actually show a different trend in bio-plausible systems. Analysis of different options in the design of the neuron, cortex, and evo-devo models showed that, in general, the scalability and reliability (as fault-tolerance and robustness) of the system grow with bio-plausibility. This can be viewed as one of the main motivations of using bio-plausible approaches in the design of such systems.

The design time and complexity almost always increased with bio-plausibility. A repeating theme in practical challenges at all stages, although more pronounced in the later stages, was the increasing complexity in system verification and testing, and parameter tuning due to the complexity and richness of such bio-plausible system. A modular system design, which allows unit testing and adding or removing different features easily, was the only solution proposed and examined to tackle this problem as it has been proven to be effective in traditional hardware and software design. In the following section more detailed suggestions are offered to tackle this problem in future.

8.2.2 Suggestions and Recommendations for Designers

The scalability and reliability benefits of the bio-plausibility combined with the distributed and parallel nature of bio-plausible systems are promising trends that can allow very large wafer-scale bio-plausible chips. Large silicon chips are not feasible mainly due to low fabrication yields over large areas of silicon. The built-in fault-tolerance, robustness, and adaptability of a bio-plausible evo-devo system can resolve the yield problem. The increased scalability allows leveraging the massive computational power of such a large multi-core chip efficiently.

A modular design approach seems to be the most effective way to mitigate the increasing complexity of the design, testing and parameter tuning in bio-plausible systems. Using random number generators instead of higher level models is also an effective method for testing and tuning the lower level models. This was used, for example, in the testing and parameter tuning of the developmental model before adding the evolutionary model to the system and in testing and improving the neuron model before designing the cortex and evo-devo models. Both hardware and software engineers use a series of tools, standards, procedures, and practices to tackle the increasing complexity of hardware and software systems. It appears very beneficial to follow a similar approach by developing standards, tools, software modules and libraries, hardware IP cores, procedures, metrics, benchmarks and codes of practice for the design and testing of bio-plausible digital systems that allow researchers and designers of bio-plausible digital systems to share, reuse, replace, and compare bio-plausible modules, and cope with the complexity of the design, verification, testing, and tuning of bio-plausible digital systems. Although some libraries, benchmarks, and metrics have been developed by the research communities through recent decades, they are not comparable with the coordinated efforts of the hardware and software engineering communities and the resources available to them. In author's opinion, it is the time to start such coordinated efforts in the evolvable hardware and evolutionary computing research communities.

It appears that the bio-plausibility of lower level-models can significantly improve the bio-plausibility at higher levels of the system. It is important to think of the bio-plausibility of a lower level model not only in itself but also in relation with higher level models. For example, although it is important to have a bio-plausible neuron model that behaves as a biological neuron, it is also important that it behaves in a bio-plausible manner in relation with the cortex and evo-devo models. The possibility of evolving the structure and parameters of a neuron beyond biological norms can result a higher level bio-plausibility than a blind and static simulation of the current biological neurons.

8.2.3 Suggestions for FPGA Manufacturers

Currently, FPGA manufacturers do not appear to be concerned about the fine-grain, local, and distributed reconfigurability of their devices and other features that are crucial for bio-plausible reconfigurable devices. Current FPGA devices are more optimised for telecommunication and traditional parallel signal processing than bio-inspired systems. In the following, few suggestions are put forward for FPGA manufacturers, which can relax some of the constraints and limitations of the FPGAs as a platform for bio-plausible systems and evo-devo neural microcircuits in particular.

- Shortening the length of the reconfiguration frame as the smallest reconfigurable unit in FPGA
- Allowing smaller units of the devices to be separately, locally, and concurrently reconfigured by the FPGA fabric itself as well as through a central global reconfiguration port.
- Uniformity of the fabric and reconfiguration format
- An open reconfiguration frame format including the low-level PIPs
- A Hardwired Network-on-Chip (such as [126] or reconfigurable time-multiplexed switched networks similar to Tabula's space-time technology [3]

8.3 Critical Evaluation

Hardware Platform

This work is focused on the Xilinx Virtex-5 family of FPGAs and dynamic partial reconfiguration method, which does not represent all the possibilities. Xilinx Virtex-5 was the best representative of the available technologies at the beginning of this study and as is explained in chapter 3 it was the largest, fastest and the most feature rich family of FPGAs at that time. Apart from the size, performance and power improvements in the new FPGAs during these years, the main reconfigurability features that are useful for bio-plausible systems remained the same. Although the practical investigations in the case studies are limited to dynamic partial reconfiguration method, the virtual FPGA method was also analysed and evaluated.

Bio-plausibility Measures

This study does not necessarily use the latest and most accurate biological models and knowledge in the fields of neuroscience, neurodevelopment, and genetics as the reference for bio-plausibility of the bio-inspired models. There are definitely more recent findings, technically more accurate biological models than models used here as sources of inspiration or references. However, there is still a large gap between the existing hardware models and the biological knowledge from two decades ago. Given the relative implausibility of the current hardware models in FPGAs, aiming for very high biological accuracy would involve significantly more time and effort, and close collaboration with biologists, limiting the breadth of the study. Therefore, this study uses slightly more accurate and more detailed biological models than is the norm in hardware-based systems to investigate the challenges in improving the bio-plausibility of the hardware-based models.

Feasibility Measures

In this study, the power consumption, one of the important feasibility factors that also impacts the scalability of the system, is not investigated. Energy efficiency and consumption are two important limiting factors in scaling up brain simulators or using them in mobile systems. Heat dissipation is another related issue that limits the miniaturisation of electronic devices. None of these important constraints are included in the list of feasibility measures in this thesis. The peak power consumption, energy efficiency, and miniaturisation of an evo-devo neural microcircuit in FPGA are more than anything related to the technology of the FPGA, and still far from the acceptable range for mobile or very large-scale brain simulators such as SpiNNaker. Bio-plausible evo-devo neural systems in FPGAs are still in their infancy and it is too early to think of their large-scale or mobile applications. The focus here is the feasibility of such systems in research and experimental settings. However, the study of the energy efficiency of bio-plausible models and the role that an evo-devo model can play to improve it are interesting and important subjects for future work.

Similar Studies

There are other studies on bio-plausible evolutionary spiking neural networks in FPGAs that also report the challenges (reviewed in section 2.5.6). Most of these studies have a limited view of bio-plausibility as only using an evolutionary or spiking neuron model. Very few studies examine a wide evo-devo view, and none of those few studies are focused on FPGAs as a platform. On the other hand, this study focuses on FPGAs and examines a wide range of different aspects of bio-plausibility. Moreover, here, the factors that promote or limit the bio-plausibility and their relation with different feasibility measures are also analysed.

Case Studies and Experiments

Analysis of the design options in chapters 4 to 6 highlighted very bio-plausible, feasible, and interesting approaches that are not applied to the case studies. For example, the promising time-multiplexed switching method is not used in the case study of the Cortex model in chapter 5. Similarly, in the consecutive case study designs many detailed bio-plausible features are included, which are not used in the final case study of the integrated system. For example a parametric flexible digital neuron model is designed, implemented, and tested in chapter 4 but those parameters are fixed and not evolved in the final integration case study. The main goals of the case studies are to provide an empirical context for investigation of the challenges in practice, and to present the reader simple examples of how such bio-plausible features can be achieved in hardware models. Applying all these available features to a final integrated bio-plausible evo-devo neural microcircuit system or conducting further experiments are separate tasks, which require more time tackling each one of the many challenges highlighted in this work.

Depth and Breadth of the Investigation

Many aspects of bio-plausibility and feasibility of the evolutionary model, fitness evaluation process and interfacing of the neural microcircuits with the environment (such as the coding/decoding methods of the input/outputs) are not addressed in this study. As explained in chapter 6 the evolutionary model

can be implemented much more efficiently in software and therefore is not bound to the limitations of the hardware implementation in FPGA. It is therefore out of the scope of this study. Fitness evaluation and interfacing are both application dependent subjects, that if not studied in the context of a specific application, can be hardly covered by a single thesis. However, both of these matters are discussed briefly in general and in more depth for the specific application of the case study. Study of the bio-plausible and feasible evolutionary models, fitness evaluation and interfacing in conjunction with such a bio-plausible evo-devo system is definitely an interesting and significant subject worth pursuing.

8.4 Future Work

It is interesting and promising to extend the work here in different directions. The work here was focussed on investigations of constraints and trade-offs between bio-plausibility and feasibility. Future work may be motivated by different aims, for example to evaluate relative performance of different bio-plausible models. One avenue is experimental study of the effects of specific bio-plausible features on emergent properties of the system. Reducing the bio-plausibility of each model by replacing the neuron, cortex, or evo-devo models (or their submodules) with simpler models and investigating its effect on the feasibility measures and emergent properties (such as evolvability and scalability) of the whole system can give us a better understanding of the importance of different bio-plausible features. For example, it is very interesting to investigate the effect of reducing the neuron model to a LIF neuron model or to compare different protein folding mappings (such as Fractal Proteins, Logistic Proteins, or no protein folding).

Another direction is to apply a revised version of the case study system to a real problem. Many possible interesting design options are proposed in this thesis that are not implemented in the successive case studies. Most of these promising options are highlighted in the summary of the design options and practical considerations sections and discussed throughout chapters 4 to 7. A non-exhaustive list of some of these revisions follows:

- Hardware Platform
  - Using newer generations of Xilinx FPGAs with faster embedded processors (section 3.2)
  - Using new Altera Stratix V FPGA devices (section 3.2)
- Neuron Model
  - Adding unsupervised learning to the synapse unit (sections 4.6.1 and 4.7)
  - Adding a synapse model (section 4.7)
  - Adding non-linear interactions to distal synapses (section 4.7)
  - Adding facilitation and depression dynamics to the synapse unit (section 4.7)
  - Upgrading the soma model to a piecewise linear approximation of Izhikevich model (section 4.7)
- Cortex Model

  - Using a time-multiplexed virtual 3D intercellular communication network (section 5.5)
  - Reverse engineering the configuration frame format for PIPs and utilising them as routing resources in the Cortex model (section 5.2.3)
  - Using one of the available high speed HWICAP custom IP cores (section 5.5)
  - Using a virtual FPGA method for reconfiguration of the Cortex (sections 5.2.3 and 5.5)
  - Adding relocatability to soma cells (sections 5.2.3 and 5.5)
  - Adding activity and health feedback circuitry to the Cortex (sections 5.2.4 and 5.5)
  - Performance improvements by wrap-around delay distribution (section 5.5)

- Evo-devo Model
  - Evolving soma parameters (sections 6.5 and7.3.3)
  - Adding branching and retraction to neurite development (section 6.5)
  - Adding activity dependent development (sections 6.5 and 7.2)
  - Adding routing resource availability feedback (section 6.5)
  - Using indirect mapping for all the loci in the genome (section 6.5)
  - Evolving development time (section 7.4)
  - Evolving cortex size (section 7.4)
  - Using Fractal Proteins (section 6.2.2)
  - Parallel implementation in GPU (section 6.4.1 and chapter 7)
  - Distributed implementation in FPGA (section 6.2.4)

- Integration
  - Using Reservoir Computing approach (section 7.2.2)
  - Applying to pattern recognition and classification tasks (section 7.2.2)
  - Using an incremental fitness function (section 7.2.3)
  - Evolving lifespan, development and evaluation time (sections 7.2.3 and 7.4)
  - Using parallel, pipelined and multithreaded development, reconfiguration and evaluation processes (section 7.4)
  - Using a higher speed datalink (section 7.4)
  - Separating the clock signals of the datalink, Cortex, and embedded processor (section 7.4)

8.5 Thesis Contributions

The research described in this thesis has resulted in a number of contributions, which for clarity are highlighted below:

- Evaluation and analysis of the challenges, factors, constraints, trade-offs, and options in achieving bio-plausibility and feasibility in the design, implementation, and testing of evo-devo neural microcircuit in FPGAs
- Introducing a new bio-plausible Digital Neuron Model with structural flexibility
- Introducing a new soma model called PLAQIF (Piecewise-Linear Approximation Quadratic Integrate and Fire) with parametric flexibility
- Presenting of an efficient implementation of Digital Neuron, PLAQIF, and Cortex models in Virtex-5 FPGAs
- Proposing time-multiplexed switched networks for intercellular communication of spiking neurons on FPGAs
- Introducing a new bio-plausible reconfigurable cortex model for growth of the Digital Neuron model on an FPGA device
- Reverse engineering of the Virtex-5 reconfiguration frame format for LUTs and SRLs and development of a C library for their fast reconfiguration using the MicroBlaze embedded processor (Appendix A and B).
- In-depth analysis of the Fractal Proteins and FGRNs
- Introducing a new bio-plausible multicellular Logistic GRN evo-devo model with structural and behavioural protein and interactions for neurodevelopment on the Cortex model
- Presenting the design, implementation, and testing of a complete integrated bio-plausible evo-devo neural microcircuit system using a commercial FPGA
- Demonstrating the value of statistical analysis of random genomes in the verification, testing and tuning of neurodevelopmental models
- Suggestions and recommendation for designer and FPGA manufacturers for achieving bio-plausibility in bio-inspired systems in FPGAs
- Providing potential solutions to usual challenges in the design of bio-plausible evo-devo neural systems in FPGAs that, in many cases, can be reused in other bio-plausible designs

8.5.1 Summary

Evolving bio-plausible artificial brains in commercial FPGAs is a daunting task. With the swift advances in reconfigurable technologies and the promising emergent properties of bio-plausible systems, it is challenge worth every effort. A better understanding of the challenges is the first step towards tackling them. This research was aimed at practical investigation of the challenges, factors, constraints, trade-offs, and options in achieving bio-plausibility in the design, implementation and testing of evo-devo neural microcircuits feasible in FPGAs. It achieved this aim with a broad but deep and practical investigation of different aspects of bio-plausibility and feasibility of evo-devo neural microcircuits in FPGAs. This thesis is a stepping stone towards future bio-plausible designs: artificial brains in silicon that evolve, grow, adapt and learn.

Appendices

Appendix A

Virtex-5 Reconfiguration Frame Format

The high-level reconfiguration bitstream format of the Virtex-5 is well explained in Virtex-5 FPGA Configuration User Guide from Xilinx [411] (Frame Addressing pp. 131). The smallest reconfigurable unit in the configuration memory is a frame of 1312 bits = 41 x 32 bit = 20 (rows of CLBs) x 64 (bits per CLB row) + 32 bits for ECC and HCLK bits. These frames span vertically over a HCLKROW (big blocks of CLBS sharing same global clock signals). Based on the device size, number of these HCLKROWs is different. There are 6 HCLKROWs in XC5VLX50T. These rows and how they are addressed is explained clearly in pp. 133 of [411]. The major address can be simply calculated using the layout of the device knowing which column is containing which resource (CLB, IOB, DSP, DRAM) (Figure 6.12 in [411]).

In each major address (a column) there are minor addresses. For CLB columns there are 36 minor addresses (frames). Frames 0 to 25 are used for interconnection (except for the clock column). For all columns except for CLB and clock columns, frames 26 and 27 are used for interface to the block. From this point on it is not explained anywhere and is considered proprietary.

A LUT6 data (64 bits) is distributed over 4 frames: 26 to 29 for the left slices in a CLB (even X_{slice}), and frames 31 to 34 for the right slice in a CLB (odd X_{slice}). That is 4x64 bits for 4 LUT6s in a slice. But the data for each LUT is allocated in a peculiar order. Table A.1 shows how content of the LUTs are distributed over four different frames with different minor frame addresses. Word number n can be calculated based on the Y location of the slice in a frame (Y_{slice}) as: $n = 2Y_{slice} + (Y_{slice} > 9)$. Table A.2 gives the detailed internal addresses of each of 64 bits of LUT contents in the data blocks dedicated to each LUT (shown in table A.1) based on the type of the slice (SLICEM or SLICEL). Library functions that use this information to reconfigure Virtex-5 LUT and SRL/RAM contents are presented in appendix B.

Different methods and tools are now available for reverse engineering the format of the rest of the frames and columns [83, 28, 279].

Table A.1: Distribution of the LUTs contents over four different frames. Minor addresses of the frames for even and odd slices are also given.

Minor Address	Slice		Word n		Word n+1	
	Even	Odd	Bits	Bits	Bits	Bits
	$X_{slice} \mod 2 = 0$	$X_{slice} \mod 2 = 1$	31 - 16	15 - 0	31 - 16	15 - 0
	32	26				
	33	27	B LUT	A LUT	D LUT	C LUT
	34	28	Data	Data	Data	Data
	35	29				

Table A.2: Detailed addresses of the 64 bits of the LUT contents in the data blocks dedicated to each LUT. Two first rows of numbers are the bit numbers of the words and the rest are the address of each bit in the LUT.

	Slice		Bits in a Word															
	Even	Odd	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	$X_{slice} \mod 2 = 0$	$X_{slice} \mod 2 = 1$	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SLICEM	32	26	1	4	9	12	17	20	25	28	33	36	41	44	49	52	57	60
	33	27	0	5	8	13	16	21	24	29	32	37	40	45	48	53	56	61
	34	28	3	6	11	14	19	22	27	30	35	38	43	46	51	54	59	62
	35	29	2	7	10	15	18	23	26	31	34	39	42	47	50	55	58	63
SLICEL	32	26	3	6	11	14	19	22	27	30	35	38	43	46	51	54	59	62
	33	27	2	7	10	15	18	23	26	31	34	39	42	47	50	55	58	63
	34	28	0	5	8	13	16	21	24	29	32	37	40	45	48	53	56	61
	35	29	1	4	9	12	17	20	25	28	33	36	41	44	49	52	57	60

Appendix B

Library Functions for Faster Reconfiguration through MicroBlaze

```
/******************************************************************************
* Filename:         editlut.h
* Version:           2.19.a
* Description:      Header File for LUT reconfiguration functions
* Author: Hooman Shayani
******************************************************************************/

#ifndef EDITLUT_H_
#define EDITLUT_H_

#include "xbasic_types.h"
#include "xstatus.h"
#include "xhwicap.h"
#include "hwicap_header.h"
#include "cortexip.h"

#define N 1
#define E 2
#define S 3
#define W 4
#define D 5


#define LUTA 0
#define LUTB 1
#define LUTC 2
#define LUTD 3
void Translate(u32 * Bits);
XStatus SetLUT(XHwIcap * InstancePtr, int X, int Y,int LUT, Xuint32 *LUTBits);
XStatus SetSRR16(XHwIcap * InstancePtr, int X, int Y,int LUT, Xuint16 LUTBits);
XStatus SetSRR32(XHwIcap * InstancePtr, int X, int Y,int LUT, Xuint32 LUTBits);
void DecodeSRR16(Xuint32 * Bits, Xuint16 SHRBits);
void DecodeSRR32(Xuint32 * Bits, Xuint32 SHRBits);

#endif /*EDITLUT_H_*/
```

```
/******************************************************************************
* Filename:         editlut.c
* Version:           2.73.a
* Description:      Source File for LUT reconfiguration functions
* Author: Hooman Shayani
******************************************************************************/

#include <stdio.h>
#include "xbasic_types.h"
#include "xstatus.h"
#include "xparameters.h"
#include "xhwicap.h"
#include "xhwicap_clb_lut.h"
#include "xhwicap_i.h"
```

```
#define V5_ROWS_PER_HCLKROW 20
#define SDP_SLICE_M_COL 10 //this is only for XCVLX50T
#define READ_FRAME_SIZE 20

#define printf xil_printf

Xuint32 Buffer[250];//extern

extern XHwIcap *icapptr; /* The instance of the HWICAP device */

#if (XHI_FAMILY == XHI_DEV_FAMILY_V5)

/*****************************************************************************/
/**
*
* Writes four frame from the specified buffer and puts it in the device
* (ICAP).
*
* @param InstancePtr is a pointer to the XHwIcap instance.
* @param Top - top (0) or bottom (1) half of device
* @param Block - Block Address (XHI_FAR_CLB_BLOCK,
*  XHI_FAR_BRAM_BLOCK, XHI_FAR_BRAM_INT_BLOCK)
* @param HClkRow - selects the HClk Row
* @param MajorFrame - selects the column
* @param MinorFrame - selects frame inside column
* @param FrameData is a pointer to the first frame that is to be written
* to the device.
*
* @return XST_SUCCESS else XST_FAILURE.
*
* @note This is a blocking function.
* This function is used in conjunction with the function
* XHwIcap_DeviceRead2Frames. This function is used to write back
* the frames of data read using the XHwIcap_DeviceRead2Frames.
*
*****************************************************************************/
int XHwIcap_DeviceWrite4Frames(XHwIcap *InstancePtr, long Top, long Block,
long HClkRow, long MajorFrame, long MinorFrame,
u32 *FrameData)
{

u32 HeaderWords;
u32 Packet;
u32 Data;
u32 TotalWords;
int Status;
u32 WriteBuffer[READ_FRAME_SIZE];
u32 Index =0;

Xil_AssertNonvoid(InstancePtr != NULL);
Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
Xil_AssertNonvoid(FrameData != NULL);


/*
 * DUMMY and SYNC
 */
WriteBuffer[Index++] = XHI_DUMMY_PACKET;
WriteBuffer[Index++] = XHI_SYNC_PACKET;
WriteBuffer[Index++] = XHI_NOOP_PACKET;
WriteBuffer[Index++] = XHI_NOOP_PACKET;

/*
 * Reset CRC
 */
Packet = XHwIcap_TypelWrite(XHI_CMD) | 1;
Data = XHI_CMD_RCRC;
WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = Data;
WriteBuffer[Index++] = XHI_NOOP_PACKET;
WriteBuffer[Index++] = XHI_NOOP_PACKET;


/*
 * Bypass CRC
```

```
 */
Packet = XHwIcap_Type1Write(XHI_COR) | 1;
Data = 0x10042FDD;
WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = Data;

/*
 * ID register
 */
Packet = XHwIcap_Type1Write(XHI_IDCODE) | 1;
Data = InstancePtr->DeviceIdCode;
WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = Data;


/*
 * Setup FAR
 */
Packet = XHwIcap_Type1Write(XHI_FAR) | 1;
#if XHI_FAMILY == XHI_DEV_FAMILY_V4 /* Virtex 4 */
Data = XHwIcap_SetupFarV4(Top, Block, HClkRow,  MajorFrame, MinorFrame);
#elif XHI_FAMILY == XHI_DEV_FAMILY_V5 /* Virtex 5 */
Data = XHwIcap_SetupFarV5(Top, Block, HClkRow,  MajorFrame, MinorFrame);
#endif

WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = Data;

/*
 * Setup CMD register - write configuration
 */
Packet = XHwIcap_Type1Write(XHI_CMD) | 1;
Data = XHI_CMD_WCFG;
WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = Data;
WriteBuffer[Index++] = XHI_NOOP_PACKET;


/*
 * Setup Packet header.
 */
TotalWords = InstancePtr->WordsPerFrame + (InstancePtr->WordsPerFrame << 2);
if (TotalWords < XHI_TYPE_1_PACKET_MAX_WORDS)  {
/*
 * Create Type 1 Packet.
 */
Packet = XHwIcap_Type1Write(XHI_FDRI) | TotalWords;
WriteBuffer[Index++] = Packet;
HeaderWords = 18;
}
else {

/*
 * Create Type 2 Packet.
 */
Packet = XHwIcap_Type1Write(XHI_FDRI);
WriteBuffer[Index++] = Packet;

Packet = XHI_TYPE_2_WRITE | TotalWords;
WriteBuffer[Index++] = Packet;

HeaderWords = 19;
}

/*
 * Write the Header data into the FIFO and intiate the transfer of
 * data present in the FIFO to the ICAP device
 */
Status = XHwIcap_DeviceWrite(InstancePtr, (u32 *)&WriteBuffer[0], HeaderWords);
if (Status != XST_SUCCESS)  {
return XST_FAILURE;
}

/*
 * Write the modified frame data.
 */
```

```
Status = XHwIcap_DeviceWrite(InstancePtr,
(u32 *) &FrameData[InstancePtr->WordsPerFrame + 1],
InstancePtr->WordsPerFrame<<2);
if (Status != XST_SUCCESS) {
return XST_FAILURE;
}

/*
 * Check if the ICAP device is Busy with the last Write/Read
 */
while (XHwIcap_IsDeviceBusy(InstancePtr) == TRUE) {
;
}
//xil_printf("totwrds:%ld",TotalWords);
/*
 * Write out the pad frame. The pad frame was read from the device before
 * the data frame.
 */
Status = XHwIcap_DeviceWrite(InstancePtr, (u32 *) &FrameData[1],
    InstancePtr->WordsPerFrame);
if (Status != XST_SUCCESS) {
return XST_FAILURE;
}



/* Add CRC */
Index = 0;
Packet = XHwIcap_Type1Write(XHI_CRC) | 1;
WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = XHI_DISABLED_AUTO_CRC;

/* Park the FAR */
Packet = XHwIcap_Type1Write(XHI_FAR) | 1;

#if XHI_FAMILY == XHI_DEV_FAMILY_V4 /* Virtex 4 */
Data = XHwIcap_SetupFarV4(0, 0, 3, 33, 0);
#elif XHI_FAMILY == XHI_DEV_FAMILY_V5 /* Virtex 5 */
Data = XHwIcap_SetupFarV5(0, 0, 3, 33, 0);
#endif

WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] =  Data;

/* Add CRC */
Packet = XHwIcap_Type1Write(XHI_CRC) | 1;
WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = XHI_DISABLED_AUTO_CRC;

/*
 * Intiate the transfer of data present in the FIFO to
 * the ICAP device
 */
Status = XHwIcap_DeviceWrite(InstancePtr, &WriteBuffer[0], Index);
if (Status != XST_SUCCESS)  {
return XST_FAILURE;
}

/*
 * Send DESYNC command
 */
Status = XHwIcap_CommandDesync(InstancePtr);
if (Status != XST_SUCCESS) {
return XST_FAILURE;
}

return XST_SUCCESS;
};

/*****************************************************************************/
/**
*
* Reads two frames from the device and puts it in memory specified by the user.
*
* @param InstancePtr - a pointer to the XHwIcap instance to be worked on.
* @param Top - top (0) or bottom (1) half of device
```

```
* @param Block - Block Address (XHI_FAR_CLB_BLOCK,
* XHI_FAR_BRAM_BLOCK, XHI_FAR_BRAM_INT_BLOCK)
* @param HClkRow - selects the HClk Row
* @param MajorFrame - selects the column
* @param MinorFrame - selects frame inside column
* @param FrameBuffer is a pointer to the memory where the frames read
* from the device are stored
*
* @return XST_SUCCESS else XST_FAILURE.
*
* @note This is a blocking call.
*
******************************************************************************/
int XHwIcap_DeviceRead2Frames(XHwIcap *InstancePtr, long Top, long Block,
long HClkRow, long MajorFrame, long MinorFrame,
u32 *FrameBuffer)
{

u32 Packet;
u32 Data;
u32 TotalWords;
int Status;
u32 WriteBuffer[READ_FRAME_SIZE];
u32 Index = 0;

Xil_AssertNonvoid(InstancePtr != NULL);
Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
Xil_AssertNonvoid(FrameBuffer != NULL);
/*
 * DUMMY and SYNC
 */
WriteBuffer[Index++] = XHI_DUMMY_PACKET;
WriteBuffer[Index++] = XHI_SYNC_PACKET;
WriteBuffer[Index++] = XHI_NOOP_PACKET;
WriteBuffer[Index++] = XHI_NOOP_PACKET;

/*
 * Reset CRC
 */
Packet = XHwIcap_Type1Write(XHI_CMD) | 1;
WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = XHI_CMD_RCRC;
WriteBuffer[Index++] = XHI_NOOP_PACKET;
WriteBuffer[Index++] = XHI_NOOP_PACKET;

/*
 * Setup CMD register to read configuration
 */
Packet = XHwIcap_Type1Write(XHI_CMD) | 1;
WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = XHI_CMD_RCFG;
WriteBuffer[Index++] = XHI_NOOP_PACKET;
WriteBuffer[Index++] = XHI_NOOP_PACKET;
WriteBuffer[Index++] = XHI_NOOP_PACKET;

/*
 * Setup FAR register.
 */
Packet = XHwIcap_Type1Write(XHI_FAR) | 1;
#if XHI_FAMILY == XHI_DEV_FAMILY_V4 /* Virtex 4 */
Data = XHwIcap_SetupFarV4(Top, Block, HClkRow,  MajorFrame, MinorFrame);
#elif XHI_FAMILY == XHI_DEV_FAMILY_V5 /* Virtex 5 */
Data = XHwIcap_SetupFarV5(Top, Block, HClkRow,  MajorFrame, MinorFrame);
#endif
WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = Data;

/*
 * Setup read data packet header.
 * The frame will be preceeded by a dummy frame, and we need to read one
 * extra word - see Configuration Guide Chapter 8
 */
TotalWords = InstancePtr->WordsPerFrame+(InstancePtr->WordsPerFrame << 1) + 1;

/*
 * Create Type one packet
```

```
 */
Packet = XHwIcap_Type1Read(XHI_FDRO);
WriteBuffer[Index++] = Packet;
Packet = XHI_TYPE_2_READ | TotalWords;
WriteBuffer[Index++] = Packet;
WriteBuffer[Index++] = XHI_NOOP_PACKET;
WriteBuffer[Index++] = XHI_NOOP_PACKET;

/*
 * Write the data to the FIFO and initiate the transfer of data
 * present in the FIFO to the ICAP device
 */
Status = XHwIcap_DeviceWrite(InstancePtr, (u32 *)&WriteBuffer[0], Index);
if (Status != XST_SUCCESS)  {
return XST_FAILURE;
}

/*
 * Wait till the write is done.
 */
while (XHwIcap_IsDeviceBusy(InstancePtr) != FALSE);

/*
 * Read the frame of the data including the NULL frame.
 */
Status = XHwIcap_DeviceRead(InstancePtr, FrameBuffer, TotalWords);
if (Status != XST_SUCCESS)  {
return XST_FAILURE;
}


/*
 * Send DESYNC command
 */
Status = XHwIcap_CommandDesync(InstancePtr);
if (Status != XST_SUCCESS)  {
return XST_FAILURE;
}

return XST_SUCCESS;
};


//this array says bit i in the LUT addressing space should be in the bit LUT_Mapping[i] of the Bits[2] in SetLUT(...,Bits)
int LUT_Mapping[64]={
31,15,63,47,14,30,46,62,
29,13,61,45,12,28,44,60,
27,11,59,43,10,26,42,58,
25,9,57,41,8,24,40,56,
23,7,55,39,6,22,38,54,
21,5,53,37,4,20,36,52,
19,3,51,35,2,18,34,50,
17,1,49,33,0,16,32,48};

//translates a 64 bits to the reconfigurable format that can be used by SetLUT()
void Translate(u32 * Bits)
{
int i,j,k;
u32 Data[2]={0,0};

for(i=0; i<2;i++)
{
for(j=0; j<32;j++)
{
if((Bits[i]>>j)&0x01) //if bit i,j is input (LUT address space) is one
{
k=LUT_Mapping[i*32+j];
Data[k>>5]|= (0x01<<(k & 0x1f));
}
}
}
Bits[0]=Data[0];
Bits[1]=Data[1];
}
```

```
XStatus SetLUT(XHwIcap *InstancePtr, int X, int Y,int LUT, Xuint32 *LUTBits)
{
long Bottom = (Y<(InstancePtr->Rows>>1))? 1 : 0;; //
int ClkZone=Y/20;
long HClkRow = ClkZone+ (Bottom?(InstancePtr->HClkRows-1)-(ClkZone<<1): 0) - (InstancePtr->HClkRows>>1);//
int YinFrame = Y%20;
long MajorAddr =0;
long MinorAddr ;
int Col = (X>>1)+1;
u16 * Skips = InstancePtr->SkipCols;
int Word;

XStatus Status;
u32 Data32;
Xuint32 Bits[2];
Bits[0]=LUTBits[0]; Bits[1]=LUTBits[1];
Translate(Bits);

MajorAddr=0;
while(Col > *(Skips++))
MajorAddr++;
MajorAddr += Col;
MinorAddr = (X & 0x01) ? 26 : 32 ;
Word = 42 + (LUT>>1) + ((YinFrame>9)?1:0) + (YinFrame<<1);
// +1 extra word because for the transmission buffer to flush
// +1 extra frame (+41) because it is read so for the frame buffer to flush
// +1 word if it is LUT C or D
// +1 word if it is above the clock line
//and 2 word for every slice in the frame

//if (X==19) printf("\n\rX=%d Y=%d LUT=%d Bottom=%d, Block=%d, HClkRo=%d, MajorAddr=%d, MinorAddr=%d Word=%d\n\r",X,Y,LUT,Bottom, XHI_FAR_CLB_BLOCK,HClkRow, Maj

Status = XHwIcap_DeviceRead2Frames(InstancePtr, Bottom, XHI_FAR_CLB_BLOCK,HClkRow, MajorAddr, MinorAddr+2, &Buffer[82]);
if (Status != XST_SUCCESS) {
printf("DeviceReadFrame failed:%d\n\r",Status);
return XST_FAILURE;
}

Status = XHwIcap_DeviceRead2Frames(InstancePtr, Bottom, XHI_FAR_CLB_BLOCK,HClkRow, MajorAddr, MinorAddr, Buffer);
if (Status != XST_SUCCESS) {
printf("DeviceReadFrame failed:%d\n\r",Status);
return XST_FAILURE;
}


if( ((X & 0x03) == 0) || (X==SDP_SLICE_M_COL) ) //if a SLICE_M
{
if (LUT & 0x01)//B or D
{
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff) | (Bits[0] << 16);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff) | (Bits[0] & 0xffff0000);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff) | (Bits[1] << 16);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff) | (Bits[1] & 0xffff0000);
Buffer[Word]=Data32;
}
else // A or C
{
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff0000) | (Bits[0] & 0xffff);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff0000) | (Bits[0] >> 16);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
```

```
Data32 = (Data32 & 0xffff0000) | (Bits[1] & 0xffff);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff0000) | (Bits[1] >> 16);
Buffer[Word]=Data32;
}
}
else //if a SLICE_L
{
if (LUT & 0x01) //B or D
{
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff) | (Bits[1] & 0xffff0000);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff) | (Bits[1] << 16);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff) | (Bits[0] << 16);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff) | (Bits[0] & 0xffff0000);
Buffer[Word]=Data32;
}
else // A or C
{
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff0000) | (Bits[1] >> 16);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff0000) | (Bits[1] & 0xffff);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff0000) | (Bits[0] & 0xffff);
Buffer[Word]=Data32;
Word+=41;
Data32 = Buffer[Word];
Data32 = (Data32 & 0xffff0000) | (Bits[0] >> 16);
Buffer[Word]=Data32;
}
}


Status = XHwIcap_DeviceWrite4Frames(InstancePtr, Bottom, XHI_FAR_CLB_BLOCK,HClkRow, MajorAddr, MinorAddr, Buffer);
if (Status != XST_SUCCESS) {
printf("DeviceWrite4Frame failed:%d\n\r",Status);
return XST_FAILURE;
}
return XST_SUCCESS;
}
#endif


XStatus SetSRR16(XHwIcap *InstancePtr, int X, int Y,int LUT, Xuint16 SHRBits)
{
Xuint32 lutbits[2];
Xuint32 bits=0;
int i;
for(i=0; i<16 ;i++)
{
bits = (bits<<2) | ((SHRBits &(0x01<<i))>>i)<<1;
}
lutbits[0]=0;
lutbits[1]=bits;
return SetLUT(InstancePtr,X,Y,LUT,lutbits);
}

void DecodeSRR16(Xuint32 * Bits, Xuint16 SHRBits)
{
Xuint32 lutbits[2];
```

```
Xuint32 bits=0;
int i;
for(i=0; i<16 ;i++)
{
bits = (bits<<2) | ((SHRBits &(0x01<<i))>>i)<<1;
}
Bits[0]=0;
Bits[1]=bits;
}


XStatus SetSRR32(XHwIcap *InstancePtr, int X, int Y,int LUT, Xuint32 SHRBits)
{
Xuint32 lutbits[2];
Xuint32 bits0=0;
Xuint32 bits1=0;
int i;
for(i=0; i<16 ;i++)
{
bits1 = (bits1<<2) | ((SHRBits &(0x01<<i))>>i)<<1;
//bits1 = (bits1<<2) | ((SHRBits>>i)&0x01)<<1;
}
for(i=16; i<32 ;i++)
{
bits0 = (bits0<<2) | ((SHRBits &(0x01<<i))>>i)<<1;
//bits0 = (bits0<<2) | ((SHRBits>>i)&0x01)<<1;
}
lutbits[0]=bits0;
lutbits[1]=bits1;
return SetLUT(InstancePtr,X,Y,LUT,lutbits);
}


void DecodeSRR32(Xuint32 * Bits, Xuint32 SHRBits)
{
Xuint32 lutbits[2];
Xuint32 bits0=0;
Xuint32 bits1=0;
int i;
for(i=0; i<16 ;i++)
{
bits1 = (bits1<<2) | ((SHRBits &(0x01<<i))>>i)<<1;
//bits1 = (bits1<<2) | ((SHRBits>>i)&0x01)<<1;
}
for(i=16; i<32 ;i++)
{
bits0 = (bits0<<2) | ((SHRBits &(0x01<<i))>>i)<<1;
//bits0 = (bits0<<2) | ((SHRBits>>i)&0x01)<<1;
}
Bits[0]=bits0;
Bits[1]=bits1;
}
```

Appendix C

Embedded System Source Code

This C program that runs on a MicroBlaze was used for testing the Cortex reconfigurability, different Digital Neuron behaviours, and receiving and execution of reconfiguration commands from PC. It is using the functions in appendix B

```
/*****************************************************************************
* Filename:         configtest.c
* Version:           2.13.a
* Description:      Source File for testing reconfiguration of the Cortex
* Author: Hooman Shayani
*****************************************************************************/

#define EN_UART_SELFTEST
#define EN_COL_CONFIG
//#define EN_FROMPC
//#define EN_CODE_SECTION_1
//#define EN_CODE_SECTION_2
#define EN_CODE_SECTION_3
//#define EN_CODE_SECTION_4

#include <stdio.h>
#include "xparameters.h"
#include "xenv_standalone.h"
#include "uartlite_header.h"
#include "xhwicap.h"
#include "hwicap_header.h"
#include "editlut.h"
#include "cortex.h"
#include "xuartlite.h"
#include "xuartlite_l.h"


static XHwIcap  HwIcap;
XHwIcap *icapptr = &HwIcap;
Xuint16 CortexColBuff[3*120];
extern XUartLite UartLite;
extern int Xil_AssertWait;



void mydelay(int period);
void ConfigTestAxonalRouting(int i, int j);
inline void WriteSpikeOutWord(int SpikeInIndex, Xuint32 SpikeOutWord);
void ResetSpikeCounter(int SpikeInIndex);
int TestSomaResponse(int SpikeInIndex);
int GetSomaResponse(int SpikeInIndex);
void ProcessPCLink();

int main()
{

   XCACHE_ENABLE_ICACHE();
   XCACHE_ENABLE_DCACHE();

   print("---Entering main---\n\r");
```

```
   Xil_AssertWait = FALSE;

   /*
    * Peripheral SelfTest will not be run for RS232_Uart_1
    * because it has been selected as the STDOUT device
    */


// UART lite self test
#ifdef EN_UART_SELFTEST
   {
      XStatus status;

      print("\r\nRunning UartLiteSelfTestExample() for mdm_0...\r\n");
      status = UartLiteSelfTestExample(XPAR_MDM_0_DEVICE_ID);
      if (status == 0) {
         print("UartLiteSelfTestExample PASSED\r\n");
      }
      else {
         print("UartLiteSelfTestExample FAILED\r\n");
      }

/*
      char Input;
  int iii=0;
      u8 testbuffer[20];

      for(iii=0;iii<12;)
      {
        Input = //getchar();
        XUartLite_RecvByte(STDIN_BASEADDRESS);

        if (Input != EOF)
        {
          //xil_printf("%d\n\r", (int)Input);
          testbuffer[iii++]=(char)Input;
        }

      }
      xil_printf("hello world!");
      xil_printf("done");*/
   }

#endif

   {
      XStatus status;


      print("\r\n Runnning HwIcapTestAppExample() for xps_hwicap_0...\r\n");

      status = HwIcapTestAppExample(XPAR_XPS_HWICAP_0_DEVICE_ID);

      if (status == 0) {
         print("HwIcapTestAppExample PASSED\r\n");
      }
      else {
         print("HwIcapTestAppExample FAILED\r\n");
      }
   }


   //********************************** FROM PC ************************************************
   //the next section will get the config and sim commands from PC and returns the results to PC
   //*******************************************************************************************
#ifdef EN_FROMPC
//   ProcessPCLink();
#endif


//********************************** CODE SECTION 1 ************************************************
// the following code configures the left most edge of the cortex as axon loop backs and tests the
// spike counters for all the 120 spike outs of the cortex. uncomment it and comment the other code sections
// that may be incompatible in order to use it.

#ifdef EN_CODE_SECTION_1  // code section 1 begins here. switch this to comment/uncomment the code section
```

```
{
print("\n\rStarting Code section 1: testing spike counters and spike generation...\n\r");

ResetAndDisableCortexClock();

   int j;
#ifdef EN_COL_CONFIG
   ClearColBuff(CortexColBuff);
   for(j=0; j<120 ; j++)
   SetGlialColBuff(CortexColBuff,j,N,E,S,W,0,N,E,S,W,0,0);
   ConfigCortexCol(0,CortexColBuff);

   #else
   for(j=0; j<120 ; j++)
   {
    ConfigGlialAxon(0,j,N,E,S,W);
    ConfigGlialDend(0,j,N,E,S,W); //loopback both axons and dendrite signals on the leftmost edge of the cortex
   }
#endif
   EnableCortexClock();

   int SpikeInIndex,b;
   int n=36;
   Xuint32 a;

   for (SpikeInIndex=0; SpikeInIndex<120; SpikeInIndex++) //for each spike in
   {
    for(n=0; n<64;n++)
    {
    a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 8*(SpikeInIndex/8)); //reset the counter

    for (b=0; b<n;b++) //generate n pulses in that spike in
    {
CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,4*(SpikeInIndex/30),0x00000001<<(SpikeInIndex%30));
}
a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 8*(SpikeInIndex/8)); //read and lock the rest
if((SpikeInIndex/4)%2==1) //if it is in the second half
a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 4+8*(SpikeInIndex/8)); //read and lock the rest
a = (a>>(6*(SpikeInIndex%4))) & 0x3f;
if (a==n)
xil_printf("SpikeIn %d with %d pulses is OK                              \r",SpikeInIndex,n);
else{
xil_printf("\n\rSpikeIn %d is not working with %d pulses.********\r\n",SpikeInIndex,n);
return 1;
}
    }
   }
   print("\rCode section 1 finished successfully: Spike counters and spike generation OK.             \n\r");

}
#endif // code section 1 ends here

   //************************ CODE SECTION 2 ******************************
   //The following code configures the cortex to test synapses and different routings in the glial cells
#ifdef EN_CODE_SECTION_2 // code section 2 begins here, switch this to comment/uncomment
{
   print("\n\rStarting Code section 2: testing axonal routing...\n\r");

   int i,j,vectorindex,ii,jj;
   int SpikeInIndex,b;
   int n=36;
   Xuint32 a;
   char Params[8] =   {-4,-4,-3,-3,3,3,4,4};

   ResetAndDisableCortexClock();

   //i=3; j=0;
   for(i=0; i<10 ; i+=3)
   for(j=0; j<CORTEXROWS ; j+=4)
   {
   ConfigTestAxonalRouting(i,j);

   EnableCortexClock();

   SpikeInIndex = j;
```

```
   for(n=0; n<64;n++) //testing axonal routing around the soma cell
   {
a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 8*(SpikeInIndex/8)); //reset the counter

for (b=0; b<n;b++) //generate n pulses in that spike in
{
CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,4*(SpikeInIndex/30),0x00000001<<(SpikeInIndex%30));
}
int waitcounter;
for(waitcounter=0; waitcounter<10; waitcounter++) //kill some time to make sure that all pulses are received
;
a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 8*(SpikeInIndex/8)); //read and lock the rest
if((SpikeInIndex/4)%2==1) //if it is in the second half
a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 4+8*(SpikeInIndex/8)); //read and lock the rest
a = (a>>(6*(SpikeInIndex%4))) & 0x3f;
if (a==n)
xil_printf("Axonal rout around soma cell @(%d,%d) with %d pulses is OK                    \r",i+1,j,n);
else
{
xil_printf("\n\rAxonal route around soma cell at @(%d,%d) is not working with %d pulses.********\r\n",i+1,j,n);
return 1;
}
   }

   //testing dendritic routing and synapses around the soma cell
   ResetAndDisableCortexClock();

   ConfigGlialDend(i,j,S,D,N,W); //1
   ConfigGlialSynapse(i,j,S,E,0);

   ConfigGlialDend(i,(j+1)%CORTEXROWS,N,W,D,E); //2
   if(i>0) ConfigGlialDend(i-1,(j+1)%CORTEXROWS,0,E,0,0);
   ConfigGlialSynapse(i,(j+1)%CORTEXROWS,S,S,0);

   ConfigGlialDend(i,(j+2)%CORTEXROWS,N,D,E,W); //3
   ConfigGlialSynapse(i,(j+2)%CORTEXROWS,S,S,0);

   ConfigGlialDend(i+1,(j+2)%CORTEXROWS,N,S,W,D); //4
   ConfigGlialSynapse(i+1,(j+2)%CORTEXROWS,W,E,0);

   ConfigGlialDend(i+2,(j+2)%CORTEXROWS,N,E,D,S); //5
   ConfigGlialSynapse(i+2,(j+2)%CORTEXROWS,W,W,0);

   ConfigGlialDend(i+2,(j+1)%CORTEXROWS,D,E,S,W); //6
   ConfigGlialSynapse(i+2,(j+1)%CORTEXROWS,N,N,0);

   ConfigGlialDend(i+2,(j)%CORTEXROWS,W,E,S,D); //7
   ConfigGlialSynapse(i+2,(j)%CORTEXROWS,N,N,0);

   ConfigGlialDend(i+2,(j-1+CORTEXROWS)%CORTEXROWS,D,E,S,N); //8
   ConfigGlialSynapse(i+2,(j-1+CORTEXROWS)%CORTEXROWS,N,W,0);

   ConfigGlialDend(i+1,(j-1+CORTEXROWS)%CORTEXROWS,W,N,S,D); //9
   ConfigGlialSynapse(i+1,(j-1+CORTEXROWS)%CORTEXROWS,E,E,0);

   ConfigGlialDend(i,(j-1+CORTEXROWS)%CORTEXROWS,E,D,S,W); //10
   ConfigGlialSynapse(i,(j-1+CORTEXROWS)%CORTEXROWS,E,N,0);

   if(i==0)
   ConfigGlialAxon(i,j,W,0,0,E); //get the axonal output from the soma axon and keep the presynaptic axon connected to axonal input of the cortex
   else
   ConfigGlialAxon(i,(j-1+CORTEXROWS)%CORTEXROWS,E,0,0,N);

   ConfigSomaParam(i+1,j,-16384,-1,0x0001,Params);//config the soma for a basic functionality

   EnableCortexClock();

   int TestVector[10][2]={{0,0},{0,1},{0,2},{1,2},{2,2},{2,1},{2,0},{2,-1},{1,-1},{0,-1}};

   if((a=TestSomaResponse(j))!=0)
{
xil_printf("\n\rSoma cell at @(%d,%d) is not working with no synapse.********\r\n",i+1,j);
return 1;
}
   ResetAndDisableCortexClock();
   for(vectorindex=0; vectorindex<10; vectorindex++)
```

```
   {
   ii=i+TestVector[vectorindex][0];
   jj=(j+TestVector[vectorindex][1]+CORTEXROWS)%CORTEXROWS;
#ifdef EN_COL_CONFIG
   ClearColBuff(CortexColBuff);
   SetGlialColBuff(CortexColBuff,j+2,  N,S,W,D,E,0,W,0,0,W, (ii==i+1) && (jj==j+2)? 0x7ff0 : 0);
   SetSomaColBuff (CortexColBuff,j,-16384,-1,0x0001,Params,1,1,1,1,1,1);
   SetGlialColBuff(CortexColBuff,(j-1+CORTEXROWS)%CORTEXROWS,  W,N,S,D,E,0,0,0,E,E, (ii==i+1) && (jj==((j-1+CORTEXROWS)%CORTEXROWS))? 0x7ff0 : 0 );
   ConfigCortexCol(i+1,CortexColBuff);
   if(ii!=i+1)
   ConfigGlialSynapseWeight(ii,jj,0x7ff0);
#else
   ConfigSomaParam(i+1,j,-16384,-1,0x0001,Params);//config the soma for a basic functionality
   ConfigGlialSynapseWeight(ii,jj,0x7ff0);
#endif
   EnableCortexClock();
   mydelay(100);
   if((a=TestSomaResponse(j))!=1)
{
xil_printf("\n\rSoma @(%d,%d) with a synapse @(%d,%d). Sending %d spikes in it axon.********\r\n",i+1,j,ii,jj,a);
return 1;
}
   ResetAndDisableCortexClock();
   ConfigGlialSynapseWeight(ii,jj,0);
   }

   xil_printf("\rAll around soma cell @(%d,%d) are working OK.                \r\n",i+1,j);
   }
   print("\rFinishing code section 2: Glial Cells are OK.                              \n\r");
}
#endif

//************************ CODE SECTION 3 *******************************
//The following code configures the second from bottom left soma cell for testing different behaviours and parameters
#ifdef EN_CODE_SECTION_3 // code section 3 begins here, switch this to comment/uncomment
{
print("\n\rStarting Code section 3: testing bottom left soma cell for different behaviours...\n\r");

int i,j;
int b;
Xuint32 a;

ResetAndDisableCortexClock();

i=0; j=4; //use the second soma cell so that all the spike ins can be used and connected to synapses around it.
{
   ConfigGlialDend(i,j,S,D,N,W); //16
   ConfigGlialSynapse(i,j,W,E,32);

   ConfigGlialDend(i,(j+1)%CORTEXROWS,N,W,D,E); //32
   ConfigGlialSynapse(i,(j+1)%CORTEXROWS,W,S,64);

   ConfigGlialDend(i,(j+2)%CORTEXROWS,N,D,E,W); //64
   ConfigGlialSynapse(i,(j+2)%CORTEXROWS,W,S,128);

   ConfigGlialDend(i+1,(j+2)%CORTEXROWS,N,S,W,D); //128
   ConfigGlialSynapse(i+1,(j+2)%CORTEXROWS,W,E,256);

   ConfigGlialDend(i+2,(j+2)%CORTEXROWS,N,E,D,S); //256
   ConfigGlialSynapse(i+2,(j+2)%CORTEXROWS,W,W,512);

   ConfigGlialDend(i+2,(j+1)%CORTEXROWS,D,E,S,W); //512
   ConfigGlialSynapse(i+2,(j+1)%CORTEXROWS,N,N,1024);

   ConfigGlialDend(i+2,(j)%CORTEXROWS,W,E,S,D); //1
   ConfigGlialSynapse(i+2,(j)%CORTEXROWS,S,N,2);

   ConfigGlialDend(i+2,(j-1+CORTEXROWS)%CORTEXROWS,D,E,S,N); //2
   ConfigGlialSynapse(i+2,(j-1+CORTEXROWS)%CORTEXROWS,W,W,4);

   ConfigGlialDend(i+1,(j-1+CORTEXROWS)%CORTEXROWS,W,N,S,D); //4
   ConfigGlialSynapse(i+1,(j-1+CORTEXROWS)%CORTEXROWS,W,E,8);

   ConfigGlialDend(i,(j-1+CORTEXROWS)%CORTEXROWS,E,D,S,W); //8
   ConfigGlialSynapse(i,(j-1+CORTEXROWS)%CORTEXROWS,W,N,16);
```

```
    //set the axons to get the 10 first axonal inputs of the cortex connected to the 10 synapse  and first output from the used soma
    ConfigGlialAxon(0,0,0,0,W,N);
    ConfigGlialAxon(0,1,W,0,N,0);
    ConfigGlialAxon(0,2,W,S,N,0);
    ConfigGlialAxon(0,3,0,S,N,0);
    ConfigGlialAxon(0,4,0,0,E,0);
    ConfigGlialAxon(0,5,0,0,0,0);
    ConfigGlialAxon(0,6,0,N,0,0);
    ConfigGlialAxon(0,7,0,N,W,0);
    ConfigGlialAxon(0,8,0,0,W,0);
    ConfigGlialAxon(0,9,W,0,0,0);
    ConfigGlialAxon(0,10,0,S,0,0);
    ConfigGlialAxon(0,(-1+CORTEXROWS),0,N,0,0);

    ConfigGlialAxon(1,2,W,0,0,0);
    ConfigGlialAxon(1,3,0,S,0,0);
    ConfigGlialAxon(1,6,0,N,0,0);
    ConfigGlialAxon(1,7,0,0,W,0);
    ConfigGlialAxon(1,10,0,W,0,0);
    ConfigGlialAxon(1,(-1+CORTEXROWS),0,W,0,0);

    ConfigGlialAxon(2,0,S,0,0,0);
    ConfigGlialAxon(2,1,S,0,0,0);
    ConfigGlialAxon(2,2,S,0,0,0);
    ConfigGlialAxon(2,3,S,0,0,0);
    ConfigGlialAxon(2,4,0,0,0,0);
    ConfigGlialAxon(2,5,0,0,0,0);
    ConfigGlialAxon(2,6,0,0,N,0);
    ConfigGlialAxon(2,7,0,0,N,0);
    ConfigGlialAxon(2,8,0,0,N,0);
    ConfigGlialAxon(2,9,0,0,N,0);
    ConfigGlialAxon(2,10,0,0,W,0);
    ConfigGlialAxon(2,(-1+CORTEXROWS),W,0,0,0);

    //char Params[8] =   {-12,-12,-3,-3,0,0,12,12};    ConfigSomaParam(i+1,j,-8000,-1,32,Params);//class I
    //char Params[8] =   {-3,-4,-4,-4,3,3,-3,-4}; ConfigSomaParam(i+1,j,-32000,-1,2500,Params);//class II
    //char Params[8] =   {-14,-14,-14,-14,2,2,-2,-2}; ConfigSomaParam(i+1,j,-20000,-1,5,Params);//Tonic spiking
    //char Params[8] =   {-4,-4,-3,-3,1,1,4,4}; ConfigSomaParam(i+1,j,00,0,2048,Params);//Spike latency
    //char Params[8] =   {-4,-3,-3,-3,1,1,4,3}; ConfigSomaParam(i+1,j,00,0,2800,Params);//Integrator
    char Params[8] =   {-4,-4,-3,-3,3,3,4,4}; ConfigSomaParam(i+1,j,100,100,2048,Params); ConfigGlialSynapseWeight(i+2,(j+2)%CORTEXROWS,-1040);// bistability

    EnableCortexClock();
    mydelay(112);
    b=2;
    ResetSpikeCounter(j);

    for(a = 0; a < 1024; a+=4)
    {
    //WriteSpikeOutWord(0,a);
#if 0 //class I and II excitability
    CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0,a);
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
#endif
#if 0 //Tonic spiking
    CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0,516);
    b=1;
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
    b=1;//2 clock each
```

```
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
#endif
#if 0 // spike latency
   CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0,a<10?516:0);
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
   b=1;//2 clock each
   #endif
#if 0 // Integrator
   if(a<800)
   {
   if(a<40)
   {
   CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0, a<12 ?516:0);
   b=1;
   b=0;
   }
   else
   {
   CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0, a<52 ?516:0);
   //b=1;
   b=0;
   }
   }
   else
   {
   if(a<896)
   {
   CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0, a<812 ?516:0);
   b=0;
   }
   else
   {
   CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0, a<908 ?516:0);
   b=1;
   //b=0;
   }
   }
   //b=0;//1 clock
#endif
#if 1 //bistability
   if(a<764)
   {
   if(a<40)
   {
   CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0, a<4 ?516:0);
   b=1;
   b=0;
   }
   else
   {
   CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0, a<4 ?516:0);
   //b=1;
   b=0;
   }
   }
   else
   {
   if(a<768)
   {
   CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0, a<768 ?260:0);
   b=0;
   }
   else
   {
```

```
   CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0, a<768 ?260:0);
   b=1;
   //b=0;
   }
   }
#endif

   b=1;//2 clock each
   //b=0;//1 clock each
   //b=0;//adjusted to 2.24us per loop iteration = (38+18)*40ns which is the update cycle of the soma cell here in this dendritic loop config

   }


   /* if((a=GetSomaResponse(j))!=1)
   {
   xil_printf("\n\rSoma @(%d,%d) with a synapse @(%d,%d). Sending %d spikes in it axon.********\r\n",i+1,j,ii,jj,a);
   return 1;
   }*/

   xil_printf("\rAll  OK.                \r\n",i+1,j);
}
print("\rFinishing code section 3: different parameter behaviours are OK.                              \n\r");
}
#endif


//************************ CODE SECTION 4 ******************************
   //The following code configures a single soma unit at the bottom left (after the left edge column) corner
   // of the cortex with only one synapse and monitor its axon and dendritic signals for soma and synapse verification.
// I used this to debug the soma and synapse cells
#ifdef EN_CODE_SECTION_4  // code section 4 begins here, switch this to comment/uncomment
{
print("\n\rStarting Code section 4: testing basic functionality of the soma unit at the bottom left...\n\r");

ResetAndDisableCortexClock();

   ConfigGlialDend(0,0,0,W,0,D);
   ConfigGlialAxon(0,0,0,0,0,E);
   ConfigGlialSynapse(0,0,W,E,0x7ff0);

   ConfigGlialDend(0,1,0,W,0,E); //extended loopback for monitoring
   ConfigGlialAxon(0,1,0,0,0,0); //no axon routing

   ConfigGlialDend(0,2,0,W,0,E); //extended loopback for monitoring
   ConfigGlialAxon(0,2,0,0,0,0); //no axon routing

   ConfigGlialDend(1,2,0,0,W,S); //extended loopback for monitoring
   ConfigGlialAxon(1,2,0,0,0,0); //no axon routing

   ConfigGlialDend(0,119,0,W,0,E); //extended loopback for monitoring
   ConfigGlialAxon(0,119,0,0,0,0); //no axon routing

   ConfigGlialDend(1,119,W,0,0,N); //extended loopback for monitoring
   ConfigGlialAxon(1,119,0,0,0,0); //no axon routing


   ConfigGlialDend(2,0,0,0,0,W); //immediate loopback
   ConfigGlialAxon(2,0,0,0,0,0); //no axon routing

   ConfigGlialDend(2,1,0,0,0,W); //immediate loopback
   ConfigGlialAxon(2,1,0,0,0,0); //no axon routing

   char Params[8] = //{15,15,15,15,15,15,15,15};
   {-4,-4,-3,-3,3,3,4,4};
   ConfigSomaParam(1,0,-16384,-1,0x0001,Params);//config the soma


   print("waiting...\n\r");
   int waitcounter;
   for(waitcounter=0;waitcounter<5000000;waitcounter++)
    ;
   EnableCortexClock();

   for(waitcounter=0;waitcounter<1000;waitcounter++)
    ;
```

```
   print("sending a presynaptic spike...\n\r");
   CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0,0x1); //send a single presynaptic spike
   for(waitcounter=0;waitcounter<10;waitcounter++)
    ;

   //CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0,0x1); //send another single presynaptic spike
   for(waitcounter=0;waitcounter<1000000;waitcounter++)
      ;
   print("Code section 4 finished successfully: Bottom left soma basic functionality OK.\n\r");

}
#endif    //code section 4 ends here


         print("\r\n---Exiting main---\n\r");

   XCACHE_DISABLE_ICACHE();
   XCACHE_DISABLE_DCACHE();

   return 0;
}

void mydelay(int period)
{
int del;
for(del=0;del<period;del++)
;
}

//This function configs a verification axonal routing starting at i,j 4/1/2010
void ConfigTestAxonalRouting(int i, int j)
{
#ifdef EN_COL_CONFIG

//first col
ClearColBuff(CortexColBuff);
SetGlialColBuff(CortexColBuff,(j-1+CORTEXROWS)%CORTEXROWS,0,0,0,0,0, E,0,0,0,0, 0);
SetGlialColBuff(CortexColBuff,j,0,0,0,0,0, W,0,0,S,0, 0);
SetGlialColBuff(CortexColBuff,(j+1)%CORTEXROWS,0,0,0,0,0, S,0,0,0,0, 0);
SetGlialColBuff(CortexColBuff,(j+2)%CORTEXROWS,0,0,0,0,0, 0,S,0,0,0, 0);
ConfigCortexCol(0,CortexColBuff);

//col 1 to 11
int r;
for(r=1; r<12; r++)
{
ClearColBuff(CortexColBuff);

if(r>0 && r<i+2) //extensions of the axonal loop
{
SetGlialColBuff(CortexColBuff,(j+2)%CORTEXROWS,0,0,0,0,0, 0,W,0,0,0, 0);
SetGlialColBuff(CortexColBuff,(j-1+CORTEXROWS)%CORTEXROWS,0,0,0,0,0, E,0,0,E,0, 0);
}
if(i>0 && r==i){
SetGlialColBuff(CortexColBuff,j,0,0,0,0,0, S,0,E,0,0, 0);
SetGlialColBuff(CortexColBuff,j+1,0,0,0,0,0, S,0,0,0,0, 0);
}
if(i<7)
{
if(r==i+2)
{
SetGlialColBuff(CortexColBuff,(j-1+CORTEXROWS)%CORTEXROWS,0,0,0,0,0, 0,0,0,N,0, 0);
SetGlialColBuff(CortexColBuff,(j  )%CORTEXROWS,0,0,0,0,0, 0,E,N,0,0, 0);
SetGlialColBuff(CortexColBuff,(j+1)%CORTEXROWS,0,0,0,0,0, 0,N,E,0,0, 0);
SetGlialColBuff(CortexColBuff,(j+2)%CORTEXROWS,0,0,0,0,0, W,N,E,0,0, 0);
SetGlialColBuff(CortexColBuff,(j+3)%CORTEXROWS,0,0,0,0,0, 0,S,E,0,0, 0);
}
if(r==i+3)
{
SetGlialColBuff(CortexColBuff,(j+1)%CORTEXROWS,0,0,0,0,0, N,0,W,S,0, 0);
SetGlialColBuff(CortexColBuff,(j+2)%CORTEXROWS,0,0,0,0,0, W,N,E,S,0, 0);
SetGlialColBuff(CortexColBuff,(j  )%CORTEXROWS,0,0,0,0,0, W,0,0,N,0, 0);
SetGlialColBuff(CortexColBuff,(j+3)%CORTEXROWS,0,0,0,0,0, 0,0,S,W,0, 0);
}
if(r==i+4)
SetGlialColBuff(CortexColBuff,(j+2)%CORTEXROWS,0,0,0,0,0, 0,0,0,W,0, 0);
```

```
}
else
{
if(r==i+2)
{
SetGlialColBuff(CortexColBuff,(j-1+CORTEXROWS)%CORTEXROWS,0,0,0,0,0, 0,0,0,N,0, 0);
SetGlialColBuff(CortexColBuff,(j  )%CORTEXROWS,0,0,0,0,0, 0,0,N,0,0, 0);
SetGlialColBuff(CortexColBuff,(j+1)%CORTEXROWS,0,0,0,0,0, 0,0,N,0,0, 0);
SetGlialColBuff(CortexColBuff,(j+2)%CORTEXROWS,0,0,0,0,0, W,0,N,0,0, 0);
SetGlialColBuff(CortexColBuff,(j+3)%CORTEXROWS,0,0,0,0,0, 0,0,S,0,0, 0);
}
}
ConfigCortexCol(r,CortexColBuff);
}//next r

#else
ConfigGlialAxon(0,(j-1+CORTEXROWS)%CORTEXROWS,E,0,0,0);
ConfigGlialAxon(0,j,W,0,0,S);
ConfigGlialAxon(0,(j+1)%CORTEXROWS,S,0,0,0);
ConfigGlialAxon(0,(j+2)%CORTEXROWS,0,S,0,0);

int r;
for(r=1; r<i+2; r++)
{
   ConfigGlialAxon(r,(j+2)%CORTEXROWS,0,W,0,0);
   ConfigGlialAxon(r,(j-1+CORTEXROWS)%CORTEXROWS,E,0,0,E);
}
if(i>0){
ConfigGlialAxon(i,j,S,0,E,0);
ConfigGlialAxon(i,j+1,S,0,0,0);
}
if(i<7)
{
   ConfigGlialAxon(i+2,(j-1+CORTEXROWS)%CORTEXROWS,0,0,0,N);
   ConfigGlialAxon(i+2,(j  )%CORTEXROWS,0,E,N,0);
   ConfigGlialAxon(i+2,(j+1)%CORTEXROWS,0,N,E,0);
   ConfigGlialAxon(i+2,(j+2)%CORTEXROWS,W,N,E,0);
   ConfigGlialAxon(i+2,(j+3)%CORTEXROWS,0,S,E,0);

   ConfigGlialAxon(i+3,(j  )%CORTEXROWS,W,0,0,N);
   ConfigGlialAxon(i+3,(j+1)%CORTEXROWS,N,0,W,S);
   ConfigGlialAxon(i+3,(j+2)%CORTEXROWS,W,N,E,S);
   ConfigGlialAxon(i+3,(j+3)%CORTEXROWS,0,0,S,W);

   ConfigGlialAxon(i+4,(j+2)%CORTEXROWS,0,0,0,W);
}
else
{
   ConfigGlialAxon(i+2,(j-1+CORTEXROWS)%CORTEXROWS,0,0,0,N);
   ConfigGlialAxon(i+2,(j  )%CORTEXROWS,0,0,N,0);
   ConfigGlialAxon(i+2,(j+1)%CORTEXROWS,0,0,N,0);
   ConfigGlialAxon(i+2,(j+2)%CORTEXROWS,W,0,N,0);
   ConfigGlialAxon(i+2,(j+3)%CORTEXROWS,0,0,S,0);
}
#endif

}
inline void WriteSpikeOutWord(int SpikeInIndex, Xuint32 SpikeOutWord)
{
CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,4*(SpikeInIndex/30),SpikeOutWord);
}

void ResetSpikeCounter(int SpikeInIndex)
{
CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 8*(SpikeInIndex/8)); //reset the counter
}

int GetSomaResponse(int SpikeInIndex)
{
Xuint32 a;

a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 8*(SpikeInIndex/8)); //read and lock the rest
if((SpikeInIndex/4)%2==1) //if it is in the second half
a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 4+8*(SpikeInIndex/8)); //read the rest
return (a>>(6*(SpikeInIndex%4))) & 0x3f;
```

```
}


int TestSomaResponse(int SpikeInIndex) //testing soma with 0-64 presynaptic spikes
{
//int b;
Xuint32 a;

a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 8*(SpikeInIndex/8)); //reset the counter

//for (b=0; b<n;b++) //generate n pulses in that spike in
//{
CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,4*(SpikeInIndex/30),0x00000001<<(SpikeInIndex%30));
//}
int waitcounter;
for(waitcounter=0; waitcounter<2000; waitcounter++) //kill some time to make sure that all pulses are received
;
a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 8*(SpikeInIndex/8)); //read and lock the rest
//CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,4*(SpikeInIndex/30),0x00000001<<(SpikeInIndex%30));//send another spike for debuging
if((SpikeInIndex/4)%2==1) //if it is in the second half
a=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 4+8*(SpikeInIndex/8)); //read the rest
return (a>>(6*(SpikeInIndex%4))) & 0x3f;

}



/*
int main()
{
XHwIcap_DeviceWriteFrame()

   XCACHE_ENABLE_ICACHE();
   XCACHE_ENABLE_DCACHE();

   print("---Entering main---\n\r");




   {
      int status;

      print("\r\nRunning UartLiteSelfTestExample() for mdm_0...\r\n");
      status = UartLiteSelfTestExample(XPAR_MDM_0_DEVICE_ID);
      if (status == 0) {
         print("UartLiteSelfTestExample PASSED\r\n");
      }
      else {
         print("UartLiteSelfTestExample FAILED\r\n");
      }
   }


   {
      XStatus status;


      print("\r\n Running HwIcapTestAppExample() for xps_hwicap_0...\r\n");

      status = HwIcapTestAppExample(XPAR_XPS_HWICAP_0_DEVICE_ID);

      if (status == 0) {
         print("HwIcapTestAppExample PASSED\r\n");
      }
      else {
         print("HwIcapTestAppExample FAILED\r\n");
      }
   }

   print("---Exiting main---\n\r");

   XCACHE_DISABLE_ICACHE();
   XCACHE_DISABLE_DCACHE();

   return 0;
```

```
}

*/

#define APATERN 0x44444444
#define NAPATERN 0xbbbbbbbb
#define BPATERN 0x88888888
#define NBPATERN 0x77777777
#define RPATERN 0x22222222
#define OPATMASK 0x00fC0000
//this function processes the commands from the PC and executes them
void ProcessPCLink()
{
u8 Command,LastCommand, Col, CheckSum, AInterval, BInterval, t,f,l;
u8 MinTime, MaxTime;
u8 Timings[50];
int b;
Xuint32 IPatern;
Xuint32 OPatern;
int TimingIndex;
LastCommand = 0;
while(1)
   {
    Command = XUartLite_RecvByte(STDIN_BASEADDRESS);
u8 * p;
    if( Command == 'c') //config
{
Col = XUartLite_RecvByte(STDIN_BASEADDRESS); //col number
/*while (XUartLite_IsReceiveEmpty(STDIN_BASEADDRESS))
;
counter = XUartLite_Recv(&UartLite,(u8 *) CortexColBuff, (unsigned int)720);*/
for(p=(u8*)CortexColBuff; p<((u8*)CortexColBuff+720) ; p++)
(*p)=XUartLite_RecvByte(STDIN_BASEADDRESS);
CheckSum = XUartLite_RecvByte(STDIN_BASEADDRESS);
int i;
u8 * charbuff = (u8 *) CortexColBuff;
for(i=0, CheckSum+=Col+Command ; i<720 ; i++)
{
CheckSum += charbuff[i];
}
if(CheckSum == 0)
{
if(LastCommand!='c')
ResetAndDisableCortexClock();
if(ConfigCortexCol(Col,CortexColBuff)==XST_SUCCESS)
{
xil_printf("o");
//EnableCortexClock();
}
else
xil_printf("e");
}
else
xil_printf("c");
}
if( Command == 's') //simulation
{
CheckSum = Command;

CheckSum += (AInterval = XUartLite_RecvByte(STDIN_BASEADDRESS)); //first spike interval
CheckSum += (BInterval = XUartLite_RecvByte(STDIN_BASEADDRESS)); //second spike interval
CheckSum += XUartLite_RecvByte(STDIN_BASEADDRESS); //checksum
if(CheckSum==0) //no error
{
//ResetAndDisableCortexClock();
EnableCortexClock();
OPatern=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 0); //reset the counter
IPatern=0;
CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0,  RPATERN |((AInterval==0)?APATERN:0) | ((BInterval==0)?BPATERN:0));

b=2;//wait for 2 clock cycles
b=2;//wait for 2 clock cycles
for(t=1; t<255 ; t++)
{
if(t==AInterval)
IPatern |= APATERN;
```

```
else
IPatern &= NAPATERN;

if(t==BInterval)
IPatern |= BPATERN;
else
IPatern &= NBPATERN;

CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0, IPatern);

// b=0;//wait for 1 clock
// b=2;//wait for 2 clock cycles
//    b=1;//2 clock each
//    b=1;//2 clock each
//    b=1;//2 clock each
//    b=1;//2 clock each
OPatern=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0);
if((OPatern /*& OPATMASK*/)!=0)
break;
}
CheckSum = 'o';
XUartLite_SendByte(STDOUT_BASEADDRESS,CheckSum); //sends 'o'
XUartLite_SendByte(STDOUT_BASEADDRESS,t); //sends output spike time t
CheckSum+=t;
CheckSum = (u8)(-(char)CheckSum);
XUartLite_SendByte(STDOUT_BASEADDRESS,CheckSum); //send the checksum
}
else
{
xil_printf("c");// checksum error
}
}//end if command==s
if( Command == 'a') //an automatic series of simulations
{
CheckSum = Command;
CheckSum += (MinTime = XUartLite_RecvByte(STDIN_BASEADDRESS)); //earliest input spike interval
CheckSum += (MaxTime = XUartLite_RecvByte(STDIN_BASEADDRESS)); //latest input spike interval
CheckSum += XUartLite_RecvByte(STDIN_BASEADDRESS); //checksum
if(CheckSum==0) //no error
{
TimingIndex=0;
for(AInterval=MinTime; AInterval<=MaxTime; AInterval++)
{
for(BInterval=MinTime; BInterval<=MaxTime; BInterval++)
{
//ResetAndDisableCortexClock();
EnableCortexClock();
//OPatern=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 0);
IPatern=0;
f=255;
l=255;
CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0,  RPATERN |((AInterval==0)?APATERN:0) | ((BInterval==0)?BPATERN:0));
OPatern=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 0);//reset the counter
OPatern=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 0);//reset the counter

for(t=1; t<255 ; t++)
{

asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");

asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");

asm("nop");
```

```
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");

asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");

asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");

asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");
asm("nop");

if(t==AInterval)
IPatern |= APATERN;
else
IPatern &= NAPATERN;

if(t==BInterval)
IPatern |= BPATERN;
else
IPatern &= NBPATERN;

CORTEXIP_mWriteSlaveReg0(XPAR_CORTEXIP_0_BASEADDR,0, IPatern);
OPatern=CORTEXIP_mReadSlaveReg0(XPAR_CORTEXIP_0_BASEADDR, 0);
if( ((OPatern & OPATMASK) !=0) && t<f)
f=t;
}//next sim
if(f < 10) //if early spike
f=255; //send worst case timing code 255
//else if(l<255 && f==255)
//f=254;
Timings[TimingIndex++]=f;
}//next BInterval
}//next Avterval

CheckSum = 'o';
XUartLite_SendByte(STDOUT_BASEADDRESS,CheckSum); //sends 'o'
for(TimingIndex=0; TimingIndex<50; TimingIndex++)
{
XUartLite_SendByte(STDOUT_BASEADDRESS,Timings[TimingIndex]); //sends output spike time t
CheckSum+=Timings[TimingIndex];
}
CheckSum = (u8)(-(char)CheckSum);
XUartLite_SendByte(STDOUT_BASEADDRESS,CheckSum); //send the checksum
}
```

```
else
{
xil_printf("c");// checksum error
}
}//end if command==a
LastCommand = Command;
   }

}
```

```
/*****************************************************************************
* Filename:         cortex.h
* Version:           2.17.a
* Description:      Header File for reconfiguration of the Cortex
* Author: Hooman Shayani
*****************************************************************************/
#ifndef CORTEX_H_
#define CORTEX_H_


#define N 1
#define E 2
#define S 3
#define W 4
#define D 5

#define CORTEXCOLS 12
#define CORTEXROWS 120

#define YSOMA2SOMA 4 // vertical displacement between somas in a col >2
#define XSOMA2SOMA 3 // horizontal displacement between somas in a row >1
#define IOSOMAGAP 1 // space between iocells and first col of soma cells
#define YSOMASHIFT 0 // vertical shift of the somas in two neighbouring cols


XStatus EnableCortexClock(void);
XStatus DisableCortexClock(void);
XStatus ResetAndDisableCortexClock(void);
XStatus ConfigGlialAxon(int cellx, int celly,char NX, char EX, char SX, char WX);
XStatus ConfigGlialDend(int cellx, int celly,char NX, char EX, char SX, char WX);
XStatus ConfigGlialSynapse(int cellx, int celly,char XI, char DI, Xint32 Weight);
XStatus ConfigGlialSynapseWeight(int cellx, int celly, Xint32 Weight);
XStatus ConfigSomaParam(int cellx, int celly, Xint16 VReset, Xint16 VStart, Xint16 VBias, char *Params);
XStatus ConfigCortexCol(int colx, Xuint16 *Data);
int CortexSomaPlacement(int x , int y, int h);
XStatus ClearColBuff(Xuint16 *ColBuff);
XStatus SetGlialColBuff(Xuint16 *ColBuff,int Glialy , char ND,char ED,char SD,char WD,char DI, char NX,char EX,char SX,char WX,char XI, Xint16 SynWeight);
XStatus SetSomaColBuff(Xuint16 *ColBuff,int Somay , Xint16 V_Reset, Xint16 V_Start, Xint16 V_Bias,  char *Params, char ND, char UED, char LED, char SD, char LWD
#endif /*CORTEX_H_*/
```

```
/*****************************************************************************
* Filename:         cortex.c
* Version:           2.56.a
* Description:      Source File for reconfiguration of the Cortex
* Author: Hooman Shayani
*****************************************************************************/

#include "xbasic_types.h"
#include "xstatus.h"
#include "xparameters.h"
#include "editlut.h"
#include "cortex.h"

extern XHwIcap *icapptr;
extern Xuint32 Buffer[250];

XStatus EnableCortexClock(void)
{
```

```
CORTEXIP_mWriteSlaveReg4(XPAR_CORTEXIP_0_BASEADDR,0,0x01);
return XST_SUCCESS;
}

XStatus DisableCortexClock(void)
{
CORTEXIP_mWriteSlaveReg4(XPAR_CORTEXIP_0_BASEADDR,0,0x00);
return XST_SUCCESS;
}

XStatus ResetAndDisableCortexClock(void)
{
CORTEXIP_mWriteSlaveReg4(XPAR_CORTEXIP_0_BASEADDR,0,0x03);
CORTEXIP_mWriteSlaveReg4(XPAR_CORTEXIP_0_BASEADDR,0,0x02);
CORTEXIP_mWriteSlaveReg4(XPAR_CORTEXIP_0_BASEADDR,0,0x00);
return XST_SUCCESS;
}


XStatus ConfigGlialAxon(int cellx, int celly,char NX, char EX, char SX, char WX)
{
int x,y;
static Xuint32 axondecode[6]={0x00000000,0xAAAAAAAA,0xCCCCCCCC,0xF0F0F0F0,0xFF00FF00,0xFFFF0000};
//   none North East South West DO
//ASSERT(cellx>=0 && cellx<12);
//ASSERT(celly>=0 && celly<120);
x=12+4*cellx+1;
y=celly;

Xuint32 bits[2];
bits[0]=axondecode[NX];bits[1]=axondecode[EX];
if(SetLUT(icapptr,x,y,LUTC,bits)!=XST_SUCCESS)//set LUTMUXX0
return XST_FAILURE;
bits[0]=axondecode[SX];bits[1]=axondecode[WX];
if(SetLUT(icapptr,x,y,LUTD,bits)!=XST_SUCCESS)//set LUTMUXX1
return XST_FAILURE;

return XST_SUCCESS;
}

XStatus ConfigGlialDend(int cellx, int celly,char NX, char EX, char SX, char WX)
{
int x,y;
static Xuint32 denddecode[6]={0x00000000,0xAAAAAAAA,0xCCCCCCCC,0xF0F0F0F0,0xFF00FF00,0xFFFF0000};
//   none North East South West DO
//ASSERT(cellx>=0 && cellx<12);
//ASSERT(celly>=0 && celly<120);
x=12+4*cellx+1;
y=celly;

Xuint32 bits[2];
bits[0]=denddecode[NX];bits[1]=denddecode[EX];
if(SetLUT(icapptr,x,y,0,bits)!=XST_SUCCESS)//set LUTMUXD0
return XST_FAILURE;
bits[0]=denddecode[SX];bits[1]=denddecode[WX];
if(SetLUT(icapptr,x,y,1,bits)!=XST_SUCCESS)//set LUTMUXD1
return XST_FAILURE;

return XST_SUCCESS;
}



XStatus ConfigGlialSynapse(int cellx, int celly,char XI, char DI, Xint32 Weight)
{
int x,y;
static Xuint32 syndecode[5]={0x0000,0xAAAAAAAA,0xCCCCCCCC,0xF0F0F0F0,0xFF00FF00};
//   none North East South West
//ASSERT(cellx>=0 && cellx<12);
//ASSERT(celly>=0 && celly<120);
x=12+4*cellx;
y=celly;

Xuint32 bits[2];
bits[0]=0;bits[1]=syndecode[DI];
SetLUT(icapptr,x,y,2,bits);//set LUTMUXSD
```

```
bits[0]=0;bits[1]=syndecode[XI];
SetLUT(icapptr,x,y,3,bits);//set LUTMUXSX

SetSRR16(icapptr,x,y,0,Weight);//set weight reg

SetSRR16(icapptr,x,y,1,0x8000);//set control reg
//SetSRR16(icapptr,x,y,1,0x0001);//set control reg
return XST_SUCCESS;
}

XStatus ConfigGlialSynapseWeight(int cellx, int celly, Xint32 Weight)
{
int x,y;
x=12+4*cellx;
y=celly;

SetSRR16(icapptr,x,y,0,Weight);//set weight reg
SetSRR16(icapptr,x,y,1,0x8000);//set control reg
return XST_SUCCESS;
}


XStatus ConfigSomaParam(int cellx, int celly, Xint16 VReset, Xint16 VStart, Xint16 VBias, char *Params)
{
int x,y;
int i;
x=12+4*cellx;
y=celly; //calculate slice x and y

Xuint32 bits[2];
Xuint32 a,b;
char c;

//Set all dendritic MUXs to the external loop as the muxs are redundant!
bits[0]=0xCCCCCCCC; bits[1]=0xFF00FF00;
//bits[0]=0xAAAAAAAA; bits[1]=0xF0F0F0F0;
SetLUT(icapptr,x+1,y,LUTB,bits);//set LUTMUX0
SetLUT(icapptr,x+1,y,LUTC,bits);//set LUTMUX1
SetLUT(icapptr,x+1,y,LUTD,bits);//set LUTMUX2

//Set Param LUTs

for(i=0,a=0; i<8 ; i++)
a = (a<<1) | ((Params[i]<0) ? 0x01 : 0);
bits[0]=0; bits[1]=a|(a<<8)|(a<<16)|(a<<24);
SetLUT(icapptr,x+1,y+1,LUTB,bits);//set ParamLUT2

for(i=0,a=0,b=0; i<8 ; i++)
{
c = 15-abs(Params[i]);
a = (a<<1) | ((c & 0x04)? 0x01 : 0);
b = (b<<1) | ((c & 0x08)? 0x01 : 0);
}
bits[0]=a|(a<<8)|(a<<16)|(a<<24); bits[1]=b|(b<<8)|(b<<16)|(b<<24);
SetLUT(icapptr,x+1,y+1,LUTC,bits);//set ParamLUT1

for(i=0,a=0,b=0; i<8 ; i++)
{
c = 15-abs(Params[i]);
a = (a<<1) | ((c & 0x01)? 0x01 : 0);
b = (b<<1) | ((c & 0x02)? 0x01 : 0);
}
bits[0]=a|(a<<8)|(a<<16)|(a<<24); bits[1]=b|(b<<8)|(b<<16)|(b<<24);
SetLUT(icapptr,x+1,y+1,LUTD,bits);//set ParamLUT0

SetSRR32(icapptr,x,y+1,LUTA,(VReset&0xffff)|(((Xint32)VStart)<<16)); //set BuffReg to 0xVRSTVRST
SetSRR16(icapptr,x,y+1,LUTB,0x4000); //set CtrlReg to 0x00000002
SetSRR16(icapptr,x,y+1,LUTC,0x4000); //set TapCtrlReg to 0x00000002
SetSRR16(icapptr,x,y+1,LUTD,0x0000); //set TapReg to 0x00000000

SetSRR16(icapptr,x,y,LUTA,0x0000); //set PaddingReg to 0x00000000
SetSRR32(icapptr,x,y,LUTB,((Xuint32) VBias)<<6); //set BiasReg to VBias
SetSRR32(icapptr,x,y,LUTC,0x00000020); //set BiasCtrlReg to 0x04000000

return XST_SUCCESS;
}
```

```
//This function is a clone of a function with the same name in the Cortex.vhdl and works in the same way
int CortexSomaPlacement(int x , int y, int h)
{
int col = 0;
int ret = 0;

if(x>=IOSOMAGAP) // if has passed the initial gap with iocells
{
col = (x-IOSOMAGAP)/XSOMA2SOMA;
if ((x-IOSOMAGAP) % XSOMA2SOMA == 0 ) //if in the right col
{
if ((y-col*YSOMASHIFT) % YSOMA2SOMA == 0 ) // if it is the lower soma
ret = 1;
if ((y-1-col*YSOMASHIFT) % YSOMA2SOMA == 0 ) // if it is the upper soma
ret= 2;
}
}
// avoid having a soma cell splited by the wrap around line
if  ((y==0 && (y-1-col*YSOMASHIFT % YSOMA2SOMA) == 0) || (y==h-1 && (y-col*YSOMASHIFT % YSOMA2SOMA) == 0) )
// if an upper soma is in y=0   or an lower soma is in y=h-1
ret= 0;
return ret;
}

XStatus ConfigCortexCol(int col, Xuint16 *Data)
{

static Xuint32 denddecode[6]={0x00000000,0xAAAAAAAA,0xCCCCCCCC,0xF0F0F0F0,0xFF00FF00,0xFFFF0000};
//   none North East South West DO

static Xuint32 syndecode[5]={0x0000,0xAAAAAAAA,0xCCCCCCCC,0xF0F0F0F0,0xFF00FF00};
//   none North East South West

int y,yInZone,word;
int somacase;
XStatus Status;
long Bottom;
int ClkZone;
long HClkRow;
long MajorAddr = 0;
long MinorAddr;
u16 * Skips = icapptr->SkipCols;
int colx=6+2*col+1;
Xuint16 Data1,Data2,Data3;
Xuint32 ALUT[2],BLUT[2],CLUT[2],DLUT[2];

while(colx > *(Skips++))
MajorAddr++;
MajorAddr+=colx;

for(ClkZone = 0 ; ClkZone < 6 ; ClkZone++)
{
Bottom = (ClkZone<3)? 1 : 0;
HClkRow = ClkZone+ (Bottom?(CORTEXROWS/20-1)-(ClkZone<<1): 0) - (CORTEXROWS/40);

//Read the left slices of the col
MinorAddr = 32; // left slice

Status = XHwIcap_DeviceRead2Frames(icapptr, Bottom, XHI_FAR_CLB_BLOCK,HClkRow, MajorAddr, MinorAddr+2, &Buffer[82]);
if (Status != XST_SUCCESS) {
printf("DeviceReadFrame failed:%d\n\r",Status);
return XST_FAILURE;
}

Status = XHwIcap_DeviceRead2Frames(icapptr, Bottom, XHI_FAR_CLB_BLOCK,HClkRow, MajorAddr, MinorAddr, Buffer);
if (Status != XST_SUCCESS) {
printf("DeviceReadFrame failed:%d\n\r",Status);
return XST_FAILURE;
}

for(yInZone=0; yInZone< 20; yInZone++)
{
y=yInZone+ClkZone*20;
Data1  = Data[y*3];
Data2  = Data[y*3+1];
```

```
Data3  = Data[y*3+2];
word = 42+yInZone*2+ ((yInZone>9)?1:0);

switch(CortexSomaPlacement(col,y,120))
{
case 0: //a glial cell

DecodeSRR16(ALUT,Data3); //synapse weight reg
Translate(ALUT);

DecodeSRR16(BLUT,0x8000); //synapse control reg
Translate(BLUT);

CLUT[0] = 0;
CLUT[1] = syndecode[ (Data1>>12) & 0x0007 ]; //D
Translate(CLUT);

DLUT[0] = 0;
DLUT[1] = syndecode[ (Data2>>12) & 0x0007 ]; //Dx
Translate(DLUT);
break;

case 1: // a soma cell lower half
DecodeSRR16(ALUT,0x0000); //set PaddingReg to 0x00000000
Translate(ALUT);
DecodeSRR32(BLUT,((Xuint32) Data3)<<6); //set BiasReg to VBias
Translate(BLUT);
DecodeSRR32(CLUT,0x00000020); //set BiasCtrlReg to 0x04000000
Translate(CLUT);
DLUT[0]=0; //empty
DLUT[1]=0;
Translate(DLUT);
break;

case 2: // a soma cell upper half
DecodeSRR32(ALUT,Data[(y-1)*3]|(((Xint32)Data[(y-1)*3+1])<<16)); //set BuffReg to 0xVRSTVRST
Translate(ALUT);
DecodeSRR16(BLUT,0x4000); //set CtrlReg to 0x00000002
Translate(BLUT);
DecodeSRR16(CLUT,0x4000); //set TapCtrlReg to 0x00000002
Translate(CLUT);
DecodeSRR16(DLUT,0x0000); //set TapReg to 0x00000000
Translate(DLUT);
break;

default:
printf("*************Wrong value returned by CortexSomaPlacement()!\n\r");
break;

}

if(Buffer[word]  != ((BLUT[0]<<16) | (ALUT[0] & 0xffff)))
Buffer[word]  = (BLUT[0]<<16) | (ALUT[0] & 0xffff);

if(Buffer[word+1] != ((DLUT[0]<<16) | (CLUT[0] & 0xffff)))
Buffer[word+1] = (DLUT[0]<<16) | (CLUT[0] & 0xffff);

if(Buffer[word+41] != ((BLUT[0] & 0xffff0000) | (ALUT[0]>>16)))
Buffer[word+41] = (BLUT[0] & 0xffff0000) | (ALUT[0]>>16);

if(Buffer[word+42] != ((DLUT[0] & 0xffff0000) | (CLUT[0]>>16)))
Buffer[word+42] = (DLUT[0] & 0xffff0000) | (CLUT[0]>>16);

if(Buffer[word+82] != ((BLUT[1]<<16) | (ALUT[1] & 0xffff)))
Buffer[word+82] = (BLUT[1]<<16) | (ALUT[1] & 0xffff);

if(Buffer[word+83] != ((DLUT[1]<<16) | (CLUT[1] & 0xffff)))
Buffer[word+83] = (DLUT[1]<<16) | (CLUT[1] & 0xffff);

if(Buffer[word+123]!= ((BLUT[1] & 0xffff0000) | (ALUT[1]>>16)))
Buffer[word+123]= (BLUT[1] & 0xffff0000) | (ALUT[1]>>16);

if(Buffer[word+124]!= ((DLUT[1] & 0xffff0000) | (CLUT[1]>>16)))
Buffer[word+124]= (DLUT[1] & 0xffff0000) | (CLUT[1]>>16);
}//next y in zone
//write back
```

```
Status = XHwIcap_DeviceWrite4Frames(icapptr, Bottom, XHI_FAR_CLB_BLOCK,HClkRow, MajorAddr, MinorAddr, Buffer);
if (Status != XST_SUCCESS) {
printf("DeviceWrite4Frame failed:%d\n\r",Status);
return XST_FAILURE;
}

//Read the right slices of the col
MinorAddr = 26; // right slice

Status = XHwIcap_DeviceRead2Frames(icapptr, Bottom, XHI_FAR_CLB_BLOCK,HClkRow, MajorAddr, MinorAddr+2, &Buffer[82]);
if (Status != XST_SUCCESS) {
printf("DeviceReadFrame failed:%d\n\r",Status);
return XST_FAILURE;
}
Status = XHwIcap_DeviceRead2Frames(icapptr, Bottom, XHI_FAR_CLB_BLOCK,HClkRow, MajorAddr, MinorAddr, Buffer);
if (Status != XST_SUCCESS) {
printf("DeviceReadFrame failed:%d\n\r",Status);
return XST_FAILURE;
}

for(yInZone=0; yInZone< 20; yInZone++)
{
y=yInZone+ClkZone*20;
Data1  = Data[y*3];
Data2  = Data[y*3+1];
Data3  = Data[y*3+2];
word = 42+yInZone*2+ ((yInZone>9)?1:0);

switch(somacase=CortexSomaPlacement(col,y,120))
{
case 0: //a glial cell
ALUT[0]=denddecode[Data1 & 0x0007]; //N
ALUT[1]=denddecode[(Data1>>3) & 0x0007]; //E
Translate(ALUT);
BLUT[0]=denddecode[(Data1>>6) & 0x0007]; //S
BLUT[1]=denddecode[(Data1>>9) & 0x0007]; //W
Translate(BLUT);
CLUT[0]=denddecode[Data2 & 0x0007]; //NX
CLUT[1]=denddecode[(Data2>>3) & 0x0007]; //EX
Translate(CLUT);
DLUT[0]=denddecode[(Data2>>6) & 0x0007]; //SX
DLUT[1]=denddecode[(Data2>>9) & 0x0007]; //WX
Translate(DLUT);
break;

case 1: // a soma cell lower half
Data3= Data[(y+1)*3+2];
BLUT[0]=(Data3 & 0x01)? 0xCCCCCCCC : 0xAAAAAAAA;//UEO
BLUT[1]=(Data3 & 0x02)? 0xFF00FF00 : 0xF0F0F0F0;//LEO
CLUT[0]=(Data3 & 0x04)? 0xCCCCCCCC : 0xAAAAAAAA;//SO
CLUT[1]=(Data3 & 0x08)? 0xFF00FF00 : 0xF0F0F0F0;//LWO
DLUT[0]=(Data3 & 0x10)? 0xCCCCCCCC : 0xAAAAAAAA;//UWO
DLUT[1]=(Data3 & 0x20)? 0xFF00FF00 : 0xF0F0F0F0;//DI
Translate(BLUT);
Translate(CLUT);
Translate(DLUT);
break;

case 2: // a soma cell upper half
BLUT[0] = 0; //empty half of the ParamLUT2 (O5) feeds GND line for the soma cell!

Xuint32 DataTemp = (Data3&0xff00) | (Data3>>8);
BLUT[1] = DataTemp | (DataTemp<<16); //ParamLUT2 O6

DataTemp = (Data2&0xff) | ((Data2&0xff)<<8);
CLUT[0] = DataTemp | (DataTemp<<16); //ParamLUT1 O5

DataTemp = (Data2&0xff00) | (Data2>>8);
CLUT[1] = DataTemp | (DataTemp<<16); //ParamLUT1 O6

DataTemp = (Data1&0xff)|((Data1&0xff)<<8);
DLUT[0] = DataTemp | (DataTemp<<16); //ParamLUT0 O5

DataTemp = (Data1&0xff00) | (Data1>>8);
DLUT[1] = DataTemp | (DataTemp<<16); //ParamLUT0 O6
Translate(BLUT);
```

```
Translate(CLUT);
Translate(DLUT);
break;

default:
printf("*************Wrong value returned by CortexSomaPlacement()!\n\r");
break;
}


if(Buffer[word] != ((BLUT[1] & 0xffff0000) | ((somacase!=0)? (Buffer[word]&0xffff):(ALUT[1]>>16))))
Buffer[word] = (BLUT[1] & 0xffff0000) | ((somacase!=0)? (Buffer[word]&0xffff):(ALUT[1]>>16));

if(Buffer[word+1] != ((DLUT[1] & 0xffff0000) | (CLUT[1]>>16)))
Buffer[word+1] = (DLUT[1] & 0xffff0000) | (CLUT[1]>>16);

if(Buffer[word+41] != ((BLUT[1]<<16) | ((somacase!=0)? (Buffer[word+41]&0xffff):(ALUT[1] & 0xffff))))
Buffer[word+41] = (BLUT[1]<<16) | ((somacase!=0)? (Buffer[word+41]&0xffff):(ALUT[1] & 0xffff));

if(Buffer[word+42] != ((DLUT[1]<<16) | (CLUT[1] & 0xffff)))
Buffer[word+42] = (DLUT[1]<<16) | (CLUT[1] & 0xffff);

if(Buffer[word+82] != ((BLUT[0]<<16) | ((somacase!=0)? (Buffer[word+82]&0xffff):(ALUT[0] & 0xffff))))
Buffer[word+82] = (BLUT[0]<<16) | ((somacase!=0)? (Buffer[word+82]&0xffff):(ALUT[0] & 0xffff));

if(Buffer[word+83] != ((DLUT[0]<<16) | (CLUT[0] & 0xffff)))
Buffer[word+83] = (DLUT[0]<<16) | (CLUT[0] & 0xffff);

if(Buffer[word+123]!= ((BLUT[0] & 0xffff0000) | ((somacase!=0)? (Buffer[word+123]&0xffff):(ALUT[0]>>16))))
Buffer[word+123]= (BLUT[0] & 0xffff0000) | ((somacase!=0)? (Buffer[word+123]&0xffff):(ALUT[0]>>16));

if(Buffer[word+124]!= ((DLUT[0] & 0xffff0000) | (CLUT[0]>>16)))
Buffer[word+124]= (DLUT[0] & 0xffff0000) | (CLUT[0]>>16);

}//next yinzone

//write back
Status = XHwIcap_DeviceWrite4Frames(icapptr, Bottom, XHI_FAR_CLB_BLOCK,HClkRow, MajorAddr, MinorAddr, Buffer);
if (Status != XST_SUCCESS) {
printf("DeviceWrite4Frame failed:%d\n\r",Status);
return XST_FAILURE;
}
}//next zone
return XST_SUCCESS;
}



XStatus ClearColBuff(Xuint16 *ColBuff)
{
int i;
for(i=0; i<360 ; i++)
ColBuff[i]=0;
return XST_SUCCESS;
}

XStatus SetGlialColBuff(Xuint16 *ColBuff,int Glialy , char ND,char ED,char SD,char WD,char DI, char NX,char EX,char SX,char WX,char XI, Xint16 SynWeight)
{
ColBuff[Glialy*3] =ND | (ED<<3) | (SD<<6) | (WD<<9) | (DI<<12);
ColBuff[Glialy*3+1] =NX | (EX<<3) | (SX<<6) | (WX<<9) | (XI<<12);
ColBuff[Glialy*3+2] =SynWeight;
return XST_SUCCESS;
}


XStatus SetSomaColBuff(Xuint16 *ColBuff,int Somay , Xint16 V_Reset, Xint16 V_Start, Xint16 V_Bias,  char *Params, char ND, char UED, char LED, char SD, char LWD
{
ColBuff[Somay*3] =V_Reset;
ColBuff[Somay*3+1] =V_Start;
ColBuff[Somay*3+2] =V_Bias;


//Set Param LUTs
int i;
Xuint32 bits[2];
Xuint32 a,b;
```

```
char c;

for(i=0,a=0; i<8 ; i++)
a = (a<<1) | ((Params[i]<0) ? 0x01 : 0);
ColBuff[Somay*3+5] =((Xuint16)(ND|(UED<<1)|(LED<<2)|(SD<<3)|(LWD<<4)|(UWD<<5)))
| (((Xuint16)a)<<8); //set ParamLUT

for(i=0,a=0,b=0; i<8 ; i++)
{
c = 15-abs(Params[i]);
a = (a<<1) | ((c & 0x04)? 0x01 : 0);
b = (b<<1) | ((c & 0x08)? 0x01 : 0);
}
ColBuff[Somay*3+4] = a | (((Xuint16)b)<<8); //set ParamLUT1

for(i=0,a=0,b=0; i<8 ; i++)
{
c = 15-abs(Params[i]);
a = (a<<1) | ((c & 0x01)? 0x01 : 0);
b = (b<<1) | ((c & 0x02)? 0x01 : 0);
}
ColBuff[Somay*3+3] = a | (((Xuint16)b)<<8); //set ParamLUT0
return XST_SUCCESS;
}
```

Appendix D

IO Cell Design

The left column of the cortex contains IO cells that can feed stimuli to the cortex and record the output of the cortex from the axons routed to them. Two types of modules were designed and implemented for these IO cells. The Spike Counter module and Spike Generator module. Both modules make use of the DSP48E blocks that are located right on the edge of the region allocated to the Cortex in the Virtex-5 XC5VLX50T.

The Spike Counters Module

Fifteen of the DSP blocks were configured as 48-bit counters. By grouping the bits into eight groups of six bits and setting all the bits to zero logic except the lowest significant bit in each group that is connected to an axonal output from the Cortex, eight 6-bit spike counters were constructed. This way fifteen DSP blocks were used to construct a total of 120 6-bit counters for all the Cortex axonal outputs. Output registers of the DSP blocks were mapped to the embedded system memory address. Every time that a DSP output register is read by the processor, 32 bits of the data is read and the rest is stored in another register to be read by processor in the next instruction. Reading the output register also resets the output register, preparing it for the next round of spike counting. Figure D.1 shows the block diagram of the Spike Counter module with configuration of the DSP block as eight 6-bit counters.

The Spike Generator Module

Another DSP block is used to generate the simulation clock which shows a duration corresponding to 1ms of biological neuron simulation. This DSP is also configured as a countdown counter that resets itself to a value stored in a register when it reaches zero and send a single pulse in the output. The value controls the period of the simulation clock cycle. This value can be written to the register by the processor. This signal can be used as an interrupt to the processor for performing input/output operations in every simulation cycle.

It was possible to use the rest of the available DSP blocks for spike generators that independently generate a spike after a programmable period of time to allow the processor simply encode the stimuli in spike rates. However, for simplicity this was left to the processor to writes all the spikes to a set of registers connected to the axonal inputs of the Cortex in an interrupt routine or a loop (in every simulation cycle). These registers were memory mapped and accessible to the embedded processor by a simple memory write instruction. After each write operation these registers automatically clear so that

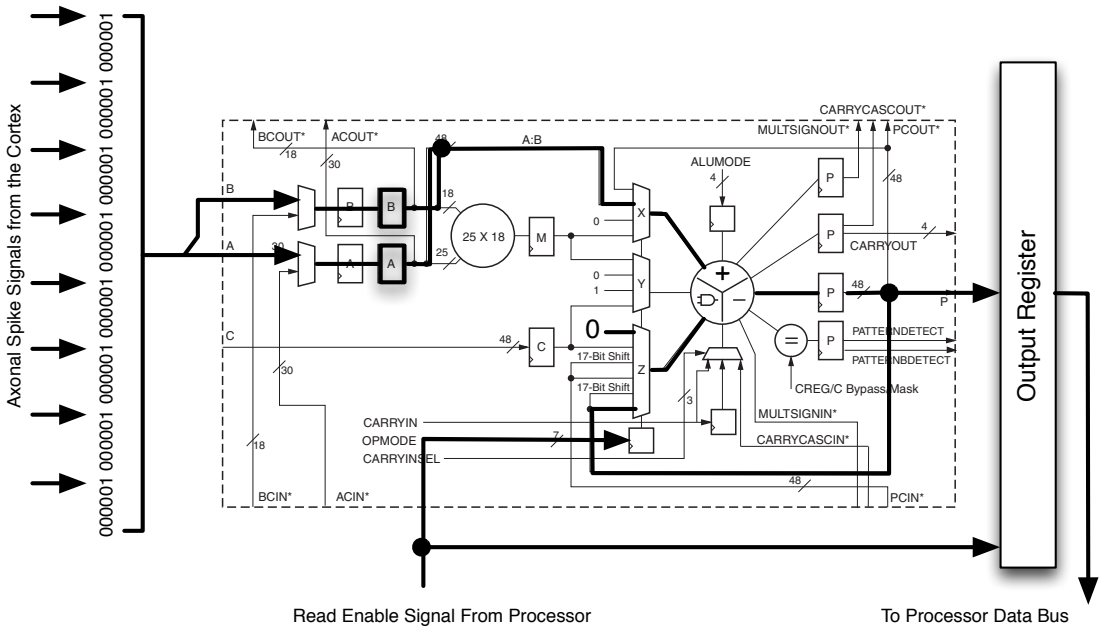


Figure D.1: Block diagram of the Spike Counter module with configuration of the DSP block as eight 6-bit counters.

they only generate a spike with pulse width of one Cortex clock cycle.

Bibliography

[1] A. M. Ahmad, G. M. Khan, S. A. Mahmud, and J. F. Miller. Breast cancer detection using cartesian genetic programming evolved artificial neural networks. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference - GECCO '12*, page 1031, New York, New York, USA, 2012. ACM Press.

[2] A. Ahmadi and M. Zwolinski. A Modified Izhikevich Model For Circuit Implementation of Spiking Neural Networks. In *LASCAS 2010: IEEE Latin American Symposium on Circuit and system*, Brasil, 2010.

[3] S. Z. Ahmed, G. Sassatelli, L. Torres, and L. Rouge. Survey of New Trends in Industry for Programmable Hardware: FPGAs, MPPAs, MPSoCs, Structured ASICs, eFPGAs and New Wave of Innovation in FPGAs. *2010 International Conference on Field Programmable Logic and Applications*, 1:291–297, Aug. 2010.

[4] A. Alaghi and J. P. Hayes. Survey of Stochastic Computing. *ACM Transactions on Embedded Computing Systems*, 12(2s):1–19, May 2013.

[5] E. Alba, G. Luque, C. A. C. Coello, and E. H. Luna. A Comparative Study of Serial and Parallel Heuristics Used to Design Combinational Logic Circuits. *Optimization Methods and Software*, 22:485–509, 2007.

[6] D. Allen, D. M. Halliday, and A. M. Tyrrell. A Hybrid Bio-inspired System: Hardware Spiking Neural Network Incorporating Hebbian Learning with Microprocessor Based Evolutionary Control Algorithm. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 2958–2965, 2006.

[7] Altera Corporation. Cyclone Device Handbook , Volume 1. Technical report, Altera Corporation, 2008.

[8] Altera Corporation. Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs. Technical report, Altera Corporation, 2010.

[9] Altera Corporation. Stratix II Device Handbook , Volume 1. Technical report, Altera Corporation, 2011.

[10] J. C. Astor and C. Adami. A Developmental Model for the Evolution of Artificial Neural Networks. *Artificial Life*, 6(3):189–218, July 2000.

[11] Atmel. 5K - 50K Gates Coprocessor FPGA with FreeRAM. Technical report, Atmel, 2006.

[12] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, New York, 1996.

[13] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 1978.

[14] S. L. Bade and B. L. Hutchings. FPGA-based stochastic neural networks-implementation. In *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pages 189–198, 1994.

[15] O. Bandman. Comparative study of cellular-automata diffusion models. *Parallel Computing Technologies*, pages 756–756, 1999.

[16] W. Banzhaf, G. Beslon, S. Christensen, J. A. Foster, F. Képès, V. Lefort, J. F. Miller, M. Radman, J. J. Ramsden, and Others. From artificial evolution to computational evolution: a research agenda. *Nature Reviews Genetics*, 7:729–735, 2006.

[17] W. Banzhaf and J. Miller. The challenge of complexity. *Frontiers of Evolutionary Computation*, 11:243–260, 2004.

[18] E. Basham and D. Parent. Compact digital implementation of a quadratic integrate-and-fire neuron. In *Engineering in Medicine and Biology Society*, pages 3543–3548, 2012.

[19] D. S. Bassett, D. L. Greenfield, A. Meyer-Lindenberg, D. R. Weinberger, S. W. Moore, and E. T. Bullmore. Efficient physical embedding of topologically complex information processing networks in brains and computer circuits. *PLoS computational biology*, 6(4):e1000748, Apr. 2010.

[20] T. Becker, W. Luk, and P. Y. K. Cheung. Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration. *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*, pages 35–44, Apr. 2007.

[21] S. Bellis, K. M. Razeeb, C. Saha, K. Delaney, C. O'Mathuna, A. Pounds-Cornish, G. de Souza, M. Colley, H. Hagras, G. Clarke, and Others. FPGA implementation of spiking neural networks-an initial step towards building tangible collaborative autonomous agents. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pages 449–452, 2004.

[22] P. Bentley. Everything Computes. In *Proceedings of Third Iteration, the Third International Conference on Generative Arts*, pages 15–24, 2005.

[23] P. Bentley and S. Kumar. Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *Proceedings of the Genetic and Evolutionary Conmputation Conference (GECCO 1999)*, pages 35–43, 1999.

[24] P. J. Bentley. Evolving beyond perfection: an investigation of the effects of long-term evolution on fractal gene regulatory networks. *Biosystems*, 76(1-3):291–301, 2004.

[25] P. J. Bentley. Fractal Proteins. *Genetic Programming and Evolvable Machines*, 5(1):71–101, 2004.

[26] P. J. Bentley. Controlling Robots with Fractal Gene Regulatory Networks. In L. de Castro and F. von Zuben, editors, *Recent Developments in Biologically Inspired Computing*, chapter 13, page 320. Idea Group Inc, 2005.

[27] P. J. Bentley. Investigations Into Graceful Degradation of Evolutionary Developmental Software. *Natural Computing*, 4(4):417–437, 2005.

[28] F. Benz, A. Seffrin, and S. A. Huss. Bil: A tool-chain for bitstream reverse-engineering. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 735–738. IEEE, Aug. 2012.

[29] E. Bergeron, M. Feeley, M.-A. Daigneault, and J. P. David. Using dynamic reconfiguration to implement high-resolution programmable delays on an FPGA. *2008 Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*, pages 265–268, June 2008.

[30] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Springer, 1999.

[31] S. Bhandari, F. Cancare, and D. Bartolini. On the Management of Dynamic Partial Reconfiguration to Speed-up Intrinsic Evolvable Hardware Systems. In *6th HiPEAC Workshop on Reconfigurable Computing (WRC)*, pages 1–10, 2012.

[32] M. Bhuiyan. Optimization and performance study of large-scale biological networks for reconfigurable computing. In *High-Performance Reconfigurable Computing Technology and Applications (HPRCTA), 2010 Fourth International Workshop on*, 2010.

[33] M. A. Bhuiyan, V. K. Pallipuram, and M. C. Smith. Acceleration of spiking neural networks in emerging multi-core and GPU architectures. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, Apr. 2010.

[34] M. A. Bhuiyan, M. C. Smith, and V. K. Pallipuram. Performance, optimization, and fitness: Connecting applications to architectures. *Concurrency and Computation: Practice and Experience*, 23(10):1066–1100, July 2011.

[35] G. Q. Bi. Spatiotemporal specificity of synaptic plasticity: cellular rules and mechanisms. *Biological Cybernetics*, 87(5):319–332, 2002.

[36] J. Biethahn and V. Nissen, editors. *Evolutionary Algorithms in Management Applications*, Berlin, 1995. Springer Verlag.

[37] C. M. Bishop. *Neural Networks for Pattern Recogntion*. Oxford University Press, 1995.

[38] H. Blume, H. T. Feldkaemper, and T. G. Noll. Model-Based Exploration of the Design Space for Heterogeneous Systems on Chip. *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 40(1):19–34, May 2005.

[39] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D. U. Hwang. Complex networks: Structure and dynamics. *Physics Reports*, 424(4-5):175–308, 2006.

[40] E. J. W. Boers and I. G. Sprinkhuizen-kuyper. Combined Biological Metaphors. In M. Patel, V. Honavar, and K. Balakrishnan, editors, *Advances in the Evolutionary Synthesis of Intelligent Agents*, pages 153–183. MIT Press Cambridge, MA, USA, 2001.

[41] S. Bohte and J. Kok. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48:17–37, 2002.

[42] S. M. Bohte. *Spiking neural networks*. PhD thesis, Leiden University, 2003.

[43] S. M. Bohte. The evidence for neural information processing with precise spike-times: A survey. *Natural Computing*, 3(2):195–206, 2004.

[44] S. M. Bohte and J. N. Kok. Applications of spiking neural networks. *Information Processing Letters*, 95(6):519–520, Sept. 2005.

[45] S. M. Bohte, L. H. Poutre, and J. N. Kok. Unsupervised clustering with spiking neurons by sparse temporal coding and multilayer RBF networks. *IEEE-EC*, 13(2):426–435, Mar. 2002.

[46] H. Bolouri and E. H. Davidson. Modeling transcriptional regulatory networks. *BioEssays : news and reviews in molecular, cellular and developmental biology*, 24(12):1118–29, Dec. 2002.

[47] A. Bonetto, A. Cazzaniga, G. Durelli, C. Pilato, D. Sciuto, and M. D. Santambrogio. An open-source design and validation platform for reconfigurable systems. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 707–710, 2012.

[48] J. Bongard and R. Pfeifer. Evolving Complete Agents using Artificial Ontogeny. In *Morpho-Functional Machines: The New Species (Designing Embodied Intelligence)*, pages 237–258. Springer-Verlag, Berlin, 2003.

[49] D. Braendler, T. Hendtlass, and P. O'Donoghue. Deterministic bit-stream digital neurons. *IEEE transactions on neural networks*, 13(6):1514–25, Jan. 2002.

[50] B. Brown and H. Card. Stochastic Neural Computation I: Computational Elements. *IEEETC: IEEE Transactions on Computers*, 50:891–905, 2001.

[51] E. Bullmore and O. Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10:186–198, 2009.

[52] H. H. Bulthoff. *Biologically Motivated Computer Vision*. Springer, 2003.

[53] D. V. Buonomano and M. M. Merzenich. Temporal Information Transformed into a Spatial Code by a Neural Network with Realistic Properties. *Science*, 267(5200):1028–1030, Feb. 1995.

[54] F. Cancare, S. Bhandari, and D. Bartolini. A bird's eye view of FPGA-based Evolvable Hardware. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, 2011.

[55] A. Cangelosi, S. Nolfi, and D. Parisi. Artificial life models of neural development. In S. Kumar and P. J. Bentley, editors, *On Growth, Form and Computers*, chapter 18, pages 339–352. Academic Press, 2003.

[56] A. Cangelosi, D. Parisi, and S. Nolfi. Cell division and migration in a genotype for neural networks (Cell division and migration in neural networks). Technical report, Institute of Psychology - CNR, Rome - Italy, 1994.

[57] J. M. P. Cardoso and M. Hübner, editors. *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*. Springer, 2011.

[58] S. Carrillo, J. Harkin, L. McDaid, and S. Pande. An efficient, high-throughput adaptive NoC router for large scale spiking neural network hardware implementations. *Evolvable Systems: From Biology to Hardware*, 6274/2010:133–144, 2010.

[59] S. Carrillo, J. Harkin, L. McDaid, S. Pande, S. Cawley, B. McGinley, and F. Morgan. Advancing interconnect density for spiking neural network hardware implementations using traffic-aware adaptive network-on-chip routers. *Neural networks : the official journal of the International Neural Network Society*, 33C:42–57, Apr. 2012.

[60] S. Carrillo, J. Harkin, L. McDaid, S. Pande, S. Cawley, and F. Morgan. Adaptive routing strategies for large scale spiking neural network hardware implementations. In *Neural Networks and Machine Learning - ICANN*, pages 77–84, 2011.

[61] A. Cassidy, S. Denham, P. Kanold, and A. Andreou. FPGA Based Silicon Spiking Neural Array. In *2007 IEEE Biomedical Circuits and Systems Conference*, pages 75–78. IEEE, Nov. 2007.

[62] S. Cawley, F. Morgan, B. McGinley, S. Pande, L. McDaid, S. Carrillo, and J. Harkin. Hardware spiking neural network prototyping and application. *Genetic Programming and Evolvable Machines*, 12(3):257–280, Apr. 2011.

[63] K. C. Chatzidimitriou and P. A. Mitkas. A NEAT Way for Evolving Echo State Networks. *Proceedings of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 909–914, Aug. 2010.

[64] D. B. Chklovskii. Synaptic Connectivity and Neuronal Morphology: Two Sides of the Same Coin. *Neuron*, 43:609–617, 2004.

[65] D. B. Chklovskii, B. W. Mel, and K. Svoboda. Cortical rewiring and information storage. *Nature*, 431:782–788, 2004.

[66] D. B. Chklovskii, T. Schikorski, and C. F. Stevens. Wiring Optimization in Cortical Circuits. *Neuron*, 34:341–347, 2002.

[67] C. Chotard and I. Salecker. Neurons and glia: team players in axon guidance. *Trends in neurosciences*, 27(11):655–61, Nov. 2004.

[68] C. A. Coello Coello and G. B. Lamont, editors. *Applications of Multi-Objective Evolutionary Algorithms*. World Scientific, Singapore, 2004.

[69] J. Cosp and J. Madrenas. Scene segmentation using neuromorphic oscillatory networks. *Neural Networks, IEEE Transactions on*, 14(5):1278–1296, Jan. 2003.

[70] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines: And Other Kernel-Based Learning Methods*. Cambridge University Press, 2000.

[71] E. Damiani, V. Liberali, and A. Tettamanzi. Dynamic Optimisation of Non-linear Feed Forward Circuits. In J. F. Miller, A. Thompson, P. Thomson, and T. C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware, Third International Conference, ICES 2000, Edinburgh, Scotland, UK, April 17-19, 2000, Proceedings*, volume 1801 of *Lecture Notes in Computer Science*, pages 41–50. Springer, 2000.

[72] S. Davies, J. Navaridas, F. Galluppi, and S. Furber. Population-based routing in the SpiNNaker neuromorphic architecture. *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, June 2012.

[73] P. Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems (Computational Neuroscience)*. The MIT Press, 2005.

[74] H. de Garis, L. de Penning, A. Buller, and D. Decesare. Early experiments on the CAM-Brain Machine (CBM). *Proceedings Third NASA/DoD Workshop on Evolvable Hardware. EH-2001*, pages 211–219, 2001.

[75] Defence Sciences Office. Systems of Neuromorphic Adaptive Plastic Scalable Electronics (SYNAPSE), 2011.

[76] F. Dellaert and R. Beer. A developmental model for the evolution of complete autonomous agents. In *From animals to animats IV*, pages 393–401. MIT Press Cambridge, MA, 1996.

[77] J. Delorme, A. Nafkha, P. Leray, and C. Moy. New opbhwicap interface for realtime partial reconfiguration of FPGA. *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on*, pages 386–391, 2009.

[78] A. Devert. When and why development is needed. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation - GECCO '09*, page 1843, New York, New York, USA, 2009. ACM Press.

[79] A. Devert and N. Bredeche. Unsupervised learning of echo state networks: A case study in artificial embryogeny. *Artificial Evolution*, 2008.

[80] A. Devert, N. Bredeche, and M. Schoenauer. Robustness and the Halting Problem for Multicellular Artificial Ontogeny. *IEEE Transactions on Evolutionary Computation*, 15(3):387–404, June 2011.

[81] E. A. Di Paolo. Evolving spike-timing-dependent plasticity for single-trial learning in robots. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 361(1811):2299–2319, 2003.

[82] J. Dickson and R. McLeod. Stochastic arithmetic implementations of neural networks with in situ learning. *Neural Networks, 1993.,*, 1993.

[83] Z. Ding, Q. Wu, Y. Zhang, L. Zhu, W. Shu, and M.-Y. Wu. Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation. *Microprocessors and Microsystems*, Jan. 2013.

[84] P. F. Dominey. Complex sensory-motor sequence learning based on recurrent state representation and reinforcement learning. *Biological Cybernetics*, 267:265–274, 1995.

[85] F. Duhem, F. Muller, and P. Lorenzini. FaRM: Fast Reconfiguration Manager for Reducing Reconfiguration Time Overhead on FPGA. In A. Koch, R. Krishnamurthy, J. McAllister, R. Woods, and T. El-Ghazawi, editors, *Reconfigurable Computing: Architectures, Tools and Applications SE - 26*, volume 6578 of *Lecture Notes in Computer Science*, pages 253–260. Springer Berlin Heidelberg, 2011.

[86] P. Dürr, C. Mattiussi, and D. Floreano. Neuroevolution with analog genetic encoding. In T. P. Runarsson, H.-G. Beyer, E. Burke, J. J. Merelo-Guervós, L. D. Whitley, and X. Yao, editors, *Parallel Problem Solving from Nature - PPSN IX*, volume 4193 of *Lecture Notes in Computer Science*, pages 671–680. Springer Berlin Heidelberg, Berlin, Heidelberg, Sept. 2006.

[87] P. Eggenberger. Creation of neural networks based on developmental and evolutionary principles. *Artificial Neural NetworksâĂŤICANN'97*, 1327:337–342, 1997.

[88] P. Eggenberger. Evolving morphologies of simulated 3D organisms based on differential gene expression. In *Proceedings of the Fourth European Conference on Artificial Life, ECAL97*, pages 205–213. MIT Press Cambridge, MA, 1997.

[89] P. Eggenberger. Combining developmental processes and their physics in an artificial evolutionary system to evolve shapes. In *On Growth, Form and Computers*. Academic Press, 2003.

[90] M. Eisenring and L. Thiele. Conflicting criteria in embedded system design. *Design & Test of Computers, IEEE*, 17(2):51–59, 2000.

[91] J. G. Elias and David P.M. Northmore. Building Silicon Nervous Systems with Dendritic Tree Neuromorphs. In *Pulsed neural networks*, pages 135–156. MIT Press Cambridge, MA, USA, 1999.

[92] D. M. Ellin and S. J. Flockton. Analysing evolvable cell design for optimisation of routing options. *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation - GECCO '07*, page 2687, 2007.

[93] D. M. Ellin and S. J. Flockton. Resource-efficient GA Repair of Cell-based Electronic Circuits. In *Proceedings of the 2008 UK Workshop on Computational Intelligence*, pages 171–176, 2008.

[94] J. Eriksson, O. Torres, A. Mitchell, and G. Tucker. Spiking neural networks for reconfigurable POEtic tissue. *Proceedings of the 5th*, pages 165–173, 2003.

[95] D. Federici. A Regenerating Spiking Neural Network. *Neural Networks*, 18(5-6):746–754, 2005.

[96] J. A. Feldman. On intelligence as memory. *Artificial Intelligence*, 169(2):181–183, Dec. 2005.

[97] C. Fernando and S. Sojakka. Pattern Recognition in a Bucket. In *Proc. of ECAL*, pages 588–597. Springer, 2003.

[98] A. Ferrari. System design: Traditional concepts and new paradigms. In *Computer Design VLSI in Computers and Processors*, 1999.

[99] A. K. Fidjeland, E. B. Roesch, M. P. Shanahan, and W. Luk. NeMo: A Platform for Neural Modelling of Spiking Neurons Using GPUs. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 137–144. IEEE, July 2009.

[100] J. Fieres, J. Schemmel, and K. Meier. Realizing biological spiking network models in a configurable wafer-scale hardware system. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 969–976. IEEE, June 2008.

[101] D. Floreano, P. Dürr, and C. Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, Jan. 2008.

[102] D. Floreano, Y. Epars, J.-c. C. Zufferey, and C. Mattiussi. Evolution of Spiking Neural Circuits in Autonomous Mobile Robots. *International Journal of Intelligent Systems*, 20:100, 2005.

[103] D. Floreano and C. Mattiussi. *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. The MIT Press, 2008.

[104] D. Floreano and F. Mondada. Evolutionary neurocontrollers for autonomous mobile robots. *Neural networks : the official journal of the International Neural Network Society*, 11(7-8):1461–1478, Oct. 1998.

[105] D. Floreano, N. Schoeni, G. Caprari, and J. Blynel. Evolutionary Bits'n'Spikes. *Artificial life VIII*, pages 335–344, Dec. 2002.

[106] R. V. Florian. Biologically inspired neural networks for the control of embodied agents. Technical report, Center of Cognitive and Neural studies, 2003.

[107] D. B. Fogel. Evolutionary Algorithms in Engineering Applications. *IEEE Transactions on Evolutionary Computation*, 2(2):72, July 1998.

[108] N. Forbes. *Imitation of life: how biology is inspiring computing*. MIT Press, 2004.

[109] R. L. B. French and R. I. Damper. Evolution of a circuit of spiking neurons for phototaxis in a Braitenberg vehicle. In *From Animals to Animats 7: Proceedings of the Seventh International Conference on Simulation of Adaptive Behavior*, 2002.

[110] K.-i. Funahashi and Y. Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Netw.*, 6(6):801–806, 1993.

[111] S. Furber. Biologically-inspired massively-parallel architectures-computing beyond a million processors. *Application of Concurrency to System Design*, 2009.

[112] S. Furber and S. Temple. On-chip and inter-chip networks for modeling large-scale neural systems. *Systems, 2006. ISCAS 2006.*, 2006.

[113] S. Furber and S. Temple. Neural systems engineering. *Journal of The Royal Society Interface*, 4(13):193–206, Apr. 2007.

[114] S. Furber, S. Temple, and A. Brown. High-Performance Computing for Systems of Spiking Neurons. *AISB'06 workshop on GC5: Architecture of Brain and Mind*, 2006.

[115] F. Gagliardi. Some Issues About Cognitive Modelling and Functionalism. *AI* IA 2007: Artificial Intelligence and Human-Oriented Computing*, pages 60–71, 2007.

[116] B. R. Gaines. *Stochastic Computing Systems*, volume 2, chapter 2, pages 37–172. New York: Plenum, 1969.

[117] J. Gauci and K. O. Stanley. Autonomous Evolution of Topographic Regularities. *Neural Computation*, 22:1860–1898, 2010.

[118] D. George. NuPic, (www.numenta.com/archives/software.php), 2008.

[119] D. George and J. Hawkins. Towards a mathematical theory of cortical micro-circuits. *PLoS computational biology*, 5(10):e1000532, Oct. 2009.

[120] F. Gers, H. de Garis, and M. Korkin. CoDi-1Bit: A simplified cellular automata based neuron model. *Artificial Evolution*, 1363/1998:315–333, 1998.

[121] W. Gerstner and W. M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[122] B. Glackin, J. Harkin, T. M. McGinnity, L. P. Maguire, and Q. Wu. Emulating Spiking Neural Networks for edge detection on FPGA hardware. In *2009 International Conference on Field Programmable Logic and Applications*, pages 670–673. IEEE, Aug. 2009.

[123] B. Glackin, L. Maguire, T. McGinnity, A. Belatreche, and Q. Wu. Implementation of a biologically realistic spiking neuron model on FPGA hardware. *Proc. of The 8th Joint Conference on Information Sciences*, 1-3:1412–1415, 2005.

[124] B. Glackin, T. M. McGinnity, L. P. Maguire, Q. X. Wu, and A. Belatreche. A Novel Approach for the Implementation of Large Scale Spiking Neural Networks on FPGA Hardware. *Lecture notes in computer science*, 3512:552–563, 2005.

[125] N. Gockel, R. Drechsler, and B. Becker. A multi-layer detailed routing approach based on evolutionaryalgorithms. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 557–562, 1997.

[126] K. Goossens, M. Bennebroek, J. Y. Hur, M. A. Wahlah, and N. X. P. Semiconductors. Hardwired Networks on Chip in FPGAs to Unify Functional and Configuration Interconnects. In *Second ACM/IEEE International Symposium on Networks-on-Chip (NOCS 2008)*, pages 45–54. IEEE, Apr. 2008.

[127] T. Gordon and P. J. Bentley. Evolving Hardware. In *Handbook of Nature Inspired and Innovative Computing*, pages 387–432. Springer, 2006.

[128] T. G. W. Gordon and P. J. Bentley. Bias and scalability in evolutionary development. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 83–90. ACM Press New York, NY, USA, 2005.

[129] T. G. W. Gordon and P. J. Bentley. Development brings scalability to hardware evolution. In *Evolvable Hardware, 2005. Proceedings. 2005 NASA/DoD Conference on*, pages 272–279, 2005.

[130] E. L. Graas, E. A. Brown, and R. H. Lee. An FPGA-based approach to high-speed simulation of conductance-based neuron models. *Neuroinformatics*, 2(4):417–36, Jan. 2004.

[131] G. W. Greenwood and A. M. Tyrrell. *INTRODUCTION TO EVOLVABLE HARDWARE A Practical Guide for Designing Self-Adaptive Systems*. Wiley-IEEE Press, 2006.

[132] F. Gruau. Automatic definition of modular neural networks. *Adaptive behavior*, 3(2):151–183, 1994.

[133] F. Gruau. Genetic micro programming of neural networks. *Advances in Genetic Programming*, pages 495–518, Aug. 1994.

[134] R. Guerrero-Rivera and T. C. Pearce. Attractor-Based Pattern Classification in a Spiking FPGA Implementation of the Olfactory Bulb. In *2007 3rd International IEEE/EMBS Conference on Neural Engineering*, pages 593–599. IEEE, May 2007.

[135] I. Gupta. On the design of distributed protocols from differential equations. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 216–225. ACM Press New York, NY, USA, 2004.

[136] I. Gupta, M. Nagda, and C. F. Devaraj. The design of novel distributed protocols from differential equations. *Distributed Computing*, 20(2):95–114, May 2007.

[137] P. Haddow and A. Tyrrell. Challenges of evolvable hardware: past, present and the path to a promising future. *Genetic Programming and Evolvable Machines*, 2011.

[138] P. C. Haddow and G. Tufte. An evolvable hardware FPGA for adaptive hardware. *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, 1:553–560, 2000.

[139] P. C. Haddow and G. Tufte. Bridging the genotype-phenotype mapping for digital FPGAs. In *Evolvable Hardware, 2001. Proceedings. The Third NASA/DoD Workshop on*, pages 109–115, 2001.

[140] M. D. Haene, B. Schrauwen, and D. Stroobandt. Accelerating Event Based Simulation for Multi-synapse Spiking Neural Networks. *Artificial Neural Networks - ICANN 2006*, 4131:760–769, 2006.

[141] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann. A design methodology for communication infrastructures on partially reconfigurable FPGAs. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007.

[142] A. N. Hampton and C. Adami. Evolution of Robust Developmental Neural Networks. In *Artificial Life IX: Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems*, 2004.

[143] B. Han and T. M. Taha. Neuromorphic models on a GPGPU cluster. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, July 2010.

[144] S. G. Hansen, D. Koch, and J. Torresen. High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro. In *2011 IEEE International Symposium on Parallel and Distributed Processing*, pages 174–180. Ieee, May 2011.

[145] S. Harding and J. F. Miller. Evolution In Materio: A Real-Time Robot Controller in Liquid Crystal. In *Evolvable Hardware*, pages 229–238. IEEE Computer Society, 2005.

[146] S. Harding, J. F. Miller, and W. Banzhaf. Developments in Cartesian Genetic Programming: self-modifying CGP. *Genetic Programming and Evolvable Machines*, 11(3-4):397–439, June 2010.

[147] J. Harkin, F. Morgan, L. McDaid, S. Hall, B. McGinley, and S. Cawley. A Reconfigurable and Biologically Inspired Paradigm for Computation Using Network-On-Chip and Spiking Neural Networks. *International Journal of Reconfigurable Computing*, 2009:1–13, 2009.

[148] S. A. Harp, T. Samad, and A. Guha. Designing application-specific neural networks using the genetic algorithm. In *Advances in neural information processing systems 2*, pages 447–454. Morgan Kaufmann, June 1990.

[149] M. Hausser and B. Mel. Dendrites: bug or feature? *Current Opinion in Neurobiology*, 13(3):372–383, June 2003.

[150] J. Hawkins and S. Blakeslee. *On Intelligence*. St. Martin's Griffin, 2005.

[151] J. Hawkins and D. George. Hierarchical Temporal Memory. Technical report, Numenta, Inc., 2006.

[152] H. H. Hellmich, M. Geike, P. Griep, P. Mahr, M. Rafanelli, and H. Klar. Emulation engine for spiking neurons and adaptive synaptic weights. In *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*, volume 5, pages 3261—-3266 vol. 5, 2005.

[153] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[154] T. Higuchi, M. Iwata, I. Kajitani, and H. Iba. Evolvable Hardware and Its Applications to Pattern Recognition and Fault-Tolerant Systems. *Lecture Notes in Computer Science*, 1062:118, 1996.

[155] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, N. Salami, N. Kajihara, and Others. Real-world applications of analog and digital evolvable hardware. *Evolutionary Computation, IEEE Transactions on*, 3(3):220–235, 1999.

[156] T. Higuchi and N. Kajihara. Evolvable hardware chips for industrial applications. *Communications of the ACM*, 42(4):60–66, 1999.

[157] T. Hoang, R. I. McKay, D. Essam, and N. X. Hoai. On Synergistic Interactions Between Evolution, Development and Layered Learning. *IEEE Transactions on Evolutionary Computation*, 15(3):287–312, June 2011.

[158] A. Hodgkin and A. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544, 1952.

[159] L. J. Hoffer. Complementary or alternative medicine: the need for plausibility. *CMAJ : Canadian Medical Association journal = journal de l'Association medicale canadienne*, 168(2):180–2, Jan. 2003.

[160] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[161] J. W. Horihan and Y.-H. H. Lu. Improving FSM Evolution with Progressive Fitness Functions. In *Great Lakes Symposium on VLSI*, pages 123–126, New York, New York, USA, 2004. ACM Press New York, NY, USA.

[162] G. S. Hornby and J. B. Pollack. The advantages of generative grammatical encodings for physical design. In *Proceedings of the 2001 Congress on Evolutionary Computation*, volume 1, pages 600–607, 2001.

[163] M. Hu, H. Li, Q. Wu, and G. S. Rose. Hardware realization of BSB recall function using memristor crossbar arrays. In *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, page 498, New York, New York, USA, June 2012. ACM Press.

[164] M. Hu, H. Li, Q. Wu, G. S. Rose, and Y. Chen. Memristor crossbar based hardware realization of BSB recall function. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, June 2012.

[165] T. Hu and W. Banzhaf. Evolvability and Speed of Evolutionary Algorithms in Light of Recent Developments in Biology. *Journal of Artificial Evolution and Applications*, 2010:1–28, 2010.

[166] N. Iannella and L. Kindermann. Finding Iterative Roots with a Spiking Neural Network. *IPL: Information Processing Letters*, 95:545–551, 2005.

[167] N. Izeboudjen, C. Larbes, and A. Farah. A new classification approach for neural networks hardware: from standards chips to embedded systems on chip. *Artificial Intelligence Review*, Mar. 2012.

[168] E. M. Izhikevich. Simple model of spiking neurons. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 14(6):1569–72, Jan. 2003.

[169] E. M. Izhikevich. Which model to use for cortical spiking neurons? *Neural Networks, IEEE Transactions on*, 15(5):1063–1070, Sept. 2004.

[170] E. M. Izhikevich. Simulation of Large-Scale Brain Models, 2005.

[171] E. M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting (Computational Neuroscience)*. The MIT Press, Nov. 2007.

[172] E. M. Izhikevich. Solving the Distal Reward Problem through Linkage of STDP and Dopamine Signaling. *Cerebral Cortex*, 17:bhl152, 2007.

[173] E. M. Izhikevich and G. M. Edelman. Large-scale model of mammalian thalamocortical systems. *Proceedings of the National Academy of Sciences of the United States of America*, 105(9):3593–8, Mar. 2008.

[174] C. Jacob. *Illustrating Evolutionary Computation with Mathematica, (The Morgan Kaufmann Series in Artificial Intelligence)*. Morgan Kaufmann, 2001.

[175] H. Jaeger. Reservoir Riddles : Suggestions for Echo State Network Research, 2005.

[176] H. Jaeger and H. Haas. Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. *Science*, 304(5667):78–80, Apr. 2004.

[177] N. Jakobi. Harnessing morphogenesis. In *On growth, form and computers*, pages 392–404. Academic Press, London, 2003.

[178] C. L. Janer, J. M. Quero, J. G. Ortega, and L. G. Franquelo. Fully parallel stochastic computation architecture. *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, 44(8):2110–2117, 1996.

[179] Y. Jewajinda and P. Chongstitvatana. A parallel genetic algorithm for adaptive hardware and its application to ECG signal classification. *Neural Computing and Applications*, June 2012.

[180] Y. Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing*, 9(1):3–12, Oct. 2003.

[181] Z. Jin. *Autonomously Reconfigurable Artificial Neural Network on a Chip*. PhD thesis, University of Pittsburgh, 2010.

[182] S. Johnston, G. Prasad, L. P. Maguire, M. Mcginnity, and T. M. McGinnity. Comparative Investigation into Classical and Spiking Neuron Implementations on FPGAs. In W. Duch, J. Kacprzyk, E. Oja, and S. Zadrozny, editors, *Artificial Neural Networks: Biological Inspirations - ICANN 2005, 15th International Conference, Warsaw, Poland, September 11-15, 2005, Proceedings, Part I*, volume 3696 of *Lecture Notes in Computer Science*, pages 269–274. Springer, 2005.

[183] B. Jones, D. Stekel, J. Rowe, and C. Fernando. Is there a Liquid State Machine in the Bacterium Escherichia Coli? In *Artificial Life, 2007. ALIFE'07. IEEE Symposium on*, pages 187–191, 2007.

[184] J. Jones. Dynamic Reconfiguration and Incremental Firmware Development in the Xilinx Virtex 5. *Topical Workshop on Electronics for Particle*, pages 583–586, 2008.

[185] 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. In *AAAI/IAAI*, pages 182–187, 1999.

[186] E. R. Kandel, J. H. Schwartz, and T. M. Jessell. *Principles of Neural Science*. McGraw-Hill, 2000.

[187] P. O. Kanold and C. J. Shatz. Subplate neurons regulate maturation of cortical inhibition and outcome of ocular dominance plasticity. *Neuron*, 51(5):627–38, Sept. 2006.

[188] N. Kapre, N. Mehta, M. DeLorimier, R. Rubin, H. Barnor, M. Wilson, M. Wrighton, and A. DeHon. Packet Switched vs. Time Multiplexed FPGA Overlay Networks. *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–216, Apr. 2006.

[189] S. A. Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437–467, 1969.

[190] G. M. Khan and J. F. Miller. Learning Games using a Single Developmental Neuron. In *Proc. of the Alife XII Conference*, pages 634–641, Odense, Denmark, 2010.

[191] G. M. Khan, J. F. Miller, and D. M. Halliday. Intelligent agents capable of developing memory of their environment. *Advances in Modelling Adaptive and Cognitive Systems*, pages 77–114, 2010.

[192] G. M. Khan, J. F. Miller, and D. M. Halliday. Evolution of cartesian genetic programs for development of learning neural architecture. *Evolutionary computation*, 19(3):469–523, Jan. 2011.

[193] M. M. Khan, G. M. Khan, and J. F. Miller. Efficient representation of Recurrent Neural Networks for markovian/non-markovian non-linear control problems. In *2010 10th International Conference on Intelligent Systems Design and Applications*, pages 615–620. IEEE, Nov. 2010.

[194] M. M. Khan, G. M. Khan, and J. F. Miller. Evolution of Optimal ANNs for Non-Linear Control Problems using Cartesian Genetic Programming. In *Proceedings of the 2010 International Conference on Artificial Intelligence, {ICAI} 2010*, volume 2, pages 339–346, 2010.

[195] J. Kilian and H. T. Siegelmann. The Dynamic Universality of Sigmoidal Neural Networks. *Information and Computation*, 128(1):48–56, July 1996.

[196] M. Kirschner and J. Gerhart. Evolvability. *Proceedings of the National Academy of Sciences of the United States of America*, 95(July):8420–8427, 1998.

[197] H. Kitano. Designing Neural Networks Using Genetic Algorithms with Graph Generation System. *Complex Systems*, 4:461–476, 1990.

[198] H. Kitano. A Simple Model of Neurogenesis and Cell Differentiation based on Evolutionary Large-Scale Chaos. *Artificial Life*, 2(1):79–99, 1995.

[199] A. Knoblauch. Neural Associative Memory for Brain Modeling and Information Retrieval. *IPL: Information Processing Letters*, 95:537–544, 2005.

[200] C. Koch. *Biophysics of computation: information processing in single neurons*. Oxford University Press, 1999.

[201] Y. Kondo and Y. Sawada. Functional abilities of a stochastic logic neural network. *Neural Networks, IEEE Transactions on*, 3(3):434–443, May 1992.

[202] P. Koopman. Embedded System Design Issues (the Rest of the Story). *Computer Design: VLSI in Computers and*, pages 310–317, 1996.

[203] T. Kowaliw and W. Banzhaf. Augmenting artificial development with local fitness. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 316–323, May 2009.

[204] J. Koza. Discovery of rewrite rules in Lindenmayer systems and state transition rules in cellular automata via genetic programming. In *Symposium on Pattern Formation (SPF-93)*, Claremont, CA, 1993.

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

[206] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[207] J. Krohn, P. J. Bentley, and H. Shayani. The Challenge of Irrationality: Fractal Protein Recipes for PI. In *Proceedings of the Genetic and Evolutionary Conmputation Conference (GECCO 2009)*, 2009.

[208] J. Krohn and D. Gorse. Fractal gene regulatory networks for control of nonlinear systems. *Parallel Problem Solving from Nature - PPSN XI*, 6239:209–218, 2011.

[209] J. Krohn and D. Gorse. Extracting Key Gene Regulatory Dynamics for the Direct Control of Mechanical Systems. *Parallel Problem Solving from Nature-PPSN XII*, 7491:468–477, 2012.

[210] S. Kumar and P. J. Bentley. An Introduction to Computational Development. In S. Kumar and P. J. Bentley, editors, *On Growth, Form and Computers*, chapter 1, pages 1–43. Academic Press, 2003.

[211] S. Kumar and P. J. Bentley, editors. *On Growth, Form and Computers*. Academic Press, 2003.

[212] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, Feb. 2007.

[213] P. R. Laming and E. Syková. *Glial Cells: Their Role in Behaviour*. Cambridge University Press, 1998.

[214] Lattice Semiconductor. LatticeXP Family Data Sheet. Technical Report November, Lattice Semiconductor, 2007.

[215] Lattice Semiconductor. Minimizing System Interruption During Configuration Using TransFR Technology. Technical Report August, Lattice Semiconductor Corp., 2011.

[216] R. Legenstein, C. Naeger, and W. Maass. What can a neuron learn with spike-timing-dependent plasticity? *Neural computation*, 17(11):2337–82, Nov. 2005.

[217] H. Li and S. X. Yang. A behavior-based mobile robot with a visual landmark-recognition system. *Mechatronics, IEEE/ASME Transactions on*, 8(3):390–400, 2003.

[218] Y. Liao. Neural networks in hardware: A survey. *Department of Computer Science, University of*, 2001.

[219] W. Lie and W. Feng-yan. Dynamic Partial Reconfiguration in FPGAs. *2009 Third International Symposium on Intelligent Information Technology Application*, pages 445–448, 2009.

[220] O. Lilienthal. *Birdflight as the Basis of Aviation*. Markowski Intl, 2000.

[221] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of theoretical biology*, 18(3):280–315, 1968.

[222] A. Lindenmayer. Developmental systems and languages in their biological context. In G. T. Herman and G. Rozenberg, editors, *Developmental Systems and Languages*, pages 1–40. North-Holland, Amsterdam, 1975.

[223] H. Liu, J. Miller, and A. Tyrrell. A biological development model for the design of robust multiplier. *Applications of Evolutionary Computing*, 3449:195–204, 2005.

[224] H. Liu, J. F. Miller, and A. M. Tyrrell. Intrinsic evolvable hardware implementation of a robust biological development model for digital systems. In *Evolvable Hardware, 2005. Proceedings. 2005 NASA/DoD Conference on*, pages 87–92, 2005.

[225] J. Liu and D. Liang. A survey of FPGA-based hardware implementation of ANNs. *Neural Networks and Brain, 2005. ICNN&B'05.*, pages 915–918, 2005.

[226] M. Liu, W. Kuehn, and Z. Lu. Run-time partial reconfiguration speed investigation and architectural design space exploration. *Field Programmable Logic*, 2009.

[227] W. Liu, M. Murakawa, and T. Higuchi. ATM Cell Scheduling by Function Level Evolvable Hardware. In T. Higuchi, M. Iwata, and W. Liu, editors, *Evolvable Systems: From Biology to Hardware, First International Conference, ICES 96, Tsukuba, Japan, October 7-8, 1996, Proceedings*, volume 1259 of *Lecture Notes in Computer Science*, pages 180–192. Springer, 1996.

[228] A. Livnat, C. Papadimitriou, N. Pippenger, and M. W. Feldman. Sex, mixability, and modularity. *Proceedings of the National Academy of Sciences of the United States of America*, 107(4):1452–7, Jan. 2010.

[229] C. Lobb, R. Fujimoto, and S. Potter. Parallel Event-Driven Neural Network Simulations Using the Hodgkin-Huxley Neuron Model. *Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*, pages 16–25, 2005.

[230] J. Lohn, G. Larchev, and R. DeMara. A Genetic Representation for Evolutionary Fault Recovery in Virtex FPGAs. In A. M. Tyrrell, P. C. Haddow, and J. Torresen, editors, *Evolvable Systems: From Biology to Hardware, Fifth International Conference, ICES 2003*, volume 2606 of *LNCS*, pages 47–56, Trondheim, Norway, 2003. Springer-Verlag.

[231] J. D. Lohn and G. S. Hornby. Evolvable hardware using evolutionary computation to design and optimize hardware systems. *Computational Intelligence Magazine, IEEE*, 1(1):19–27, 2006.

[232] M. Lukosevicius and H. Jaeger. Overview of Reservoir Recipes. Technical Report 11, School of Engineering and Science, Jacobs University, Bremen, Germany, July 2007.

[233] M. Lukosevicius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, Aug. 2009.

[234] M. Lukosevicius, H. Jaeger, and B. Schrauwen. Reservoir Computing Trends. *KI - Künstliche Intelligenz*, May 2012.

[235] A. Maache, J. Reeve, and M. Zwolinski. Optimising physical wires usage in mesh-based multi-FPGA systems using partition swapping. In *2009 International Conference on Microelectronics - ICM*, pages 252–255. IEEE, Dec. 2009.

[236] W. Maass. On the Computational Power of Noisy Spiking Neurons. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 211–217. The {MIT} Press, 1996.

[237] W. Maass. Networks of Spiking Neurons: The Third Generation of Neural Network Models. *Neural networks*, 10(9):1659–1671, 1997.

[238] W. Maass. Noisy Spiking Neurons with Temporal Coding have more Computational Power than Sigmoidal Neurons. Technical report, Institute for Theoretical Computer Science, 1999.

[239] W. Maass and C. M. Bishop. *Pulsed Neural Networks*. MIT Press Cambridge, MA, USA, 1999.

[240] W. Maass, T. Natschläger, and H. Markram. Fading memory and kernel properties of generic cortical microcircuit models. *Journal of physiology, Paris*, 98(4-6):315–30, 2004.

[241] W. Maass, T. Natschläger, H. Markram, Maass, Natschlager, and Markram. Real-time Computing Without Stable States: A New Framework for Neural Computation Based on Perturbations. *NEURCOMP: Neural Computation*, 14(11):2531–2560, Nov. 2002.

[242] L. P. Maguire, T. M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin. Challenges for large-scale implementations of spiking neural networks on FPGAs. *Neurocomputing*, 71(1-3):13–29, Dec. 2007.

[243] S. W. Mahfoud. Crowding and Preselection Revisited. *Parallel Problem Solving From Nature*, 2:27–36, 1992.

[244] M. Mahsal Khan, A. Masood Ahmad, G. Muhammad Khan, J. F. Miller, A. Prieto, F. Sandoval, and M. Atencia. Fast learning neural networks using Cartesian genetic programming. *Neurocomputing*, 121:274–289, Dec. 2013.

[245] D. Malkin. *The Evolutionary Impact of Gradual Complexification on Complex Systems*. PhD thesis, Deptartment of Computer Science, University College London, UCL, 2008.

[246] A. Manwani and C. Koch. Detecting and Estimating Signals in Noisy Cable Structures, I: Neuronal Noise Sources. *Neural Computation*, 11(8):1797–1829, Nov. 1999.

[247] H. Markram. The blue brain project. *Nature reviews. Neuroscience*, 7(2):153–60, Feb. 2006.

[248] H. Markram, D. Pikus, A. Gupta, and M. Tsodyks. Potential for multiple mechanisms, phenomena and algorithms for synaptic plasticity at single synapses. *Neuropharmacology*, 37(4-5):489–500, 1998.

[249] M. Martincigh and A. Abramo. A new architecture for digital stochastic pulse-mode neurons based on the voting circuit. *Neural Networks, IEEE Transactions on*, 16(6):1685–1693, Nov. 2005.

[250] M. Maslanka and M. Gorgon. A survey of FPGA implementations of Artificial Spiking Neurons Models. *Bio-Algorithms and Med-Systems*, 8(1):77, 2012.

[251] R. M. May. Simple mathematical models with very complicated dynamics. *Nature*, 261:459–467, 1976.

[252] S. Maya, R. Reynoso, C. Torres, and M. Arias-Estrada. Compact spiking neural network implementation in FPGA. In *FPL 2000*, pages 270–276, 2000.

[253] P. Mazumder and E. M. Rudnick. *Genetic algorithms for VLSI design, layout & test automation*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1999.

[254] W. S. McCulloch and W. Pitts. A Logical Calculus of the Idea Immanent in Nervous Activity. *Bull. Math. Biophys.*, 5:115–133, 1943.

[255] E. J. E. McDonald. Runtime FPGA partial reconfiguration. *IEEE A&E Systems Magazine*, 23(7):10–15, 2008.

[256] M. Meeter, J. Jehee, and J. Murre. Neural Models that Convince: Model Hierarchies and Other Strategies to Bridge the Gap Between Behavior and the Brain. *Philosophical Psychology*, 20(6):749–772, Dec. 2007.

[257] N. Mehrtash, D. Jung, H. H. Hellmich, T. Schoenauer, V. T. Lu, and H. Klar. Synaptic plasticity in spiking neural networks (SP(2)INN): a system approach. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 14(5):980–92, Jan. 2003.

[258] H. Meinhardt. *Models of biological pattern formation*. Academic Press London, 1982.

[259] H. Meinhardt. *The algorithmic beauty of sea shells*. Springer, 2003.

[260] J. Miller. A developmental method for growing graphs and circuits. In *ICES 2003*, pages 93–104, 2003.

[261] J. F. Miller. Evolving Developmental Programs for Adaptation, Morphogenesis, and Self-Repair. In W. Banzhaf, T. Christaller, P. Dittrich, J. T. Kim, and J. Ziegler, editors, *Advances in Artificial Life: 7th European Conference, Ecal 2003 Dortmund, Germany, September 14-17, 2003 Proceedings*, Lecture Notes in Artificial Intelligence, pages 256–265, Dortmund, Germany, 2003. Springer.

[262] J. F. Miller. *Cartesian Genetic Programming (Natural Computing Series)*. Springer, 2011.

[263] J. F. Miller and W. Banzhaf. Evolving the program for a cell: From French flags to Boolean circuits. In *On Growth, Form and Computers*, pages 278–302. Academic Press London, 2003.

[264] J. F. Miller and G. M. Khan. Where is the brain inside the brain? *Memetic Computing*, 3(3):217–228, June 2011.

[265] J. F. Miller and S. L. Smith. Redundancy and Computational Efficiency in Cartesian Genetic Programming. *Evolutionary Computation, IEEE Transactions on*, 10(2):167–174, 2006.

[266] J. F. Miller and P. Thomson. Cartesian Genetic Programming. *Genetic Programming, Proceedings of EuroGP'2000*, 1802:121–132, 2000.

[267] N. Miskov-Zivanov, A. Bresticker, D. Krishnaswamy, S. Venkatakrishnan, P. Kashinkunti, D. Marculescu, and J. R. Faeder. Regulatory network analysis acceleration with reconfigurable hardware. *Conference proceedings : ... Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Conference*, 2011:149–52, Jan. 2011.

[268] J. Misra and I. Saha. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, 74(1-3):239–255, Dec. 2010.

[269] M. Mokhtar, D. Halliday, and A. Tyrrell. Hippocampus-Inspired Spiking Neural Network on FPGA. *Evolvable Systems: From Biology to Hardware*, 5216:362–371, 2008.

[270] J. Moreno, J. Eriksson, and J. Iglesias. Implementation of Biologically Plausible Spiking Neural Networks Models on the POEtic Tissue. *Evolvable Systems: From*, pages 188–197, 2005.

[271] J. M. Moreno, Y. Thoma, E. Sanchez, J. Eriksson, J. Iglesias, and A. Villa. The POEtic Electronic Tissue and Its Role in the Emulation of Large-Scale Biologically Inspired Spiking Neural Networks Models. *Complexus*, 3(1-3):32–47, 2006.

[272] F. Morgan, S. Cawley, B. Mc Ginley, S. Pande, L. Mc Daid, B. Glackin, J. Maher, J. Harkin, and B. McGinley. Exploring the evolution of NoC-based spiking neural networks on FPGAs. *2009 International Conference on Field-Programmable Technology*, pages 300–303, Dec. 2009.

[273] V. Mountcastle. An organizing principle for cerebral function: The unit model and the distributed system. *The Mindful Brain*, 1978.

[274] M. Nagda, C. F. Devaraj, I. Gupta, and G. Agha. DiffGen : A Toolkit for Generating Distributed Protocol Code. In *Poster at 23rd ACM Symp. Principles of Distributed Computing (PODC), 2004*, 2004.

[275] J. Navaridas, M. Luj'n, L. A. Plana, J. Miguel-Alonso, and S. B. Furber. Analytical Assessment of the Suitability of Multicast Communications for the SpiNNaker Neuromimetic System. *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 1–8, June 2012.

[276] D. Nikolić, S. Haeusler, W. Singer, and W. Maass. Temporal dynamics of information content carried by neurons in the primary visual cortex. In *Advances in Neural Information Processing Systems*, volume 19, pages 1041–1048, 2007.

[277] S. Nolfi, O. Miglino, and D. Parisi. Phenotypic plasticity in evolving neural networks. In D. Gaussier and J.-D. Nicoud, editors, *From Perception to Action: Proceedings of the International Conference*, pages 146–157. IEEE Comput. Soc. Press, 1994.

[278] T. G. Noll, T. von Sydow, B. Neumann, J. Schleifer, T. Coenen, and G. Kappen. Reconfigurable Components for Application-Specific Processor Architectures. In M. Platzner, J. Teich, and N. Wehn, editors, *Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications*, pages 25–49. Springer Netherlands, Dordrecht, 2010.

[279] J.-B. Note and E. Rannaud. From the bitstream to the netlist. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, page 264, New York, NY, USA, 2008. ACM.

[280] Numenta Inc. Grok (www.numenta.com/grok_info.html).

[281] Numenta Inc. HIERARCHICAL TEMPORAL MEMORY including HTM Cortical Learning Algorithms (VERSION 0.2.1). Technical report, Numenta, Inc., 2011.

[282] C. W. Omlin and C. L. Giles. Constructing deterministic finite-state automata in sparse recurrent neural networks. In *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on*, volume 3, pages 1732—-1737 vol.3, 1994.

[283] C. Ortega and A. Tyrrell. Biologically inspired fault-tolerant architectures for real-time control applications. *Control Engineering Practice*, 7(5):673–678, May 1999.

[284] S. Pande, F. Morgan, G. Smit, T. Bruintjes, J. Rutgers, B. McGinley, S. Cawley, J. Harkin, and L. McDaid. Fixed Latency On-Chip Interconnect for Hardware Spiking Neural Network Architectures. *Parallel Computing*, Apr. 2013.

[285] H. Parvez and H. Mehrez. *Application-Specific Mesh-Based Heterogeneous FPGA Architectures*, volume 2010. Springer, 2010.

[286] N. D. Patel, S. K. Nguang, and G. G. Coghill. Neural network implementation using uniformly weighted bit-streams. *2008 IEEE International Conference on Industrial Technology*, pages 1–6, Apr. 2008.

[287] C. Patterson, J. Garside, E. Painkras, S. Temple, L. a. Plana, J. Navaridas, T. Sharp, and S. Furber. Scalable communications for a million-core neural processing architecture. *Journal of Parallel and Distributed Computing*, Feb. 2012.

[288] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 4th edition, 2011.

[289] M. J. Pearson, C. Melhuish, A. G. Pipe, M. Nibouche, I. Gilhesphy, K. Gurney, and B. Mitchinson. Design and FPGA implementation of an embedded real-time biologically plausible spiking neural network processor. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 582–585, 2005.

[290] M. J. Pearson, A. G. Pipe, B. Mitchinson, K. Gurney, C. Melhuish, I. Gilhespy, and M. Nibouche. Implementing Spiking Neural Networks for Real-Time Signal-Processing and Control Applications: A Model-Validated FPGA Approach. *Neural Networks, IEEE Transactions on*, 18(5):1472–1487, 2007.

[291] C. A. Pena-Reyes and M. Sipper. Evolutionary computation in medicine: an overview. *Artificial Intelligence in Medicine*, 19(1):1–23, 2000.

[292] D. Perlis. Hawkins on intelligence: Fascination and frustration. *Artificial Intelligence*, 169(2):184–191, Dec. 2005.

[293] F. W. Pfrieger. Role of glia in synapse development. *Current Opinion in Neurobiology*, 12:486–490, 2002.

[294] T. M. Pinkston and J. Duato. Interconnection Networks. In J. L. Hennessy and D. A. Patterson, editors, *Computer Architecture: A Quantitative Approach*, chapter Appendix F, pages 1–117. Morgan Kaufmann, 5th edition, 2011.

[295] R. E. Pino. Memristor-based synapse design and training scheme for neuromorphic computing architecture. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–5. Ieee, June 2012.

[296] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, 1990.

[297] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits (2nd Edition)*. Prentice Hall, 2003.

[298] R. A. Raff. Evo-devo: the evolution of a new discipline. *Nature reviews. Genetics*, 1(1):74–9, Oct. 2000.

[299] S. Ramsey, D. Orrell, H. Bolouri, and Others. Dizzy: stochastic simulation of large-scale genetic regulatory networks. *Journal of Bioinformatics and Computational Biology*, 3(2):415–436, 2005.

[300] A. Rast, F. Galluppi, S. Davies, L. Plana, C. Patterson, T. Sharp, D. Lester, and S. Furber. Concurrent heterogeneous neural model simulation on real-time neuromimetic hardware. *Neural networks : the official journal of the International Neural Network Society*, 24(9):961–78, Nov. 2011.

[301] A. Rast, X. Jin, F. Galluppi, L. Plana, C. Patterson, and S. Furber. Scalable event-driven native parallel processing: The spinnaker neuromimetic system. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 21–30, 2010.

[302] J. Reisinger and R. Miikkulainen. Selecting for evolvable representations. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation - GECCO '06*, page 1297, New York, New York, USA, July 2006. ACM Press.

[303] K. L. Rice, M. A. Bhuiyan, T. M. Taha, C. N. Vutsinas, and M. C. Smith. FPGA Implementation of Izhikevich Spiking Neural Networks for Character Recognition. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 451–456. Ieee, Dec. 2009.

[304] S. Risi and K. Stanley. Indirectly encoding neural plasticity as a pattern of local rules. *From Animals to Animats 11*, 6226:533–543, 2010.

[305] S. Risi and K. O. Stanley. Enhancing es-hyperneat to evolve more complex regular neural networks. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11*, pages 1539–1546, New York, New York, USA, 2011. ACM Press.

[306] S. Risi and K. O. Stanley. An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons. *Artificial life*, 18(4):331–63, Jan. 2012.

[307] A. Rodan and P. Tino. Minimum complexity echo state network. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 22(1):131–44, Jan. 2011.

[308] A. Rodan and P. Tino. Simple deterministically constructed cycle reservoirs with regular jumps. *Neural computation*, 24(7):1822–52, July 2012.

[309] J. J. Rodriguez-Andina, M. J. Moure, and M. D. Valdes. Features, Design Tools, and Application Domains of FPGAs. *IEEE Transactions on Industrial Electronics*, 54(4):1810, 2007.

[310] D. Roggen. *Multi-cellular Reconfigurable Circuits : Evolution , Morphogenesis and Learning*. PhD thesis, EPFL, 2005.

[311] D. Roggen, D. Federici, and D. Floreano. Evolutionary morphogenesis for multi-cellular systems. *Genetic Programming and Evolvable Machines*, 8(1):61–96, 2007.

[312] D. Roggen, S. Hofmann, Y. Thoma, and D. Floreano. Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot. *Evolvable Hardware, 2003. Proceedings. NASA/DoD Conference on*, pages 189–198, 2003.

[313] B. F. Romanescu and D. J. Sorin. Core cannibalization architecture. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*, page 43, New York, New York, USA, 2008. ACM Press.

[314] E. Ros, E. M. Ortigosa, R. Agís, R. Carrillo, and M. Arnold. Real-time computing platform for spiking neurons (RT-spike). *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 17(4):1050–63, July 2006.

[315] E. Ros, E. M. Ortigosa, R. Agis, R. R. Carrillo, A. Prieto, M. Arnold, and E. M Ortigosa. Spiking neurons computing platform. *Lecture Notes in Computer Science*, 3512:471–478, 2005.

[316] M. Rouhipour, P. Bentley, and H. Shayani. Systemic computation using graphics processors. *Evolvable Systems: From Biology to Hardware*, 6274/2010:121–132, 2010.

[317] M. Rouhipour, P. J. Bentley, and H. Shayani. Fast bio-inspired computation using a GPU-based systemic computer. *Parallel Computing*, 36(10-11):591–617, Oct. 2010.

[318] M. Rubinov and O. Sporns. Brain Connectivity Toolbox, http://www.brain-connectivity-toolbox.net, 2010.

[319] M. Rubinov and O. Sporns. Complex network measures of brain connectivity: uses and interpretations. *NeuroImage*, 52(3):1059–69, Sept. 2010.

[320] M. Rucci and G. Edelman. Adaptation of orienting behavior: from the barn owl to a robotic system. *Robotics and Automation,*, 15(1):96–110, 1999.

[321] A. G. Rust, R. O. D. Adams, M. Schilstra, and H. Bolouri. *Evolving computational neural systems using synthetic developmental mechanisms*, chapter 19, pages 353–376. Academic Press, 2003.

[322] M. Salami, M. Murakawa, and T. Higuchi. Data Compression Based on Evolvable Hardware. In T. Higuchi, M. Iwata, and W. Liu, editors, *Evolvable Systems: From Biology to Hardware, First International Conference, ICES 96, Tsukuba, Japan, October 7-8, 1996, Proceedings*, volume 1259 of *Lecture Notes in Computer Science*, pages 169–179. Springer, 1996.

[323] L. Salwinski and D. Eisenberg. In silico simulation of biological network dynamics. *Nature biotechnology*, 22(8):1017–9, Aug. 2004.

[324] E. Sanchez, A. Perez-Uribe, A. Upegui, Y. Thoma, J. Moreno, A. Napieralski, A. Villa, G. Sassatelli, H. Volken, and E. Lavarec. PERPLEXUS: Pervasive Computing Framework for Modeling Complex Virtually-Unbounded Systems. In *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pages 587–591. Ieee, Aug. 2007.

[325] M. Santambrogio, A. Cazzaniga, A. Bonetto, and D. Sciuto. ReBit: A Tool to Manage and Analyse FPGA-Based Reconfigurable Systems. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 220–227, May 2011.

[326] J. Schaffer, D. Whitley, and L. Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 1–37. IEEE Comput. Soc. Press, June 1992.

[327] J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. Gomez. Training recurrent networks by Evolino. *Neural computation*, 19(3):757–79, Mar. 2007.

[328] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki. A Statically Scheduled Time-Division-Multiplexed Network-on-Chip for Real-Time Systems. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pages 152–160. IEEE, May 2012.

[329] B. Schrauwen and M. D'Haene. Compact digital hardware implementations of spiking neural networks. Sixth FirW PhD Symposium, page on CD, 2005.

[330] B. Schrauwen, M. D'Haene, D. Verstraeten, and J. V. Campenhout. Compact hardware liquid state machines on FPGA for real-time speech recognition. *Neural networks : the official journal of the International Neural Network Society*, 21(2-3):511–23, 2008.

[331] B. Schrauwen and J. van Campenhout. Parallel hardware implementation of a broad class of spiking neurons using serial arithmetic. In *Proceedings of ESANN*, pages 623–628, 2006.

[332] B. Schrauwen, D. Verstraeten, and J. Van Campenhout. An overview of reservoir computing: theory, applications and implementations. In *Proceedings of the 15th European Symposium on Artificial Neural Networks*, pages 471–482, 2007.

[333] N. Sedcole. *Reconfigurable platform-based design in FPGAs for video image processing*. PhD thesis, Imperial College of Science, Technology and Medicine University of London, 2006.

[334] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. A new architecture for digital stochastic pulse-mode neurons based on the voting circuit. *Computers and Digital Techniques, IEE Proceedings* -, 153(3):157–164, 2006.

[335] L. Sekanina. Evolution of Digital Circuits Operating as Image Filters in Dynamically Changing Environment. *Proc. of the 8th International Conference on Soft Computing Mendel*, 2:33–38, 2002.

[336] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[337] H. Shayani and P. J. Bentley. A more bio-plausible approach to the evolutionary inference of finite state machines. *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2937–2944, 2007.

[338] H. Shayani, P. J. Bentley, and A. M. Tyrrell. An FPGA-based Model Suitable for Evolution and Development of Spiking Neural Networks. In *Proc of 16th European Symposium on Artificial Neural Networks, Advances in Computational Intelligence and Learning*, 2008.

[339] H. Shayani, P. J. Bentley, and A. M. Tyrrell. Hardware Implementation of a Bio-plausible Neuron Model for Evolution and Growth of Spiking Neural Networks on FPGA. In *AHS '08: Proceedings of the 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 236–243, Washington, DC, USA, June 2008. IEEE Computer Society.

[340] G. Shepherd. *The synaptic organization of the brain*. OUP USA, 2004.

[341] S. Singh and P. James-Roxby. Lava and JBits: From HDL to bitstream in seconds. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pages 91–100, 2001.

[342] 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. *Evolutionary Computation, 1997., IEEE Transactions on*, 1(1):83–97, Apr. 1997.

[343] T. Smith, P. Husbands, P. Layzell, and M. O'Shea. Fitness Landscapes and Evolvability. *Evolutionary Computation*, 10(1):1—-, Mar. 2002.

[344] S. Song, K. D. Miller, and L. F. Abbott. Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *nature neuroscience*, 3(9):919–926, 2000.

[345] W. Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall, 8th edition, 2009.

[346] K. O. Stanley. Real-time neuroevolution in the NERO video game. *IEEE transactions on evolutionary computation*, 9(6):653—-, 2005.

[347] K. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, May 2007.

[348] K. O. Stanley. Evolving Neural Networks. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion - GECCO Companion '12*, pages 805–826, New York, New York, USA, July 2009. ACM Press.

[349] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, Jan. 2009.

[350] K. O. Stanley and R. Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[351] K. O. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. *Artificial life*, 9(2):93–130, Jan. 2003.

[352] K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21(1):63–100, 2004.

[353] J. J. Steil. Backpropagation-Decorrelation: online recurrent learning with O(N) complexity. *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, 2:843–848, 2004.

[354] J. J. Steil. Online reservoir adaptation by intrinsic plasticity for backpropagation–decorrelation and echo state learning. *Neural Networks*, 20(3):353–364, 2007.

[355] P. N. Steinmetz, A. Manwani, C. Koch, M. London, and I. Segev. Subthreshold voltage noise due to channel fluctuations in active neuronal membranes. *Journal of computational neuroscience*, 9(2):133–48, 2000.

[356] S. Stepney, S. L. Braunstein, J. A. Clark, A. Tyrrell, A. Adamatzky, R. E. Smith, T. Addis, C. Johnson, J. Timmis, P. Welch, R. Milner, and D. Partridge. Journeys in non-classical computation I: A grand challenge for computing research. *International Journal of Parallel, Emergent and Distributed Systems*, 20(1):5–19, Mar. 2005.

[357] J. Strunk, T. Volkmer, and K. Stephan. Impact of Run-Time Reconfiguration on Design and Speed - A Case Study Based on a Grid of Run-Time Reconfigurable Modules inside a FPGA. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009.

[358] J. Taheri and A. Zomaya. Artificial Neural Networks. In A. Zomaya, editor, *Handbook of Nature-Inspired and Innovative Computing*, pages 147–185. Springer US, 2006.

[359] F. Takens. Detecting strange attractors in turbulence. *Dynamical Systems and Turbulence*, 898:366–381, 1981.

[360] J. G. Taylor. Book review: Jeff Hawkins and Sandra Blakeslee, On Intelligence. *Artificial Intelligence*, 169:192–195, 2005.

[361] G. Tempesti, D. Mange, E. Petraglio, and Y. Thoma. Developmental Processes in Silicon : An Engineering Perspective. In *Evolvable Hardware, 2003. Proceedings. NASA/DoD Conference on*, pages 255–264. IEEE Computer Society Press, 2003.

[362] Y. Thoma and E. Sanchez. An adaptive FPGA and its distributed routing. *Proc. ReCoSoc '05 Reconfigurable Communication-centric SoC*, pages 43–51, 2005.

[363] D. Thomas and W. Luk. FPGA Accelerated Simulation of Biologically Plausible Spiking Neural Networks. *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 45–52, 2009.

[364] A. Thompson. An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics. In T. Higuchi, M. Iwata, and L. Weixin, editors, *Proc. 1st Int. Conf. on Evolvable Systems (ICES'96)*, pages 390–405, Berlin, 1997. Springer-Verlag.

[365] J. Timmis, M. Amos, W. Banzhaf, and A. Tyrrell. Going back to our roots: second generation biocomputing. *Journal of Unconventional Computing*, 2:349–378, 2006.

[366] O. Torres, J. Eriksson, J. M. Moreno, and A. Villa. Hardware optimization and serial implementation of a novel spiking neuron model for the POEtic tissue. *Biosystems*, 76(1-3):201–208, 2004.

[367] J. Torresen. A Divide-and-Conquer Approach to Evolvable Hardware. *Lecture Notes in Computer Science*, 1478:57–65, 1998.

[368] A. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London Series B*, 237:37–72, 1952.

[369] A. J. Turner and J. F. Miller. Cartesian Genetic Programming encoded Artificial Neural Networks : A Comparison using Three Benchmarks. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference - GECCO '13*, page 1005, New York, New York, USA, July 2013. ACM Press.

[370] A. Tyrrell and E. Sanchez. Poetic tissue: An integrated architecture for bio-inspired hardware. *ICES 2003, From Biology to Hardware*, pages 129–140, 2003.

[371] A. M. Tyrrell. Reconfigurable and Evolvable Architectures and their role in Designing Computational Systems. In *Proceedings Of The 2011 International Conference On Engineering Of Reconfigurable Systems & Algorithm (ERSA 2011)*, pages 148–156. CSREA Press, 2011.

[372] A. Upegui. *Dynamically Reconfigurable Bio-inspired Hardware*. PhD thesis, Lausanne, EPFL, 2006.

[373] A. Upegui. Dynamically Reconfigurable Hardware for Evolving Bio-Inspired Architectures. *Intelligent Systems for Automated Learning and Adaptation: Emerging Trends and Applications*, 2010.

[374] A. Upegui, C. A. Pena-Reyes, and E. Sanchez. An FPGA platform for on-line topology exploration of spiking neural networks. *Microprocessors and Microsystems*, 29(5):211–223, June 2005.

[375] A. Upegui, A. Perez-Uribe, Y. Thoma, and E. Sanchez. Neural Development on the Ubichip by Means of Dynamic Routing Mechanisms. *Evolvable Systems: From Biology to Hardware: 8th International Conference Proceedings, ICES 2008, LNCS 5216*, 5216:392–401, 2008.

[376] A. Upegui and E. Sanchez. Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs. *Evolvable Systems: From Biology to Hardware*, 3637/2005:56–65, 2005.

[377] A. Upegui and E. Sanchez. Evolving Hardware with Self-reconfigurable Connectivity in Xilinx FPGAs. In *Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on*, pages 153–162, 2006.

[378] A. Upegui, Y. Thoma, and E. Sanchez. The PERPLEXUS Bio-Inspired Hardware Platform : A Flexible and Modular Approach. *International Journal of Knowledge-Based and Intelligent Engineering Systems*, 12(3):201–212, 2008.

[379] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. Moreno, and J. Madrenas. The Perplexus bio-inspired reconfigurable circuit. In *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pages 600–605. Ieee, Aug. 2007.

[380] A. Upegui, Y. Thoma, H. F. Satizabal, F. Mondada, P. Retornaz, Y. Graf, A. Perez-Iribe, and E. Sanchez. Ubichip, ubidule, and marxbot: a hardware platform for the simulation of complex systems. *ICES 2010, Biology to Hardware*, pages 286–298, 2010.

[381] A. A. Upegui, C. A. Peña Reyes, E. Sanchez, and C. A. Pena-Reyes. A hardware implementation of a network of functional spiking neurons with hebbian learning. *Biologically Inspired Approaches to Advanced Information Technology*, 3141:233–243, 2004.

[382] M. van Daalen, P. Jeavons, and J. Shawe-Taylor. A stochastic neural architecture that exploits dynamically reconfigurable FPGAs. In *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, pages 202–211, 1993.

[383] M. van Daalen, T. Kosel, P. Jeavons, and J. Shawe-Taylor. Emergent activation functions from a stochastic bit-stream neuron. In *Electronics Letters*, volume 30, pages 331–333, 1994.

[384] T. van der Zant, V. Becanovíc, K. Ishii, H. U. Kobialka, and P. G. Ploger. Finding Good Echo State Networks to Control an Underwater Robot Using Evolutionary Computations. In *Proceedings of the 5th IFAC symposium on Intelligent Autonomous Vehicles (IAV04)*, 2004.

[385] N. V. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, New York., 2000.

[386] Z. Vasicek and L. Sekanina. Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. *Genetic Programming and Evolvable Machines*, 12(3):305–327, Mar. 2011.

[387] V. K. Vassilev and J. F. Miller. Scalability Problems of Digital Circuit Evolution: Evolvability and Efficient Designs. In *Evolvable Hardware*, pages 55–64. IEEE Computer Society, 2000.

[388] P. Verbancsics and K. O. Stanley. Constraining connectivity to encourage modularity in hyperneat. *Department of Electrical Engineering and Computer Science, University of Central Florida, UCF Dept. of EECS Technical Report CS-TR-10-10*, 2010.

[389] Verstraeten, Schrauwen, Stroobandt, and V. Campenhout. Isolated Word Recognition with the Liquid State Machine: A Case Study. *IPL: Information Processing Letters*, 95:521–528, 2005.

[390] D. Verstraeten, B. Schrauwen, M. D'Haene, and D. Stroobandt. An experimental unification of reservoir computing methods. *Neural Netw.*, 20(3):391–403, 2007.

[391] D. Verstraeten, B. Schrauwen, and D. Stroobandt. Reservoir Computing with Stochastic Bitstream Neurons. In *Proceedings of the 16th Annual ProRISC Workshop*, pages 454–459, 2005.

[392] K. A. Vinger and J. Torresen. Implementing Evolution of FIR-Filters Efficiently in an FPGA. In *Evolvable Hardware*, pages 26–32. IEEE Computer Society, 2003.

[393] J. Von Neumann. Theory of self-reproducing automata, 1966.

[394] J. Von Neumann. First Draft of a Report on the EDVAC. *Annals of the History of Computing, IEEE*, 15(1):11–21, 1993.

[395] J. Von Neumann. *The Computer and the Brain*. Yale University Press, 2000.

[396] J. Vreeken. Spiking neural networks, an introduction. Technical report, Institute for Information and Computing Sciences, Utrecht University, 2002.

[397] J. F. Wakerly. *Digital Design: Principles and Practices, Third Edition*. Prentice Hall, 1999.

[398] J. A. Walker and J. F. Miller. Evolution and Acquisition of Modules in Cartesian Genetic Programming. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 187–197, Coimbra, Portugal, 2004. Springer-Verlag.

[399] J. A. Walker, J. F. Miller, and R. Cavill. A multi-chromosome approach to standard and embedded cartesian genetic programming. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 903–910. ACM Press New York, NY, USA, 2006.

[400] B. Webb. Can robots make good models of biological behaviour? *The Behavioral and brain sciences*, 24(6):1033–50; discussion 1050–94, Dec. 2001.

[401] R. K. Weinstein, M. S. Reid, and R. H. Lee. Methodology and design flow for assisted neural-model implementations in FPGAs. *IEEE transactions on neural systems and rehabilitation engineering : a publication of the IEEE Engineering in Medicine and Biology Society*, 15(1):83–93, Mar. 2007.

[402] S. Wolfram. *A New Kind of Science*. Wolfram Media, Inc., Champaign, IL, 2002.

[403] D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, Apr. 1997.

[404] L. Wolpert. *Principles of Development*. Oxford Higher Education, 2001.

[405] Xilinx. *ML505/ML506 Evaluation Platform User Guide*, Oct. 2007.

[406] Xilinx Inc. *Difference-Based Partial Reconfiguration*. Xilinx, Dec. 2007.

[407] Xilinx Inc. *Virtex-5 Family Overview LX, LXT, and SXT Platforms*. Xilinx, 2007.

[408] Xilinx Inc. Virtex-5 FPGA User Guide. Technical report, Xilinx Inc., 2008.

[409] Xilinx Inc. Virtex-5 Libraries Guide for HDL Designs. Technical report, Xilinx Inc., 2009.

[410] Xilinx Inc. Virtex-5 FPGA Data Sheet : DC and Switching Characteristics Virtex-5 FPGA Electrical Characteristics. Technical report, Xilinx Inc., 2010.

[411] Xilinx Inc. Virtex-5 FPGA Configuration User Guide. Technical report, Xilinx, 2011.

[412] Xilinx Inc. 7 Series FPGAs Overview, Advance Product Specification. Technical report, Xilinx Inc., 2012.

[413] Xilinx Inc. Spartan-3 FPGA Family Datasheet. Technical report, Xilinx Inc., 2012.

[414] Xilinx Inc. Xilinx Constraints Guide. Technical report, Xilinx Inc., 2012.

[415] S. Yang and T. M. McGinnity. A biologically plausible real-time spiking neuron simulation environment based on a multiple-FPGA platform. *ACM SIGARCH Computer Architecture News*, 39(4):78, Dec. 2011.

[416] Z. Yang, A. Murray, F. Worgotter, K. Cameron, and V. Boonsobhak. A neuromorphic depth-from-motion vision model with STDP adaptation. *Neural Networks, IEEE Transactions on*, 17(2):482–495, 2006.

[417] X. Yao. Evolving Artificial Neural Networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[418] X. Yao and T. Higuchi. Promises and challenges of evolvable hardware. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 29(1):87–97, 1999.

[419] P. Zahadat, T. Schmickl, and K. Crailsheim. Evolving Reactive Controller for a Modular Robot: Benefits of the Property of State-Switching in Fractal Gene Regulatory Networks. *From Animals to Animats 12*, 7426:209–218, 2012.

[420] S. Zhan, J. F. Miller, and A. M. Tyrrell. A developmental gene regulation network for constructing electronic circuits. *Evolvable Systems: From Biology to Hardware*, 2008.

[421] D. Zhang, H. Li, and S. Y. Foo. A simplified FPGA implementation of neural network algorithms integrated with stochastic theory for power electronics applications. In *Industrial Electronics Society, 2005. IECON 2005. 32nd Annual Conference of IEEE*, page 6pp., 2005.

[422] J. Zhu and P. Sutton. FPGA Implementations of Neural Networks - a Survey of a Decade of Progress. *Field Programmable Logic and Application*, 2778:4–7, 2003.

[423] A. Y. Zomaya. *Handbook of Nature-Inspired And Innovative Computing: Integrating Classical Models with Emerging Technologies*. Springer, 2006.