NTNU
Norwegian University of
Science and Technology

Reservoir Computing in the Spiking Domain Using Developmental Cellular Automata Machines

Øyvind Robertsen

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

As computer systems and networks grow in size and complexity, traditional top-down engineering techniques are quickly beckoming inadequate for achieving the desired results. Designing such systems that are robust and resilient, are able to adapt and self-regulate, can self-reproduce and learn autonomously is a tremendously hard task. These features are however present in many biological organisms, wherein they emerge through evolution and development. In the fields of unconventional and biologically inspired computing, these techniques are used to create computing systems with the same type of complexity as that found in biological systems. Cellular Automata (CAs) are an example of a biologically inspired computing system that can achieve complex, global computation through local interaction between simple cells at a vast scale.

While the computational capabilities of CAs have been researched extensively, they have not seen mainstream adaption as a computational paradigm. Programmability and encoding/decoding of input/output are two major challenges facing cellular computing systems. Manually specifying the functionality of each cell in such a way that the desired emergent global behavior of the CA as a whole is achieved, is infeasible for non-trivial systems. Problem input is usually encoded in the initial state of the system, and output is decoded from the state after the system has been simulated some amount of time.

The Cellular Automata Research Platform (CARP) is an FPGA-based implementation of a developmental cellular architecture, aiming to facilitate research into use of artificial development and evolution to create cellular computing systems. At an abstract level, it implements a dynamical system with dynamical structure $(DS)^2$, a system for which both behavior and structure are emerging properties. Behavior influences further structural development and vice versa.

In this thesis, the platform is extended to incorporate the developmental CA into a reservoir computing architecture. Reservoir computing (RC) is a novel approach to machine learning in which temporal input is imposed as perturbations on a dynamic reservoir and output is read out by performing a linear classification of the reservoir state some time after the initial perturbation. By combining RC and developmental CAs, the CARP system solves many of the issues relating to programming of and I/O encoding/decoding with cellular computing systems. It also opens up new possibilities for the developed organisms to adapt and learn based on their environment. The CARP platform has been extended with a reconfigurable readout layer implemented as a spiking neural network (SNN) that classifies the dynamics of the reservoir, the developmental CA. An SNN is chosen to allow the system operate entirely in the spiking domain, as input data and the dynamic behavior of the reservoir is already spiking in nature.

The extended platform has been verified through extensive testing, both in simulation and end-to-end on actual hardware.

Sammendrag

Etter hvert som datasystemer og nettverk vokser i størrelse og kompleksitet, er tradisjonelle topp-ned teknikker raskt blitt utilstrekkelige for å oppnå de ønskede resultatene. Å designe robuste systemer som er i stand til å tilpasse seg og selvregulere, som kan selvreprodusere og lære autonomt er en utrolig vanskelig oppgave. Disse egenskapene er imidlertid tilstede i mange biologiske organismer, hvor de fremkommer gjennom evolusjon og utvikling. I forskningsområdene ukonvensjonell og biologisk inspirert databehandling brukes disse teknikkene til å lage beregningsarkitekturer med samme type kompleksitet som det som finnes i biologiske systemer. Cellulære Automata (CA) er et eksempel på et biologisk inspirert datasystem som kan oppnå komplisert, global beregning gjennom lokalt samspill mellom enkle celler i stor skala.

Mens beregningsevnen til CAer har blitt forsket på over lengre tid, har de ikke blitt tatt i bruk som et beregningsmessig paradigme i stor skala. Programmerbarhet og koding/dekoding av data inn og ut er to store utfordringer for cellulære datasystemer. Manuell spesifisering av funksjonaliteten til hver celle på en slik måte at den ønskede fremtredende globale oppførselen til CAen som helhet oppnås, er praktisk ugjennomførbart for ikke-trivielle systemer. Input data til problemet er vanligvis kodet i systemets innledende tilstand, og data ut dekodes fra tilstanden etter at systemet har blitt simulert over tid.

Cellular Automata Research Platform (CARP) er en FPGA-basert implementasjon av en cellulær arkitektur, med sikte på å fasilitere forskning på bruk av kunstig utvikling og evolusjon for å skape cellulære datasystemer. På et abstrakt nivå implementerer plattformen et dynamisk system med dynamisk struktur $(DS)^2$, et system hvor både oppførsel og struktur er fremvoksende egenskaper. Atferd påvirker videre strukturell utvikling og omvendt.

I denne oppgaven blir plattformen utvidet til å inkorporere kunstig utviklede CAer i en reservoir computing arkitektur. Reservoir Computing (RC) er en ny tilnærming til maskinlæring hvor temporal inndata påføres som forstyrrelser på et dynamisk reservoar, og data leses ut ved å utføre en lineær klassifisering av reservoar-tilstanden en stund etter den første forstyrrelsen. Ved å kombinere RC og utviklings-CA, løser CARP-systemet mange av problemene knyttet til programmering av og I/O enkoding/dekoding av cellulare beregningsarkitekturer. Det åpner også nye muligheter for de utviklede strukturene til å tilpasse seg og lære basert på deres miljø. CARP-plattformen er utvidet med et rekonfigurerbart avlesingslag implementert som et spiking neural network (SNN) som klassifiserer dynamikken i reservoaret, en cellulær struktur under utvikling. En SNN er valgt for å tillate at systemet opererer helt i spiking-domenet, ettersom data inn og den dynamiske oppførelsen av reservoaret allerede er spikes.

Den utvidede plattformen har blitt verifisert gjennom omfattende testing, både i simulering og ende-til-ende på faktisk maskinvare.

Preface

This master's thesis has been conducted at the Department of Computer Science at the Norwegian University of Science and Technology under supervision of Professor Gunnar Tufte.

The thesis counts for 30 credits and is the continuation of a 15-credit specialization project conducted fall 2016. It concludes a 5-year master of science in Computer Science.

I would like to thank Gunnar Tufte for introducing me to an exciting field of research and for invaluable guidance and inspiration throughout the work on this thesis.

Øyvind Robertsen
2017 June 9th

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, research into computation using non-traditional physical mediums and paradigms, so called unconventional computing, has seen increased interest. With challenges currently facing traditional architectures, such as the von Neumann bottleneck and ensuring continued scalability and reliability, unconventional computating presents possible solutions from a new perspective. Biologically inspired computing is one approach to unconventional computating. It seeks to apply evolution, developement and other biological processes onto the design of both computer architectures and artificial intelligence systems. A common trait for many biological systems is that complex behaviour emerges from local interactions between simple units, whereas traditional computer architectures have been designed in a top-down fashion, composing complex modules and directing how information should flow between them. Alongside complex emergent behavior, biological systems often exhibit abilities such as self-reproduction, self-regulation and strong adaptability. These are desireable properties in computing systems, but they are hard to implement.

Cellular computing, introduced by Sipper in [34], is one example of a paradigm utilizing a bottom-up design methodology. Consisting of three core principles; simplicity, vast parallelism and locality, cellular computing seeks to harness the emergence of complex global behaviour from local interaction between simple cells at a large scale. One of the central problems with the paradigm is how one should go about designing/programming cellular systems. Specifying the functionality and potentially the connectivity of each cell manually while ensuring that the desired behaviour is achieved at a global level, is infeasible, so a different approach is needed. A potential solution is to automate the design through artificial evolution [35].

The POE-model, introduced by Sanchez et al [33], is a taxonomy commonly used to describe bio-inspired design methodologies using three categories: phylogeny, ontogeny and epigenesis. Phylogenesis relates to evolution, ontogeny encompasses systems that mimic biological development and epigenetic systems adapt to environmental change. These cat-

egories are not mutually exclusive. The use of artificial evolution to program a cellular computing system is an example of an phylogenetic system.

In artificial evolution, an individual represents a potential solution to some problem. In the case of using evolution to design cellular computing systems, the individual, or genome, has to encode both the behavior of each cell as well as the system, or organism, as a whole in order to represent a complete solution. This means that the size of the genome grows linearly with the size of the organism. In natural evolution, the genome has a different role. It serves as a set of rules governing the growth and development of cells at a local level, based on the types of surrounding cells and environmental feedback. This can be incorporated into the cellular computing paradigm as an ontogenetic aspect [41]. By evolving the developmental rules instead of the system as a whole, arbitrarily large and complex organisms can develop from a single cell based on a genome of fixed size. Systems that separate growth and behavior in this manner are called dynamical systems with dynamical structure (DS^2) [42].

The Cellular Automata Research Platform (CARP) is a long-running project at NTNU, dedicated to developing hardware that facilitates research into artificial evolution and development of cellular computing architectures. Based on the Virtual SBlock architecture presented by Haddow and Tufte [13], the system consists of programmable cells laid out in a regular one-, two- or three-dimensional grid, where each cell is connected to the cells in the von Neumann neighborhood around it. Cells can be in one of two states, either alive or dead. Based on their type, cells are programmed with a look-up-table (LUT) governing the transition between states based on the states of neighboring cells and the state of the cell itself. All cells are updated synchronously in discrete time steps. Development is simulated as a separate process, wherein cells transition between types using a LUT of development rules, taking both states and types of neighboring cells as input. This process also happens synchronously and in descrete development steps.

The CARP system is implemented on reconfigurable hardware, an FPGA, and is controlled by a program running on a host computer. Typically, the host program will implement the phylogenetic aspect of the system by evolving a population wherein each individual is a set of developmental rules. The CARP hardware is used to assess the fitness of each individual through development and simulation of the cellular organism.

Reservoir Computing (RC) is an exciting, new field of research within machine learning and intelligent systems. RC-systems work by imposing input data as perturbations on a dynamic system (the reservoir), and performing a linear classification of the reservoir state some time after the initial perturbation. Feedback from the classifier is routed back into the reservoir to allow it to regulate and adapt based on its own performance. In the specialization project leading up to this thesis, a proof of concept of a cellular reservoir and a readout layer implemented as a spiking neural network was simulated in software, with positive results. A common problem with cellular computing systems is that due to their very nature, it is often hard to formulate problems correctly and to interpret the dynamics of the system as answers to those problems. Combining the developmental, cellular architectures of the CARP system with the abstract computational concept of the RC paradigm will yield a more consistent framework for applying cellular computing to

real-world problems.

In this thesis, the CARP system has been extended to include epigenetic aspects into the design and development of cellular architectures. This is done by adding a trainable readout-module based on spiking neural networks which processes the dynamic behavior of cells in real-time and feeding its output back into the cellular reservoir. The system has also been ported to run on new hardware and the codebase has been partially ported to Chisel, a hardware definition domain specific language implemented in Scala.

1.1 Outline

This thesis is organized in the following chapters:

- **Chapter 2 - Background**: An overview of theoretical concepts on which the work presented in this thesis is built upon. Also gives an introduction to FPGA technology as well as an overview of some related work.
- **Chapter 3 - Previous Work**: A review of the history of the CARP project.
- **Chapter 4 - Platform**: Information regarding the physical hardware used to run the platform and the toolchains used to develop the project.
- **Chapter 5 - Implementation**: An overview of the implemented system and its constituent parts.
- **Chapter 6 - Verification**: Descriptions of tests used to verify system functionality.
- **Chapter 7 - Discussion**: A review of challenges with the system implemented in this thesis and possible future work.
- **Chapter 8 - Conclusion**: Conluding remarks.
- **Appendices**

Chapter 2

Background

2.1 Evolution

In nature, evolution is the process governing change and preservation of hereditary traits in populations of biological organisms. It allows species to adapt to their environment over generations through reproduction, variation and survival of the fittest [6].

Artificial evolution seeks to harness the powerful adaptive capabilities of natural evolution and apply them to general problem solving and learning. While research on the subject has branched into many different sub-areas, the general concept of optimizing a population of individuals with respect to some fitness function using mechanisms inspired by natural evolution, is referred to using the umbrella term Evolutionary Computation (EC) [2].

One of the greatest strengths of EC is how universally applicable it is. Evolutionary algorithms have successfully been applied to many different problem domains, such as robotics [10], bioinformatics [18], medicine [9] and many more.

2.1.1 Genetic Algorithm

The most common type of EA is the Genetic Algorithm (GA) [12]. GAs use, as shown in 2.1, genetic operators such as mutation, crossover and selection to evolve a population of potential solutions to a problem, subject to some fitness function. The fitness function deems how fit an individual is, and thereby how likely it is to be selected for reproduction. It is a measure of how well the indivual performs as a solution to the problem at hand. The algorithm continues until an individual with a fitness higher than some predetermined threshold is found. Individuals are represented as genomes, commonly with a genotype encoded as a bitstring which serves as a blueprint to create the solution, the phenotype.

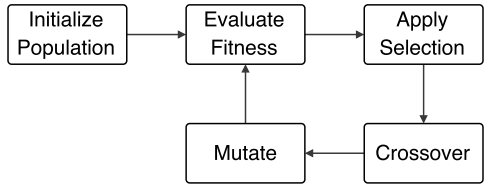


Figure 2.1: Genetic Algorithm process.

2.1.2 Genetic Programming

Genetic Programming (GP), introduced by John Koza [19], is a technique attempting to automate the programming of computers. This is done by evolving a population of programs whose fitness is evaluated by executing them and comparing to the desired results. Programs are encoded in genotypes as tree structures, as opposed to as binary strings. This allows crossover and mutation operators to be implemented in such away that the resulting programs are structurally sound.

2.2 Development

Biological development is the natural process that allows complex multi-cellular organisms to be built starting from a single cell using instructions encoded in the DNA of the organism. The most easily recognizable example is the development of humans from a single cell, the zygote, containing the combined genetic material of the parents, through cell division and differentiation. The human genome does not contain an exhaustive description and blueprint of each individual human. Rather, it consists of instructions governing how cells should divide and differentiate based on their surrounding cells and feedback from the environment.

Developmental processes in nature have many properties that make them desireable to mimic through artificial development. For instance, the number of cells in the human body is orders of magnitudes larger than the amount of information encoded in our DNA [3]. In general, the performance of GAs and GP implementations decline as the size of the genome increases, as mutations are more likely to be detrimental with regards to fitness. By introducing development as an indirect mapping between genotype and phenotype, programs and structures that scale to arbitrary dimensions can be produced while still maintaining a search space that the EA method in question can efficiently explore. Systems that combine evolution and development in this way are often referred to as EvoDevo systems [14].

Where evolution allows a species to adapt over the span of generations, development is an ongoing process throughout the lifetime of each individual, allowing for adaption based on changes to the environment [40]. This makes EvoDevo particularily well suited in the design of robust and adaptive artificial intelligence agents.

2.3 Cellular Computing

Most computing devices in use today have been developed on the foundation of the von Neumann architecture [44], a single complex processor performing one complex task at a time. Recently, the field of cellular computing has seen growing interest. Cellular computing, as described by Sipper in [34], is built on three principles: simplicity, vast parallelism and locality. It seeks to exploit emergent computational capabilities between large numbers of locally connected simple cells. Sipper presents cellular computing as an abstract framework, within which many variations of the paradigm can exist based on a number of properties. These include cell type (which types of values a cell can take; discrete or continuous), cell definition (how the behavior of cells is specified), cell mobility (wether or not cells can move within their environment), cell connectivity (how cells are connected to eachother; regular grid, (un)directed graph), topology of underlying environment (if any), connection lines (what information to transmit between connected cells), temporal dynamics (asynchronous vs. synchronous updating schemes), uniformity (in cell type and connectivity) and determinism. Some well known examples of paradigms that fit within the framework of cellular computing are Random Boolean Networks (RBNs) and Cellular Automata (CAs).

2.3.1 Cellular Automata

The most well known example of cellular computing is the Cellular Automaton (CA). Consisting of cells connected in a regular grid that transition between discrete states based on the states of the cells in their neighborhood [1]. While each cell is not capable of much on their own, the behavior emerging from interactions between cells can give rise to complex dynamics. Figure 2.2 shows the behavior over eight timesteps of a simple, 1-dimensional, uniform CA starting from a single cell, with each time-step shown on a new line. The rule governing state transitions is shown in the boxes to the right in the figure. Each box gives the neighborhood conditions on the top row and the resulting state on the bottom.

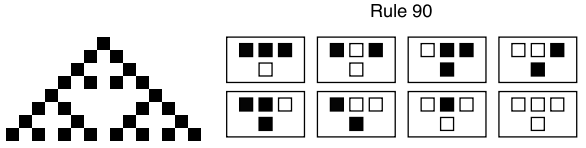


Figure 2.2: Eight timesteps for a uniform 1-dimensional CA. The boxes to the right show the rule with which the cells are configured, Rule 90.

Research into the computational capabilities of CAs can be said to have started in the 1940s, with John von Neumann and Stanislaw Ulam designing a 2D CA capable of self-reproduction [34]. Moving forward, the dynamics and behavior of specific CA rules was

[1] The von Neumann-neighborhood, consisting of cells directly north, east, south and west of a cell, is a common choice.

examined in detail. For instance, the rule used in Game of Life, introduced by mathematician John Conway in 1982, was proven to be capable of universal computation [5]. Stephen Wolfram shifted focus from studying specific rules, to investigating characteristics of groups of rules, resulting in his classification of the CA rulespace:

1. Rules leading to homogenous state for all cells. Regardless of the initial configuration of the cells, they all converge to the same state after a transient period.

2. Rules leading to stable or periodic structures.

3. Rules leading to chaotic patterns.

4. Rules leading to complex, long-lived structures. This is the only class that contains non-trivial automata.

Wolfram proposed that the rules capable of universal computation, such as Game of Life, reside in Class 4.

$$\lambda = 1 - \frac{q}{tot} \tag{2.1}$$

Like Wolfram, Christopher Langton has performed quantitative and qualitative studies of the CA rulespace [20]. He hypothesized that it is more likely to find rules capable of complex computational behavior in regions of the rulespace where there is a phase transition between ordered and chaotic dynamics. He introduced the λ-parameter as a measure of heterogeneity for a rule. It is calculated, as shown in 2.1, where q is the number of transitions in a rule that lead to a chosen quiescent, or dead, state, and tot is the total number of transitions. For a CA with N possible states and a neighborhood-size of K, the total number of transitions is K^N. A λ-value of 0 indicates an entirely homogenous rule, where all possible neighborhood configurations result in a transition to the quiescent state. Maximally heterogenous rules will have a λ of $1-1/N$. Figure 2.3 shows how Wolframs 4 classes map onto Langtons λ-space, with Class 4 coinciding with phase transition between ordered and chaos behavior, the so called Edge of Chaos.

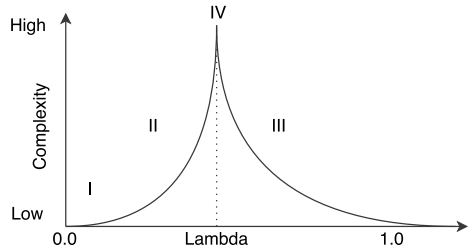


Figure 2.3: Wolframs rule classes mapped onto Langtons λ-space.

2.3.2 Developmental Cellular Architectures

One of the major challenges faced by cellular computing systems is how to program one. Manually programming the functionality and connectivity of each cell to achieve the desired emergent properties is both exceptionally time-consuming and hard to do when the problem to be solved involves global coordination. Sipper proposes to automate the programming of cellular systems through an adaptive process such as an EA. In [27], Mitchell et al. use a GA to evolve CA rules for solving problems requiring global coordination. Each individual is a bitstring representing the next-states for all possible neighborhood configurations. With a neighborhood size of 7, each genome is $2^7 = 128$ bits long yielding a search space size of 2^{128}. The authors are able to successfully evolve CAs that solve the problems for which they where created.

A different approach is taken by Sipper [35] in order to evolve non-uniform CAs. Where Mitchell et al. evolve a population of rules, doing this for a non-uniform CA would require an exceptionally large genome in order to specify the rule for each cell, increasing the search space exponentially, making it infeasible to search through using a standard GA. Sipper instead works with a single CA initialized with a random rule in each cell. The fitness of each cell is accumulated over some number of simulations of the CA starting from different initial state configurations, after which evolutionary mechanisms are applied in a local manner, between connected cells. Through this method, Sipper is able to evolve non-uniform CAs that outperform the uniform ones evolved by Mitchell et al. on the same tasks.

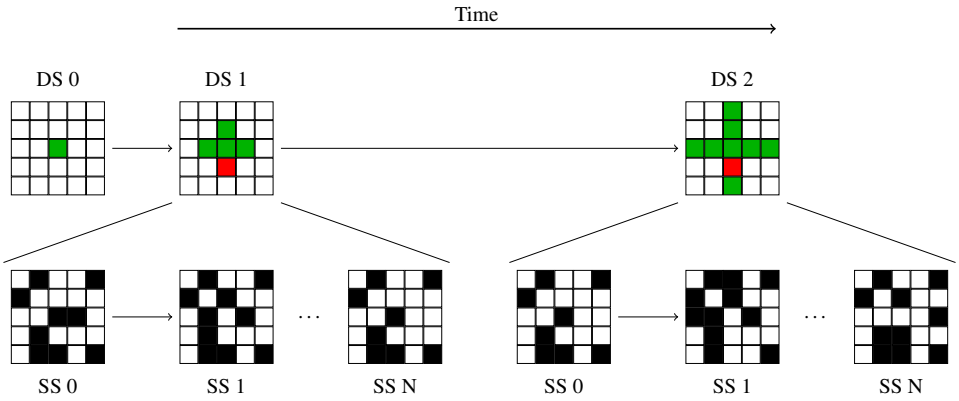


Figure 2.4: Development starting from a single green cell using the growth rule in Figure 2.5.

A third approach to the design of cellular computing systems is presented by Haddow and Tufte in [41]. The authors use a developmental model to allow complex non-uniform CAs to grow from a single cell, as shown in Figure 2.4. Development occurs in discrete time steps, so called development steps (DS). Between each DS, a number of state steps (SS) occur, simulating the behavior of the organism developed so far as a CA. The rules governing the development process, the genome, take both cell type and state into consideration when deciding how to proceed. This means that the behavioral dynamics of the organ-

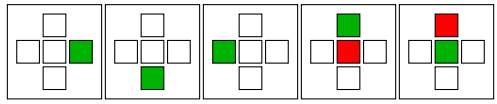


Figure 2.5: Growth rules for a cellular developmental system where cells are either empty or of the green type.

ism being developed regulates the developmental process. This type of coevolution of both structure and functionality is often referred to as dynamical systems with dynamical structure $(DS)^2$ [37].

While the three approaches to programming of cellular computating systems outlined above achieve positive results, the task is still hard. In the case with the developmental approach taken by Tufte, creating developmental rules is as difficult as manually specifying type and functionality for each cell. Combining development with evolution is possible [29], but the genome size needed to express all regulatory possibilities in the genome is so large that the resulting size of the search space makes it hard for the EA to reliably converge. There is in other words still a ways to go before programming of cellular computing systems can be considered a solved problem.

Another challenge faced by such systems is how to formulate problems and encode/decode their inputs and outputs. In the works by Sipper and Mitchell, problem input is encoded as the initial state of the CA, and the output is interpreted from the state of cells after a number of state steps. This approach takes away from the generality of cellular systems, as an evolved/developed system will only be able to work specifically for the problem and encoding scheme it was initially designed around. Adapting a system to apply to different problems or to slight variations in input encoding will almost always require it to be developed from scratch.

2.4 Reservoir Computing

Artificial Neural Networks (ANNs) are a commonly used computational model in machine learning and bio-inspired computing. Simple, feed forward ANNs lend themselves well to problems were data can be spatially correlated, such as classification. Many real world problems however, are temporal in nature. Recurrent neural networks (RNNs) have been shown to be powerful tools for solving temporal problems such as stock market prediction [21], learning context free/sensitive languages [11] and speech synthesis [45]. Training RNNs is computationally expensive and often requires application specific adaptions of generalized training algorithms in order to reliably converge [15]. Several techniques have been proposed that circumvent problems related to training, such as Echo State Networks [16] (ESNs), Liquid State Machines [24] (LSMs) and Backpropagation Decorrelation learning [38]. These all share the common feature of only training weights of the output layer of the network, while leaving the hidden layers of the network untrained or

simply subject to weight scaling. In [43], Verstraeten et al. propose that systems based on this idea should be unified under the term reservoir computing (RC).

In general, reservoir computing as a term describes any computational system where a dynamic reservoir is excited by input data and output is generated by performing classification/regression over reservoir state. Figure 2.6 shows the basic architecture of any reservoir computing system. With its origins in research on various types of recurrent neural networks and training thereof, the reservoir in RC systems is often represented as an RNN [43]. However, any dynamic system capable of eventually forgetting past perturbations and of responding distinctly to different perturbations, can in principle be used. Snyder et al. [36] investigate using Random Boolean Networks, Yilmaz uses Cellular Automata [46] and Fernando et al. use a bucket of water [8].

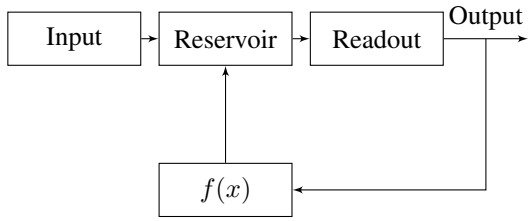


Figure 2.6: Basic overview of an RC system.

As outlined in Section 2.3 cellular computing systems are capable of complex, vast parallell computing. Combined with artificial development they are also highly adaptive. In this thesis we combine reservoir computing and developmental cellular automata in an attempt to provide a framework allowing for easier use and development of cellular systems. Figure 2.7 shows how input data perturbs the behavioral part of the reservoir, the emerging CA, and output is extracted by using the readout layer to classify the dynamics in the CA. RC systems provide a layer of abstraction between I/O encoding/decoding and the computation occurring in the CA. With this approach the goal of development and evolution is not to create a CA configuration that is able to solve a specific problem under specific I/O conditions, but rather to develop CAs with strong general computation capabilities and strong ability to adapt to different input perturbations. A CA that is general enough, could be used in many different contexts simply by swapping out the readout layer.

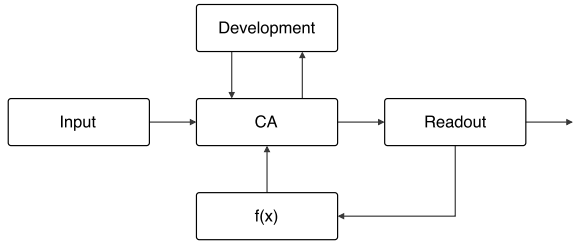


Figure 2.7: RC system with developmental cellular reservoir.

2.4.1 Spiking Neural Networks as Readout Layers

Artificial neural networks can be grouped into three generations, based on the characteristics of their base computational unit, the neuron. The first generation, based on McCulloch-Pitts neurons [26], simple threshold gates, allows for universal computation on digital input/output values. In the second generation, neurons apply a non-linear, continuous activation function on the weighted sum of their inputs.

The third generation of networks bases itself on spiking neurons, which model the interaction between biological neurons more closely. In this model, a neuron v fires when its potential P_v exceeds a threshold θ_v. The potential is, at any time, the sum of the postsynaptic potentials, resulting from firing of presynaptic neurons. The contribution of a spike from presynaptic neuron u at time s to the potential P_v of postsynaptic neuron v is given by $w_{u,v} \cdot \varepsilon_{u,v}(t-s)$, where $w_{u,v}$ is a weight representing the strength of the synapse connecting u and v, and $\epsilon_{u,v}(t-s)$ models the response of the spike as a function of time passed since the spike occurred. A synapse can be both excitatory and inhibitory, meaning that its contribution to the total potential P_v can be both positive and negative. A biologically plausible response function is shown in figure 2.8. From a machine learning perspective, the trainable part of a spiking neural network, is the weight $w_{u,v}$, determining to what degree spikes from a neuron u influences the potential of neuron v.

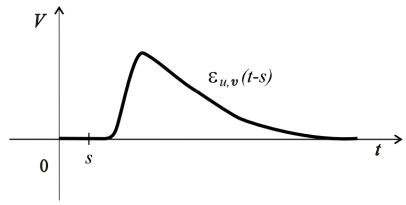


Figure 2.8: Common spike response function shape, figure taken from [23].

Spiking Neural Networks (SNNs) are of particular interest in the context of a cellular RC system, where the reservoir dynamics are spiking in nature (i.e. a cell can be either alive or dead). By using a spiking neural network as a readout layer, data can flow through the RC-system as spikes from end to end. In the specialization project leading up to this thesis, experiments to examine the viability of SNNs as readout layers in RC system were carried out with successful results (see Appendix C).

2.5 Related Work

2.5.1 IBM Truenorth

The IBM Truenorth is a modular, non-von Neumann, ultra-low power computer simulating a massive network of biologically plausible spiking neurons [2]. It consists of 4096

[2] `http://www.research.ibm.com/articles/brain-chip.shtml`

neurosynaptic cores, each simulating 256 neurons and 256×256 synapses, resulting in a total of 1 million neurons being simulated. The cores are connected using an on-chip mesh network, allowing the core-count to be scaled without adding extra circuitry. During operation the platform consumes < 100 mW and is capable of 46 billion synaptic operations per second, per watt.

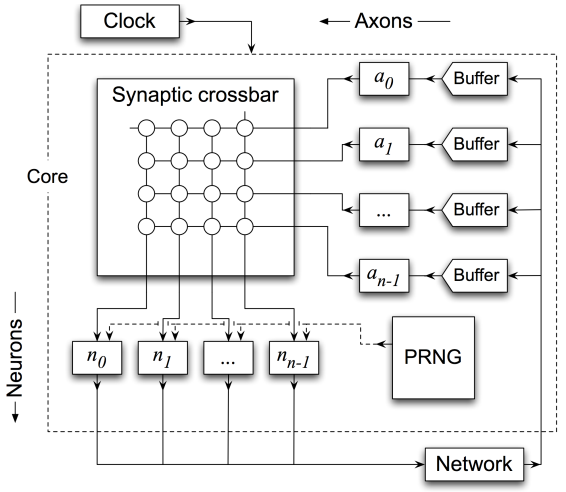


Figure 2.9: Conceptual architecture of a neurosynaptic core used in the TrueNorth chip. Reprint from [30].

IBM has implemented an ecosystem of algorithms, libraries, simulators, a programming language and an integrated development environment to support the platform. Several applications for the chip has also been developed, such as a multi-object detection and classification system operating on 240×400 pixel, 3-color video input at 30 frames per second.

2.5.2 Tensor Processing Unit

To accelerate training and utilization of machine learning models implemented using their TensorFlow[3], Google have designed an Application Specific Integrated Circuit (ASIC), the Tensor Processing Unit (TPU)[17]. Designed to provide exceptionally fast matrix mutliplication, the TPU is built around a Matrix Multiply Unit (MMU) capable of 2^{16} 8-bit multiply-and-add operations on signed/unsigned integers per cycle. The rest of the logic on the chip is dedicated to moving and organising data in such a way that the MMU is maximally utilized.

When compared with its contemporary CPU and GPU competitors, the TPU operates approximately 15-30x faster, and with 30-80x higher TeraOps/Watt.

[3] `https://www.tensorflow.org/`
[4] `https://cloudplatform.googleblog.com`

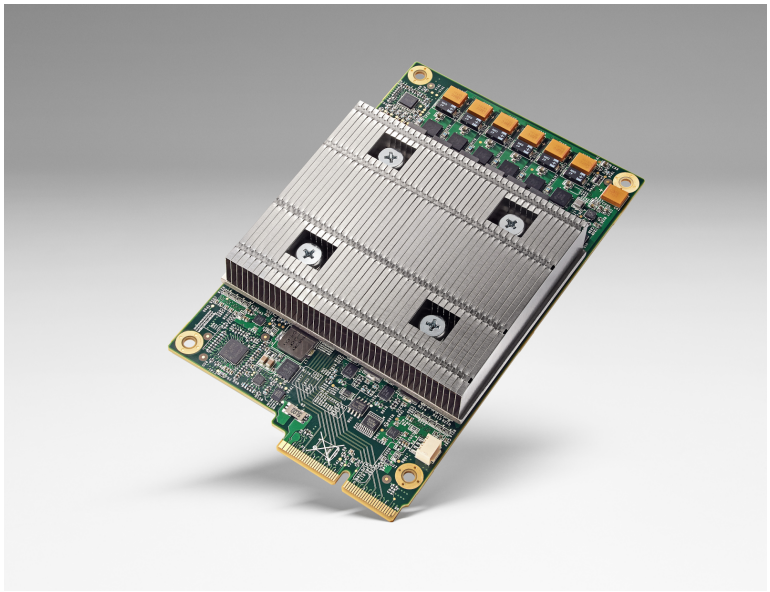


Figure 2.10: The Google Tensor Processing Unit. Reprinted from [4]

Chapter 3

Previous Work

The work presented in this thesis extends the Cellular Automata Research Platform (CARP), an FPGA-based system implemented specifically to support and accelerate research into growth and evolution of developmental systems based on cellular automata. The original implementation was done by Djupdal in 2003. Over the years, the system has been extended as well as optimized to run on newer hardware through a series of master theses. The following sections provide an overview of the evolution of the system.

Figure 3.1 shows the original overall design of the system, indicating how the host program is responsible for evolving genotypes, while developing genotypes into phenotypes and simulating their behaviour in the CA is implemented on the FPGA.

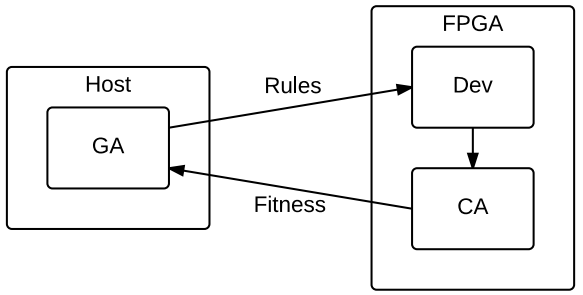


Figure 3.1: General system design. Figure reprinted from [22].

3.1 Djupdal

The original design of the CARP system was made by Djupdal in 2003 [7], to support further research into elvolvable hardware based on the SBlock-architecture proposed by Tufte

and Haddow [13]. The system was implemented on a NallaTech BenERA FPGA-board communicating over a Parallell Component Interconnect (PCI) bus with a CompactPCI host-computer.

Figure 3.2 shows the overall architecture of the resulting hardware platform. It consists of the SBlock Matrix (SBM), Block RAM (BRAM) for storing the state and type of each cell, a development unit, control logic, and a PCI communication unit.

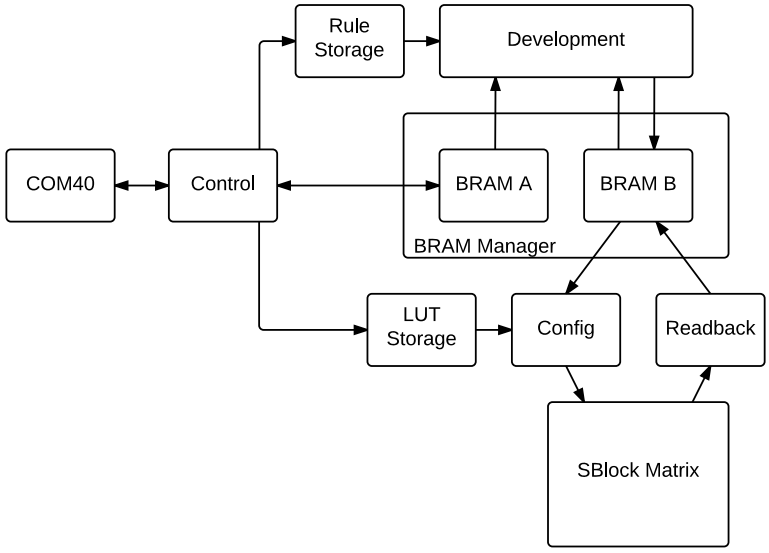


Figure 3.2: High-level block diagram of the hardware platform as implemented by Djupdal. Figure taken from [22].

A host computer, running a genetic algorithm exploring the space of possible development rules, controls the system. Each genotype is transferred to the system and developed into its phenotype before the SBM is stepped some number of times. New states and types can then be transferred back to the host to calculate a fitness score.

A genotype consists of a set of initial cell states and types, development rules and LUTs corresponding to the cell types possible. Upon initializing the system with a new genotype, states and types are written to BRAM A, while development rules and LUTs have their own separate BRAMs. During development, cells are read from BRAM A, tested against development rules and written back to BRAM B, now possibly changed as a result of "hitting" a rule. The development unit tests 8 rules on 2 cells each cycle in raster order. This means that for sets of rules larger than 8, several sweeps over the cells is necessary. For these additional sweeps, cells are read from BRAM B so as not to overwrite the result of a rule hit in a previous sweep if no rules hit in the later ones. The two BRAMs can be swapped logically, avoiding having to transfer between the two in order to start a new development step.

Based on the cell types stored in BRAM A, the SBM can be configured. Each cell type

corresponds to a LUT stored in the LUT storage BRAM, with which the SBlocks corresponding to cells of that type is configured. State steps can then be performed either one at a time, writing each new set of states back to BRAM B, or in batches, writing only the final set of states back.

3.2 Aamodt

In Djupdal's design it was necessary to transfer cell states to the host to calculate fitness. To avoid this bottleneck, Aamodt extended the system with an on-board fitness module in 2005 [1]. Additionally, to gain further insight into the development process, he added logging-modules for the development unit.

Figure 3.3 shows the overall system as implemented by Aamodt. The modules added are as follows; a Run-Step Function (RSF) that calculates the number of live cells, a BRAM module to buffer these numbers, a fitness function and the two logging modules from the development unit.

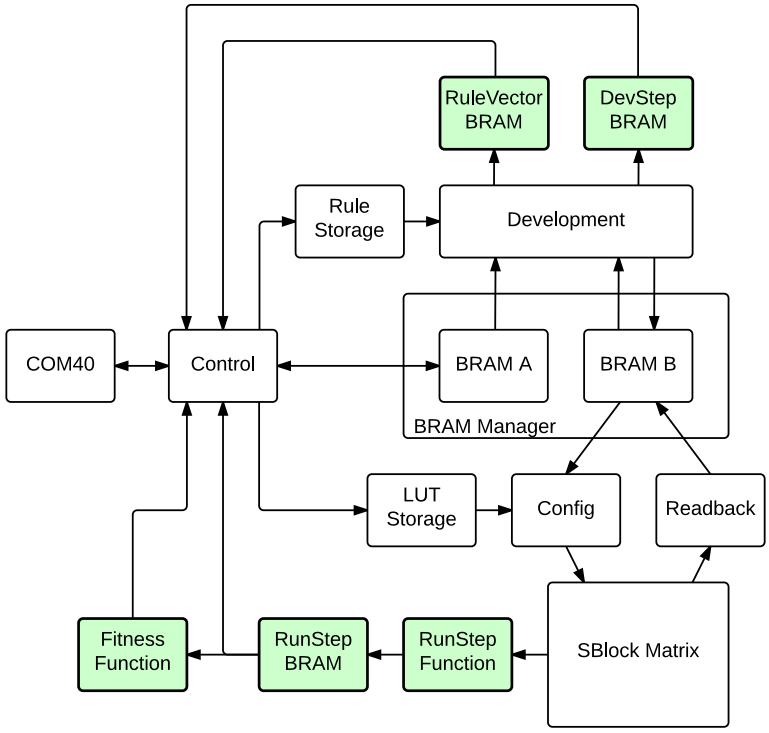


Figure 3.3: High-level block diagram of the hardware platform after Aamodt's work. Additions are highlighted in green. Figure taken from [22].

The two BRAMs added to the development module logs information regarding the devel-

opment process. The Development Step BRAM stores which rules were triggered for all cells during the most recent development step, while the Rule Vector BRAM stores which rules were triggered overall for the last 256 development steps. The RSF is a large adder tree, calculating the total number of live cells in the SBlock matrix after each state step. These totals are buffered in the RunStep BRAM before being processed by the fitness function.

3.3 Støvneng

In 2014, the system was further extended and optimized by Støvneng [39], in expectation of new FPGAs aquired by NTNU. In addition to rewriting existing modules to better utilize the resources available on the new chips, he also modified the SBlock matrix to allow for 3D CAs and implemented an on-chip Discrete Fourier Transform (DFT) for processing the cell count from the RSF-module into the frequency domain.

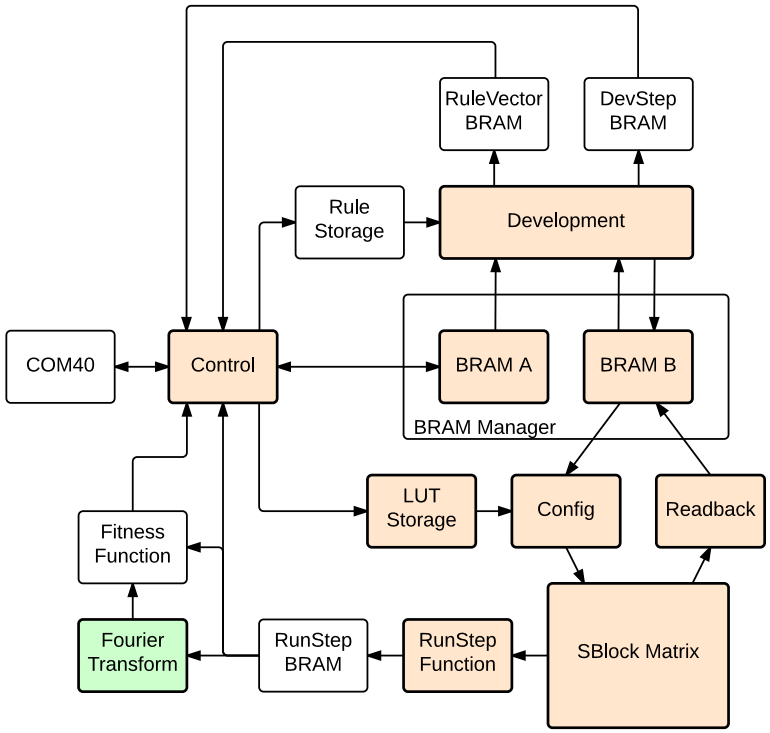


Figure 3.4: High-level block diagram of the hardware platform after Støvneng's work. Additions are highlighted in green, and optimizations and 3D modifications in orange. Figure taken from [22].

Figure 3.4 shows how the DFT module is added to the overall design, as well as which modules have been optimized and rewritten to support 3D CAs. Overall, the optimizations made to the system yielded a 4x speedup for most operations. The design as a whole was

also made more parameterized and portable by removing many hard coded constants and rewriting modules relying on features specific to the old FPGA.

As the new hardware did not arrive in time, Støvneng was unable to test his changes on the actual FPGA, but the design in its entirety was verified through simulation.

3.4 Lundal

After Støvneng finished his work, it became clear that the new hardware would be delayed yet again. To allow for further development and testing of the platform on actual hardware, a development board with a similar FPGA as the one in the anticipated new hardware was ordered. As this newer line of FPGAs has removed support for PCI in favor of PCIe, both the communication module on the hardware platform and the software side driver had to be updated. While testing the new communication module, several issues with the current implementation were uncovered, such as commands not functioning according to specification and extensive use of outdated FPGA features. This led to the decision of rewriting the platform from scratch, a job undertaken by Lundal in 2015 [22].

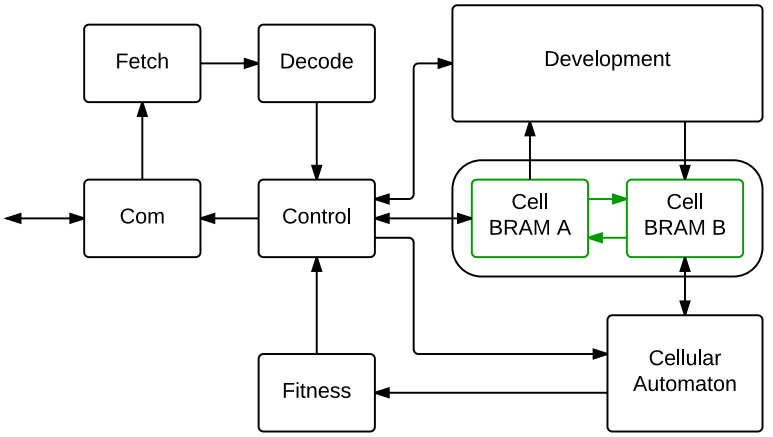


Figure 3.5: Overview of the reorganized hardware platform as implemented by Lundal. Figure taken from [22].

Through his work, Lundal focused on making the hardware platform more modular, easier to configure and coherently structured. All dependencies on specific hardware features were removed in favor of letting the synthesizer infer based on which resources are available on the targeted chipset. Support for both 2D and 3D SBMs was implemented by Aamodt as two separate designs. Lundal unified this into one design, utilizing the fact that all 2D CAs can be implemented as one-high 3D CAs. The software API was also made more complete and user-friendly, by implementing abstractions that allow the end user to focus more on the code related to the experiment being performed and less on the technicalities necessary to operate the hardware platform.

Since Djupdal's original design, the development module has remained largely unchanged, implementing a developmental model based on research by Haddow and Tufte [41]. In this model two types of developmental rules are considered; growth-rules indicating how an organism develops spatially, and change rules describing how existing cells can change their type. Both rule types consist of a condition and a result. Conditions take into account the types of the cells in the von Neumann neighborhood around the target cell and the type of the target cell itself. The result of a growth rule is the direction in which the target cell should grow. For a change rule, the result is the new type of the target cell. Change rules can only affect cells that have already been grown into and growth rules can only grow into cells that are empty.

Lundal implemented a simpler development module based on Tufte and Nicheles research [28]. Here, all rules are effectively change rules and all rules are evaluated for all cells. The functionality of growth rules in the old design can still be implemented within this scheme. Where the differentiation of growth and change in the old design is closer to the type of cell development that happens in biological systems, the new system is more applicable to generic dynamical systems.

Lundal verified the reimplemented CARP platform both through simulation and end-to-end integration tests with the synthesized design .

Chapter 4

Platform

In 2016, the long anticipated new hardware was finally installed and was ready to be used for the CARP project. The new machine contains four Convey Wolverine Application Accelerators, coprocessors aimed at accelerating key parts of algorithms in high-performance computing through reconfigurable hardware. Each accelerator is equipped with a state-of-the-art Xilinx FPGA and large, high-bandwidth on-chip memory.

The following sections describe the Wolverine accelerator architecture, the toolchain used to synthesize the hardware design and finally the toolchain used for the software API.

4.1 Convey Wolverine WX-2000 Application Accelerator

The Convey Wolverine WX-2000 is a PCIe-mounted coprocessor equipped with a Xilinx Virtex-7 XC7V2000T FPGA and four DDR3 SO-DIMM slots allowing for up to 64 GB of on-chip memory. Figure 4.1 shows how a Wolverine coprocessor is organized at a high level. The host computer communicates with the card over a Peripheral Component Interconnect Express (PCIe) bus at a max bandwidth of 8 GB/s. On the coprocessor, the Host Interface controller (HIX) is responsible for decoding the data sent from the host. Data can either be stored in the on-board memory, or passed directly to the FPGA. The FPGA is divided logically into Application Engines, the number of which can be configured before synthesis. This is the entrypoint into which a custom FPGA design is inserted. The Convey Personality Development Kit (PDK) provides each AE with interfaces to the HIX, the AEs memory controller and the other AEs.

With 2 million logic cells, 46 MB of BRAM and up to 2.8 Tb/s serial bandwidth, the XC7V2000T is one of Xilinx' higher end FPGAs. Earlier, FPGAs have been scaled monolithically following Moore's Law, in the same way conventional processors have been scaled. With the XC7V2000T however, Xilinx has opted to combine 4 separate dies

into one large virtual FPGA with their Stacked Silicon Interconnect (SSI) technology [32]. Each die, or Super Logic Region (SLR), has its own clocking and configuration circuitry. For monolithic designs, these signals would have to be routed throughout the entire die in complex ways to avoid critical paths that are too long. With this circuitry replicated in each SLR, the resources required for routing clocking and configuration signals is significantly lower, opening up the possibility to use these resources to interconnect the SLRs instead. This, along with advances in manifacturing techniques has allowed Xilinx to scale their FPGAs even further and opening up new use cases for them. Programming FPGAs using SSI is no different from any other FPGA. The Xilinx design flow toolchain distributes designs across multiple SLRs if needed.

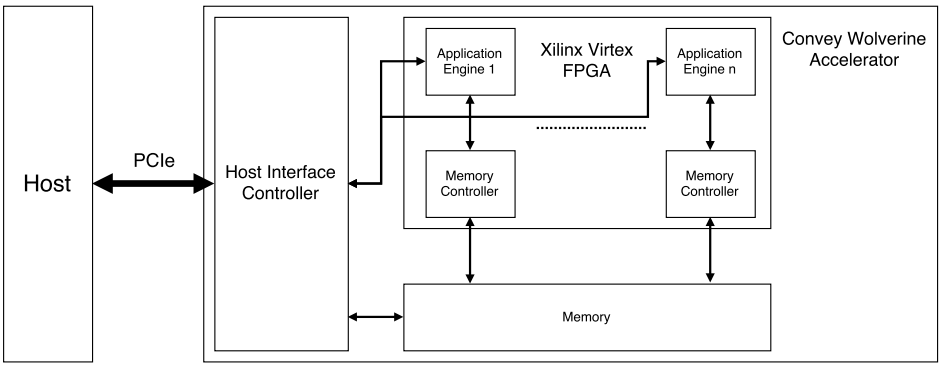


Figure 4.1: High level overview of the Convey Wolverine Accelerator architecture.

4.2 Hardware Toolchain

All previous iterations of the CARP hardware platform have been implemented entirely in VHDL. At the start of this project, the decision was made to start porting the codebase to Chisel, a hardware construction DSL implemented in the Scala programming language.

4.2.1 Chisel

Chisel[1] is an open-source hardware construction language developed at UC Berkeley. Where many other hardware design tools implemented in high-level programming languages are of the "C-to-gates" variety, tools that try to automagically infer hardware based on a description of the desired computation, Chisel is based upon using the computational tools in Scala to describe how a circuit should be wired. Leveraging Scalas typesystem and functional programming tools, Chisel encourages code reuse, genericity and designing systems that are highly parameterizable.

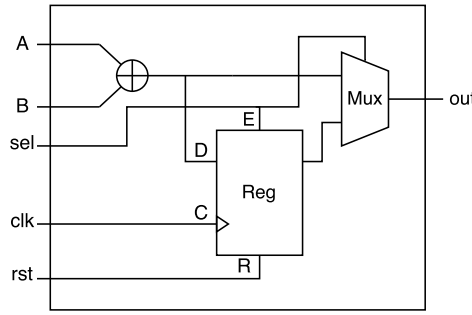
[1] `https://chisel.eecs.berkeley.edu/`

```
import chisel._

class Sample(w: Int) extends Module {
  val io = new Bundle {
    val a   = UInt(INPUT, width=w)
    val b   = UInt(INPUT, width=w)
    val sel = Bool(INPUT)
    val out = UInt(OUTPUT, width=w)
  }
  val stored = Reg(init=UInt(0))
  val sum = io.a + io.b

  when (sel) {
    io.out := stored
  }.otherwise {
    stored := sum
    io.out := sum
  }
}
```

(a) Chisel source



(b) Circuit

Figure 4.2: Chisel source and corresponding circuit.

Listing 4.2a shows how to implement the circuit in Figure 4.2b. The `io` bundle defines the inputs and outputs of the module. In this case a boolean input signal and three unsigned integer signals, two inputs and an output, all of whose bitwidth is determined by the `w` parameter passed to the module when it is instantiated. `val s = Module(new Sample(32))` will create an instance of the `Sample` module with 32-bit wide `UInt`s.

When executed, a Chisel program creates a internal graph representation of the circuit described by the program. Depending on the parameters given to the program, the program will output either Verilog or a C++ simulator of the circuit. To create a fully functional FPGA-image the Verilog output can be incorporated into an FPGA design flow.

To avoid having to start entirely from scratch, it was decided that porting the CARP hardware platform to Chisel would be done in a top-down fashion. That is, starting by instantiating Lundal's toplevel module as a blackbox in the Chisel design and implement downward in the module hierarchy as needed when extending the functionality of the platform. Figure 4.3 gives an overview of the build process. All parameterization is gathered in the Chisel part of the design and passed on into the VHDL modules when they are instantiated in the resulting Verilog.

For further details regarding the development setup for the hardware part of the CARP project, see Appendix D.

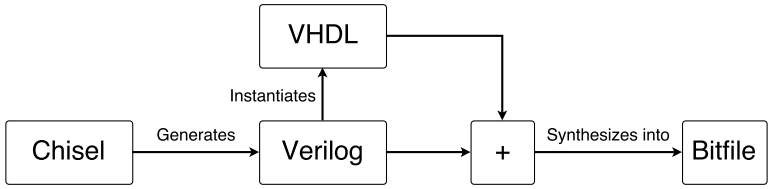


Figure 4.3: Build process overview.

4.3 Software Toolchain

The software API is written in C, compiled on CentOS 6.5, Linux kernel version 2.6.32-431 with GCC version 4.4.7. Outside of the C standard library, the API has only one dependency; the Convey Personality Development Kit. It is included via the `wmd_user.h` header and provides functions and routines related to communicating with the Convey coprocessor.

Chapter 5

Implementation

Figure 5.1 shows a high level overview of the CARP hardware platform extended with a readout module. While the system has been extended with new functionality and partially ported to Chisel, the overall architecture of the system has been largely preserved. In Figure 5.2, a more detailed view of the system is shown. Modules are annotated with either a C(hisel) or a V(HDL) in the upper right corner to indicate their porting status. The top level logic and wiring gluing all the modules together is all done in Chisel.

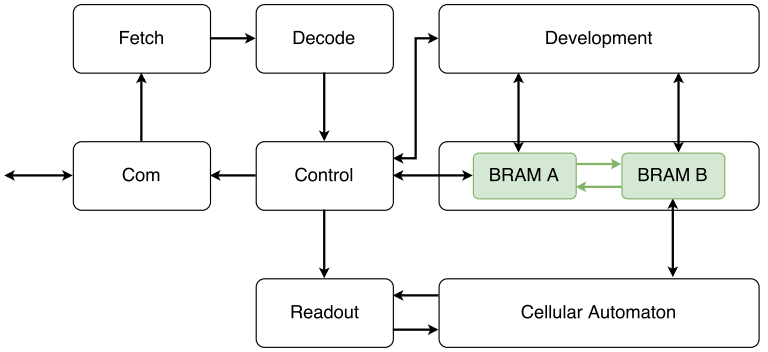


Figure 5.1: Block diagram of the CARP hardware architecture extended with a readout layer.

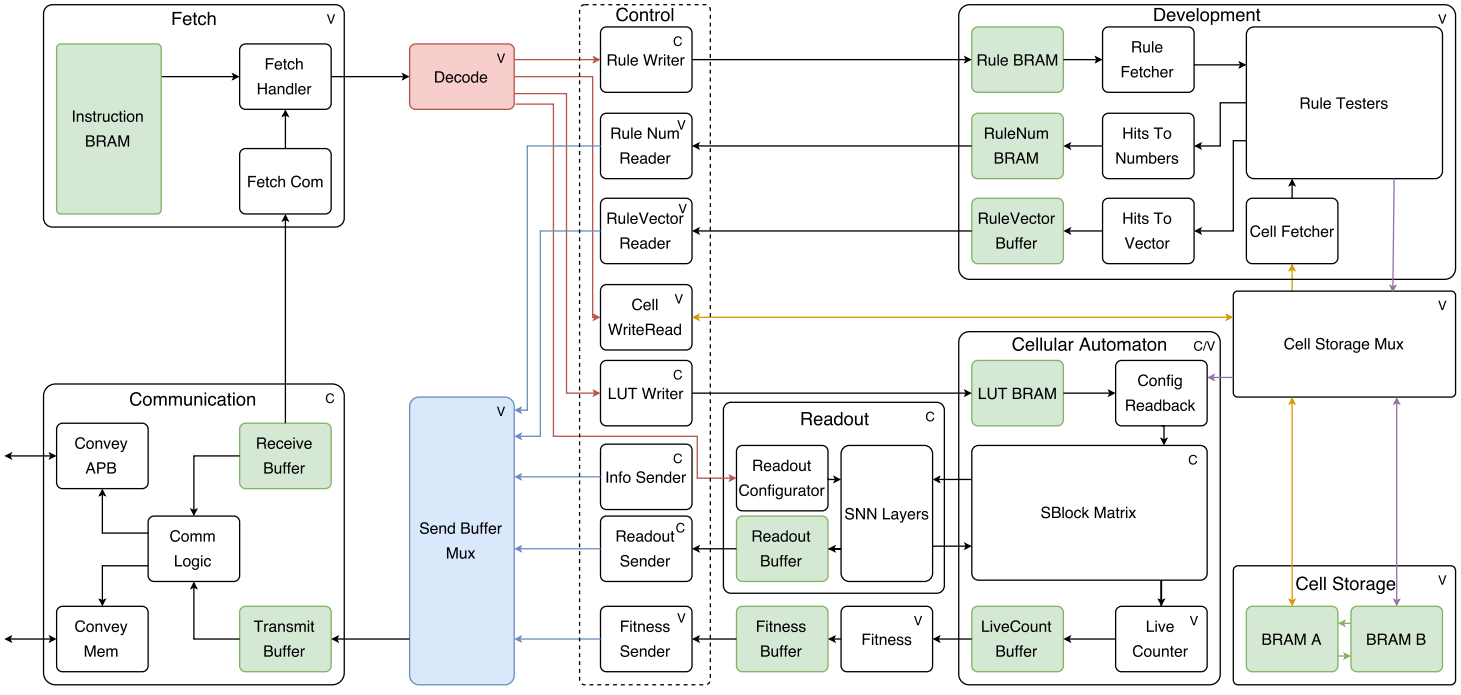


Figure 5.2: Modified reprint from [22] showing a detailed overview of the CARP architecture and its constituent modules. Modules annotated with a C in the upper right corner are implemented in / have been ported to Chisel, while those annotated with a V are implemented in VHDL. Child modules not annotated are implemented in the same language as their parent modules. Signals indicate flow of data or direction of communication, control signals and detailed interfaces are ommited for clarity.

5.1 General overview

The CARP hardware system executes instructions in a a Fetch, Decode, Execute loop. Instructions transferred from the host program controlling the coprocessor are placed in the Receive Buffer. The Fetch module reads instructions one at a time from the Receive buffer and either stores them for later use in the Instruction BRAM, or passes them on to the Decode module. The Decode module is responsible for parsing instructions, extracting parameters and orchestrating control signals throughout the rest of the system. In the Execute phase, different modules perform work depending on the type of instruction. The major players in the Execute phase are the Development, Cellular Automaton and Readout modules.

The system operates in a pipelined fashion, with Fetch, Decode and Execute being separated stages. Since few of the modules in each stage complete in the same number of clock cycles, modules are interlocked to ensure consistency and to avoid hazards. At the top level, this is implemented using two signals, Run and Done. Each module receives the Run signal as an input, indicating wether or not it is safe to execute. When a module is completes its work, it sets its Done signal high. To interlock a group of modules, the run signal is determined by the logical And operation of the done signals of all the modules.

Cell types and states are stored in the Cell Storage BRAMs, ready to be used either in development, where they are used to determine further growth and change to the organism they constitute, or in the cellular automaton, which simulates the behavior of the organism.

The following sections describe the additions and changes made to the system to add support for a reconfigurable Readout module based on a spiking neural network.

5.2 Communication

As outlined in section 4.1, the FPGA on the Convey coprocessor is logically divided into Application Engines within which any custom logic can be implemented. Each AE can communicate with the host, the on-board memory and other AEs through various interfaces implemented by the Convey PDK, shown in Figure 5.3. To be able to run the CARP hardware platform on the coprocessor, the communication module has been reimplemented to utilize these interfaces for data transfer.

Figure 5.4 shows how the CARP hardware is implemented within the Convey AE architecture. The communication module has been moved out of the main CARP module and rewritten from scratch. In an effort to decouple the CARP platform from the underlying hardware and its communication interfaces, the new module exposes two very generic interfaces facing "outward," an Advanced Peripheral Bus (APB) [1] and a memory bus, as shown in Figure 5.5. These are specified fully in Appendix A. Using generic interfaces with established conventions that are easy to connect to other communication interfaces

[1] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0024c/index.html` (Requires registration)

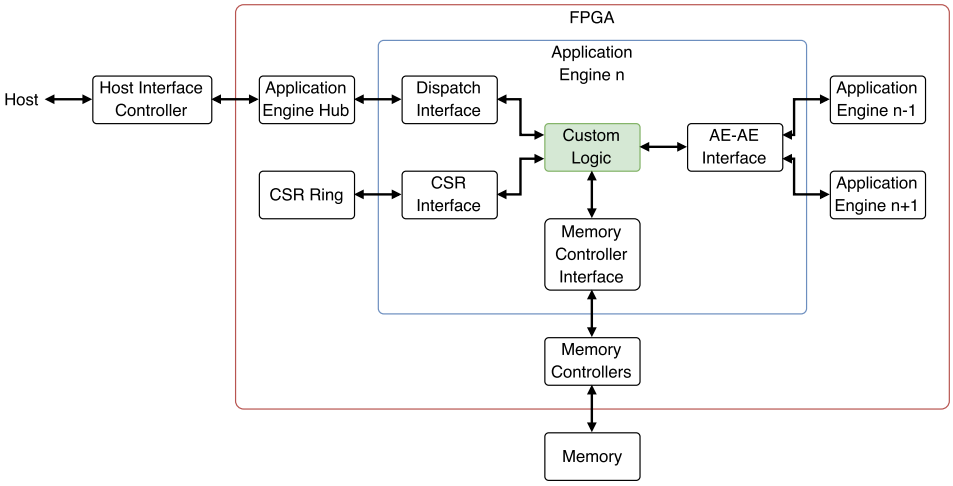


Figure 5.3: Overview of the Convey Application Engine architecture.

makes it easy to move the system to a different platform, should the Convey coprocessors become obsolete or defunct.

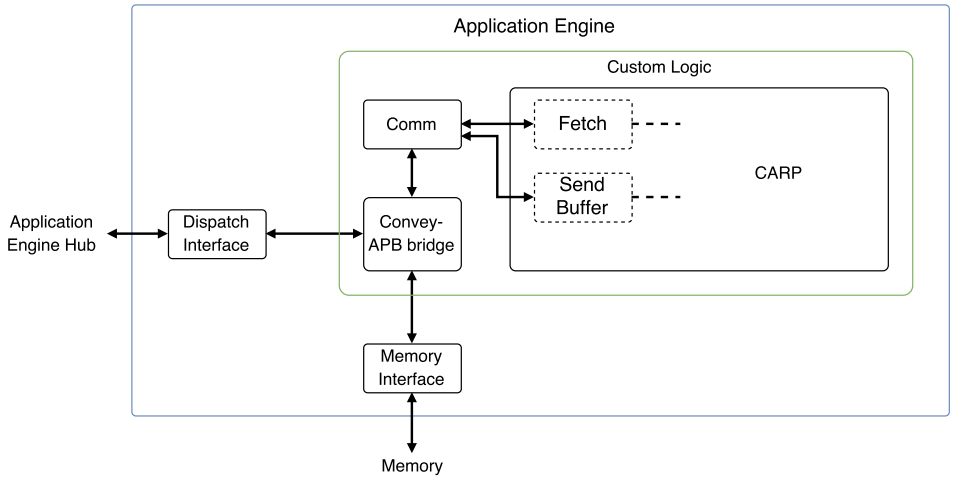


Figure 5.4: Implementation of the CARP platform within the AE architecture.

To avoid having to rewrite the Fetch and Send Buffer Mux modules inside CARP, the new communication module provides the same interface towards them as the old one did; two data buffers, transmit and receive, buffer count signals and read/write enable signals. Both data buffers are implemented as FIFO-queues with counter registers and ready-valid access interfaces.

Figure 5.6 shows the state machine controlling the operation of the communication mod-

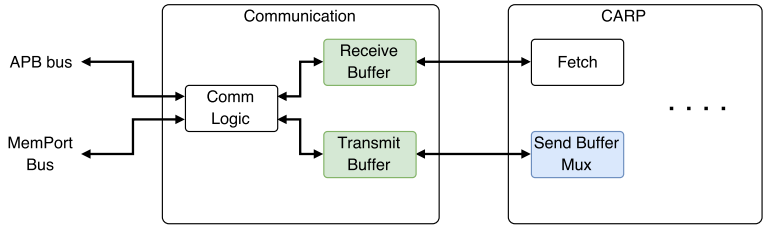


Figure 5.5: The communication module interfaces with the CARP platform through two buffers, transmit and receive. Outwards, the communication module exposes two buses, an APB bus and a generic memory interface.

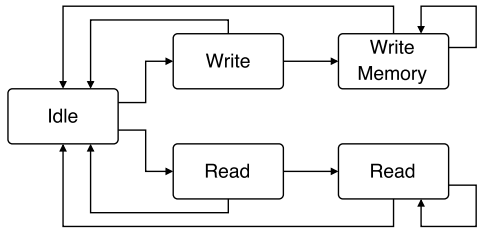


Figure 5.6: State machine controlling the communication module.

ule. From the idle state, a transition to either the Write or Read states can be triggered by asserting the PSEL signal on the APB bus. Depending on wether or not the PWRITE signal is asserted, the state machine will transition into the corresponding state. In this case, write refers to writing data from the host to the receive buffer, and read refers to reading data from the transmit buffer to the host. In the Write state, three bits in the PADDR signal is used to further determine what is to happen. The possible actions are as follows:

1. Write PWDATA to Receive Buffer. Transition to Idle state.
2. Set starting address register to PWDATA. Transition to Idle state.
3. Set end address register to PWDATA. Transition to Idle state.
4. Prepare to start transferring data from memory to Receive Buffer. Transition to Write Memory state.

In the Write Memory state, read requests are generated on the MemPort bus and received data is stored into the Receive Buffer. The Read and Read Memory states have similar functionality, but writes data from the Transmit Buffer either directly to the APB bus or to memory. In general, transfers of five 32-bit words or less are done via the APB, while larger transfers go via memory. This is however something that is specified in the SDK, not implemented in hardware. That means that the system can be implemented to run on platforms without on-board memory, as the MemPort interface can simply be tied off in that case.

The ConveyApbBridge-module serves, as the name implies, as a bridge between the in-

terfaces provided by the Convey PDK and the Communication module. The Dispatch interface drives the APB bus, while the Memory Controller interface is wired against the MemPort.

5.3 Readout

The Readout module extends the CARP platform with a reconfigurable Spiking Neural Network operating in a data-driven fashion, synchronized to the clock of state steps performed in the Cellular Automaton. Each layer of the network is implemented as a stage in a pipeline. With every step of the CA, new input is fed to the input layer and its output is fed as input to the next layer and so on throughout the network. In other words, the number of state steps it takes for data to propagate through the readout module as a whole is equal to the number of layers in the network. The output from the final layer is routed back into the CA. It is also stored in the Readout Buffer, from where it can be read back to the host.

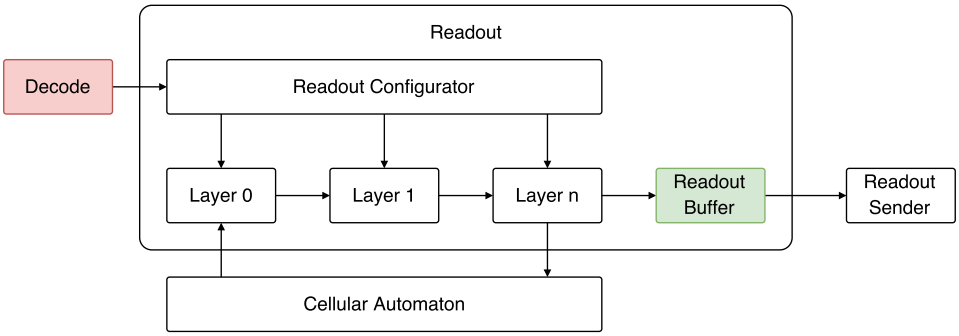


Figure 5.7: Logical overview of the Readout module.

Figure 5.7 shows, at a high level of abstraction, how the Readout module is connected to the CARP system as whole. A subset of cell states are routed out of the CA and into the Readout module as input to the first layer of the network, the input layer. Within each layer, a number of neurons process the input to the layer, as shown in Figure 5.8. Based on this input, they update their activation values, which again are fed to the next layer to be used as input to those neurons in the next step.

The activation value of a neuron can be either 0 or 1, based on a very simple calculation, as shown in Figure 5.9. For each of its incoming edges, a neuron has a pair of registers, the edge weight and a counter. The counter keeps track of how many spikes the neuron has received via the corresponding edge, and the weight is a threshold, indicating how many spikes must arrive via the edge before the neuron can fire. When all counters values are equal to or greater than their weights, the neuron fires and the counters are reset.

To reduce the amount of resources required for the implementation, some restrictions apply with regards to which network topologies are possible to implement. All networks must

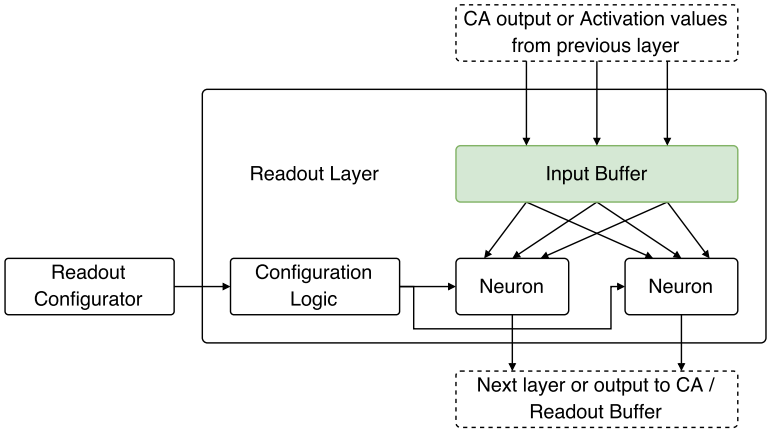


Figure 5.8: Logical overiew of one network layer in the Readout module.

be entirely feed-forward, that is they can not contain any recurrent connections, the total number of neurons can not exceed $2^{16} - 1$ and the final layer must contain only a single neuron. The topology of a network is given as a parameter to the module at synthesis time, and can not be reconfigured while the system is running. It describes how many layers the network consists of and how many neurons each of them consist of. Networks are also implemented fully connected, all neurons in a layer are connected to all neurons in the previous layer.

The module can be in one of two states, the default Processing state and the Configuration state. In the Processing state, data flows through the network in the manner described above. As mentioned, the topology of the network is defined at synthesis time. The weights however, can be reconfigured while the system is running. This allows for rapid exploration of networks with different characteristics. The transition to the Configuration state is triggered when the Readout module receives a WriteWeight instruction from the Decode module. Accompanying the instruction is a payload consisting of a 16-bit address and an 8-bit weight value. Figure 5.10 shows an example of how edges and their corresponding weight registers are addressed, starting from address 0 in the top left, increasing top to bottom between the CA output and the first layer, then continuing from the top in the next.

A WeightWrite instruction completes the same clock cycle it is received, returning the module to the Processing state in the subsequent cycle. This is done by wiring the weight bus directly into each neuron and decoding the address value using entirely combinatorial logic to decide which weight should be updated. At the top level, the address is used to decide into which layer the weight should be written. That layer then receives a high write enable signal and the address of the weight within the layer. Similarily, the targeted layer, uses this address to find the neuron to which the targeted weight belongs and passes the write enable signal into it along with the address of the weight within that neuron. The neuron asserts write enable for the targeted register which will then contain the new weight value the subsequent clock cycle.

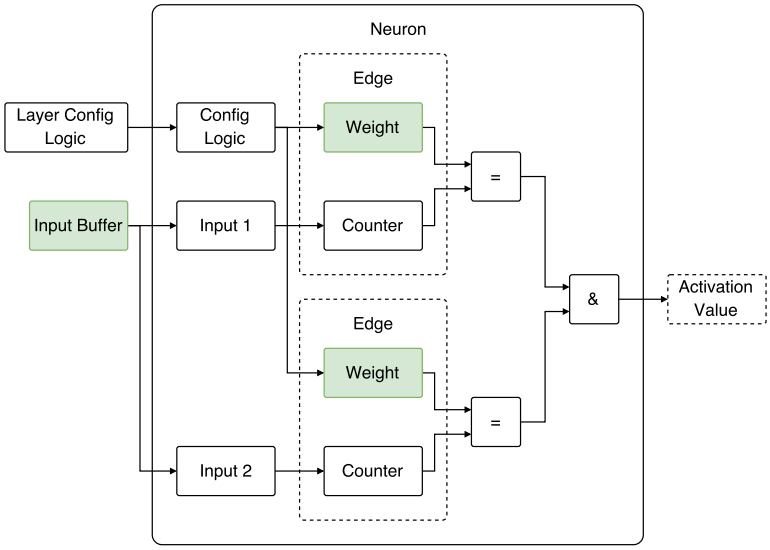


Figure 5.9: Logical overview of a single neuron in the Readout module.

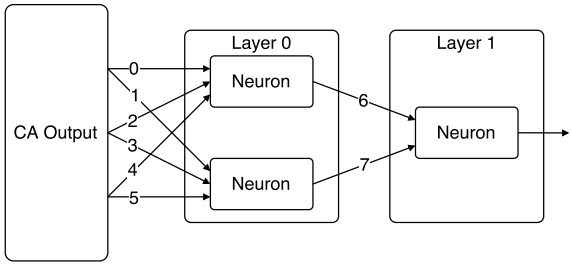


Figure 5.10: Example of how edges/weights are addressed counting from left to right, top to bottom.

With the configuration in Figure 5.10 as an example; a WriteWeight instruction arrives with the address 7 in the payload. At the top level, the module knows that addresses in the range $[6, 7]$ belong to Layer 1. The address of the targeted weight within that layer is found by subtracting the total number edges between all previous layers from the global address, i.e. $7 - 6 = 1$. Inside that layer, the correct neuron is found by utilizing the fully connected nature and calculating $Addr \operatorname{div} NeuronsPreviousLayer$. The address of the weight within that neuron is calculated with $Addr \bmod NeuronsPreviousLayer$. In this example, the only neuron in the layer receives the weight address 1 and updates the correct weight accordingly.

5.4 Cellular Automaton

Alongside the Development and Readout modules, the Cellular Automaton module forms the core of the CARP system. It is responsible for simulating the dynamic behavior of and interaction between cells. The CA module has been partially ported to Scala and modified to support routing live cell states out of the SBlock Matrix and feedback from the Readout Module back in. To achieve this, the top-level logic responsible for orchestrating configuration of the SBlock Matrix and the matrix itself have been reimplemented from scratch in Chisel, while the Live Count related modules remain unchanged and are still implemented in VHDL.

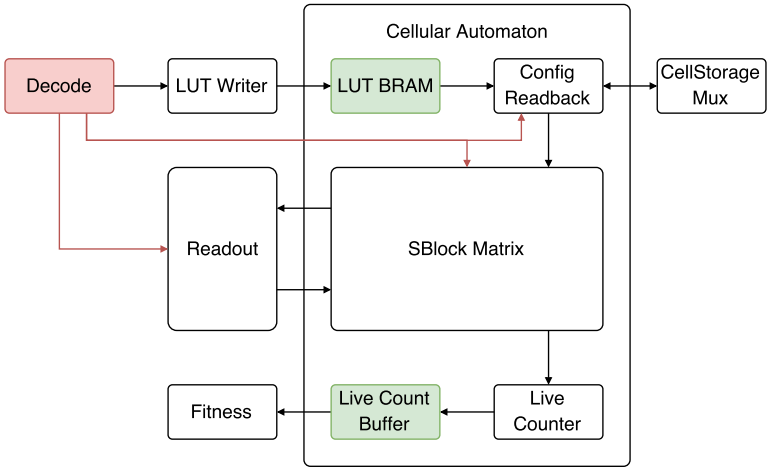
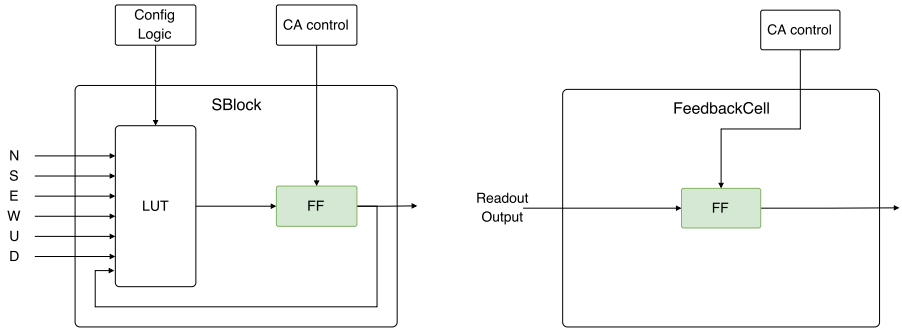


Figure 5.11: The Cellular Automaton module and surrounding modules.

As shown in Figure 5.11, the central part of the CA module is the SBlock Matrix, a 2D/3D matrix of modules connected in a regular grid. The modules can be either an SBlock or a FeedbackCell. An SBlock uses the states of the the cells in its von Neumann neighborhood as input to a configurable Look-Up-Table (LUT) and updates its own state, stored in a Flip-flop (FF), with the output from the LUT. A FeedbackCell on the other hand, receives only the output from the Readout module as input and updates its state with this value. Both are shown in Figure 5.12. State updates happen synchronously throughout the matrix, all cells march in step so to speak. At synthesis time, the SBlock Matrix utilizes two lists of coordinates, one indicating which cells should be FeedbackCells, and one indicating which cells' states should be routed out of the module to be wired as input to the Readout module. All cells not in the FeedbackCells list will be instantiated as SBlocks.

The state machine in Figure 5.13 controls the operation of the CA module. Starting from the Idle state, the module can transition into one of three states based on instructions received from the Decode module. In the Configuration state, the SBlocks in the SBlock Matrix are configured one row at a time. LUTs are configured with different values depending on the cell type of the cell the SBlock simulates. FeedbackCells are not configurable and can not develop into a different cell type, so they are simply ignored with regards to LUT



(a) SBlock **(b)** FeedbackCell

Figure 5.12: Detailed view of the SBlock and FeedbackCell modules used in the SBlock Matrix.

configuration. The state of each cell is also read from Cell Storage and configured into FFs.

In the Readback state, the state of each cell is read back to the Cell Storage. Similar to configuration, this happens row by row.

The Step state is the core functionality of the CA module. Here, all cells/SBlocks synchronously update their states. The instruction also includes a 16-bit number in its payload, indicating how many steps to perform in bulk. When a step has completed, a signal is sent to the Readout module indicating that it should perform one step of its pipeline, and start processing the new cell states it has received.

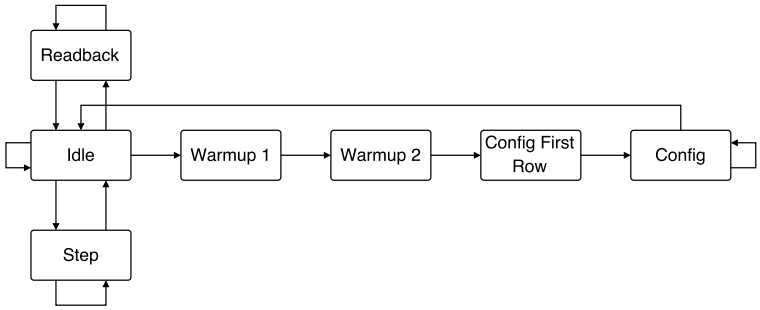


Figure 5.13: State machine controlling the operation of the CA module.

While the Live Count and Fourier Transform modules implemented by Lundal have not been used in the work presented in this thesis, they have been left unchanged and functional for possible future use.

5.5 Miscelaneous Modules

In addition to the major additions and changes to the system outlined in the previous sections, several smaller changes have been made throughout the system to accomodate the new functionality.

5.5.1 Readout Sender

The Readout Sender is a small module added to facilitate transferring Readout output data from the Readout Buffer to the host. It receives instructions from the Decode module and places data in the Transfer Buffer of the Communication module via the Send Buffer Mux. A single ReadReadout instruction causes 32 words to be transferred.

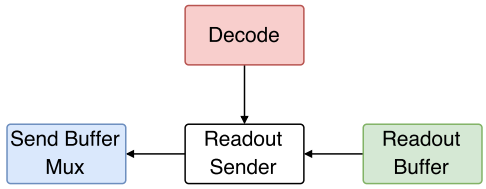


Figure 5.14: The readout sender orchestrates transfers of readout-data to the host.

5.5.2 Decode

The Decode module is responsible for parsing instructions, setting control signals and passing instruction parameters to modules. It has been extended to support the two new instructions. For the ReadReadout instruction, it signals the Readout Sender to initialize a transfer, as well as setting the necessary control signals to the Send Buffer Mux to allow data from the Readout Sender to pass through to the Transfer Buffer. In the case of a WriteWeight instruction, the address and weight value parameters are extracted from the instruction payload passed on to the Readout module along with the control signals indicating that a weight should be updated.

5.5.3 Information Sender

Parameterization has been used extensively throughout the CARP hardware system to allow for as much flexibility as possible. To avoid having to update the software API every time the system is synthesized with new parameters, Lundal introduced the Information Sender, a module that allows the API to query the hardware for information. When a ReadInformation instruction is issued, it places a number of system parameters into Transfer Buffer; CA size, wether or not wrapping is enabled, number of bits per cell state and type, control flow counter sizes, maximum number of development rules and information about

the fitness module. The module has been extended to also include information relating to the Readout module. The added parameters are; number of network layers in the Readout topology, number of output cells from the CA, and the number of neurons in each network layer. Since the system can be synthesized with any number of network layers, the total size of the payload generated by the Information Sender will depend on this number.

5.6 Parameterization

Many aspects of the CARP hardware platform are parameterized. With the introduction of the Readout module, a few new parameters have been added. Readout Buffer size, Readout Weight Bits and Readout Address Bits are integer values controlling bus widths and buffer sizes. The CA Output Cells and CA Feedback Cells parameters are lists of (z, y, x) coordinate triplets indicating the location of output and Feedback cells respectively. The Readout Topology parameter determines the number of network layers and number of neurons in each. For instance, Readout Topology value of $(10, 5, 3, 2, 1)$ will result in a 5 layer network with 10 neurons in layer 0, 5 in layer 1, 3 in layer 2, 2 in layer 3 and 1 neuron in the output layer. A full list of parameters is given in Table 5.1.

To leverage the strengths of Chisel and the Object-Oriented aspects of the underlying Scala, parameters have been organized into an interface (or trait in Scala jargon), `CarpParameters`. This allows for more flexibility in expressing the more complicated parameters such as CA Output Cells and CA Feedback Cells.

5.7 Software API

The CARP software API provides a clean and structured interface into controlling and utilizing the functionality implemented in the hardware part of the system. Figure 5.15 shows how the API is structured. The main part, indicated in green, provides functions for connecting to, resetting, sending instructions to and receiving data from the platform. The two optional modules, Print and PostScript, provide utilities for visualizing the datastructures used in the API.

In this thesis, the communication API module has been reimplemented to utilize the Convey SDK for communication with the coprocessor. Support for new functionality related to the Readout module has also been added.

Parameter	Values	Note
Tx/Rx Buffer Address Bits	[1, ∞]	Determines size of Rx/Tx Buffers
ProgramCounter Bits	[1, 16]	Restricted by ISA/Decode
Matrix Width	[2, 256]	Restricted by ISA/Decode
Matrix Height	[2, 256]	Restricted by ISA/Decode
Matrix Depth	[1, 256]	Restricted by ISA/Decode
Matrix Wrap	True, False	
Cell Type Bits	[1, 32]	Restricted by ISA/Decode
Jump Counter Amount	[1, 256]	Restricted by ISA/Fetch
Jump Counter Bits	[1, 32]	Restricted by ISA/Fetch
LUT Configuration Bits	1, 2, 4, 8	Restricted by SBlocks
Rule Amount	[1, ∞]	
Rules Tested in Parallell	[1, ∞]	
Rule Vector Buffer Size	[1, ∞]	
Fitness Buffer Size	[1, ∞]	
Readout Buffer Size	[1, ∞]	
Readout Weight Bits	[1, 8]	Restricted by ISA/Decode
Readout Address Bits	[1, 16]	Restricted by ISA/Decode
Readout Topology	N/A	List of positive integers
CA Output Cells	N/A	List of (z, y, x) coordinates
CA Feedback Cells	N/A	List of (z, y, x) coordinates

Table 5.1: List of parameters for the CARP hardware

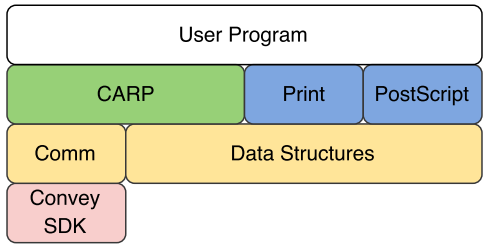


Figure 5.15: Structure of the CARP API. The main API is colored green, optional utilities blue, internal dependencies yellow and external dependencies red. Modified reprint from [22].

5.7.1 Communication

The Convey SDK provides functions for implementing a commuication flow between host and coprocessor as an abstraction above direct communication via PCIe. Figure 5.16 shows the general sequence of actions required to establish connection with the CARP

hardware system. The communication module in the software API has been updated to support this flow as well as the protocol implemented in the hardware Communication module, described in Section 5.2.

5.7.2 Readout

With the addition of the Readout module, the CARP hardware system supports two new instructions, ReadReadout and WriteWeight. Support for both of these have been added to the CARP API. Additionally, a data structure for working with Spiking Neural Networks has been implemented alongside helper/utility functions for batched updates of weights in the Readout module, initialization and printing of networks.

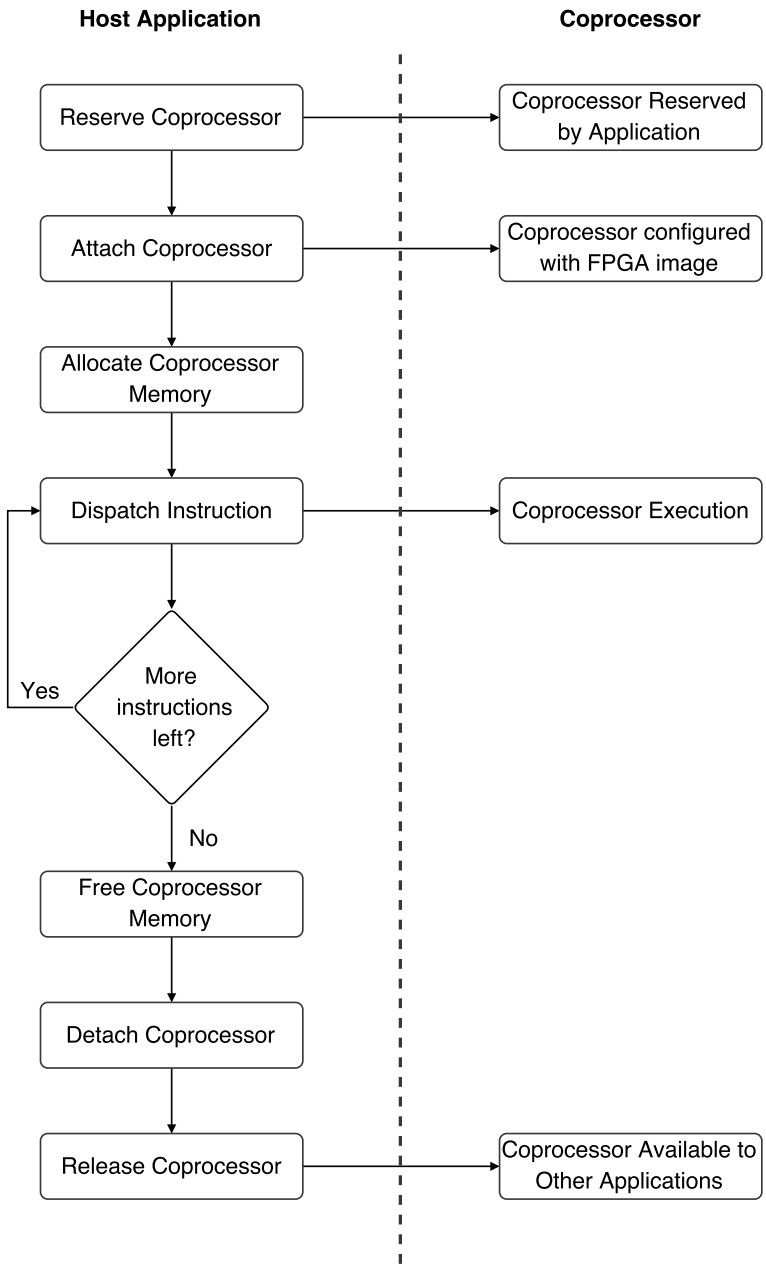


Figure 5.16: Diagram illustrating the flow of communication between host and a Convey coprocessor.

Chapter 6

Verification

The functionality of the extended CARP platform has been verified through extensive testing, both through simulations and using the synthesised design running on the Convey coprocessor.

6.1 Tests

6.1.1 Unit Tests

In addition to generating Verilog, Chisel is also capable of producing cycle-accurate simulations of circuits in C++. The framework also provides tools necessary for writing module-level unit tests that target these simulations. To verify functionality and ensure interoperability with surrounding modules, tests have been written for all modules implemented entirely in Chisel. These include the InformationSender, Readout, SNNLayer, Neuron, ReadoutSender and LUTWriter modules. Modules that blackbox VHDL modules, such as the new CA module, can not be simulated in this way, as the C++ simulator does not support mixed-language simulation yet.

6.1.2 Functional Tests

To verify the integrity of the platform as a whole, Lundal wrote a series of functional tests interacting with the hardware platform via the C API. These have been updated and extended to ensure compatibility with changed functionality. New tests have also been written to verify new functionality, focusing especially on the interaction between the Readout and Cellular Automaton modules.

Tests have been ran for multiple configurations of the system, including for SBlock Matrices of different dimensions and Spiking Neural Networks with different topologies.

6.2 Example

To showcase the functionality added to the system, the example organism in Figure 6.3 has been created. It uses 10 cell types and 19 development rules, and regulates its own growth through feedback from the readout layer.

As shown in Figure 6.1, the platform is configured with an 8x8 matrix of cells. FeedbackCells are placed at coordinates (1, 1), (1, 5), (5, 1) and (5, 5), with the cells surrounding (1, 1) as output cells. The developmental rules are created to have the organism grow in spirals around the FeedbackCells. All cell types are associated with a LUT that sets their state to 1 regardless of the state of neighbors. Between each development step, two state steps are performed. One to let the Readout module process the recently updated states, and one to update the state of the FeedbackCells update their state based on the output from the Readout module. As shown in Figure 6.2, the Readout module is configured with a single neuron that will fire once it has received spikes along all input edges. When all cells around the FeedbackCells are grown into, the Readout module fires a spike. The development rules interpret this spike as a sign of overpopulation and kills the cells directly above, below, left and right of the FeedbackCells. The cells adjacent to these then die of starvation the next development step.

This example shows how the Readout module can be used to classify the behavioral dynamics of a cellular organism, and how that classification can be used to regulate further growth and development.

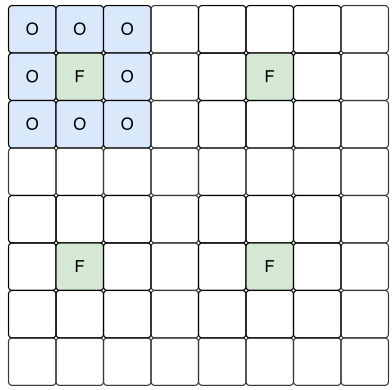


Figure 6.1: CA configuration used in creating a self-regulating organism. Green cells are FeedbackCells, to which Readout output is routed, and blue cells are output cells, regular SBlocks whose output state is routed as input to the Readout module.

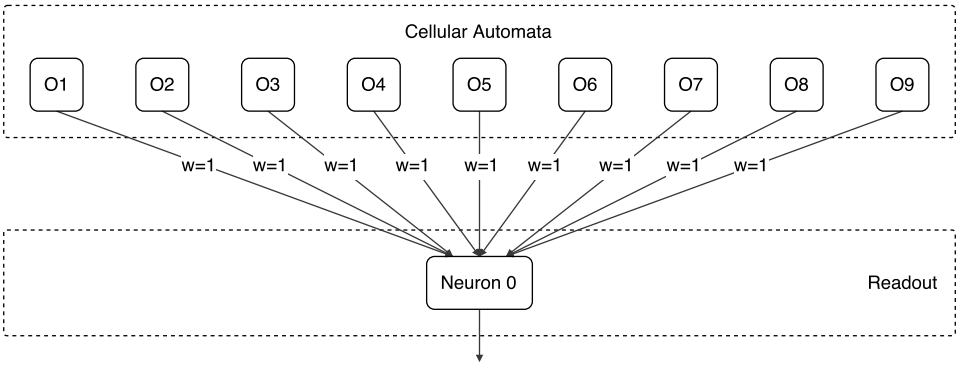
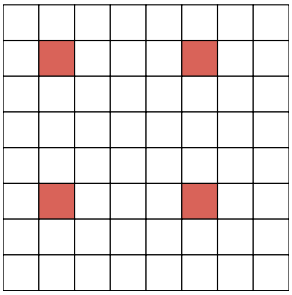
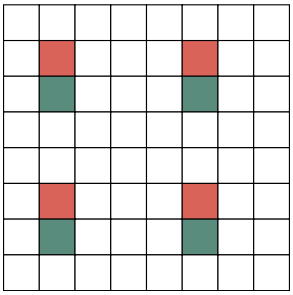


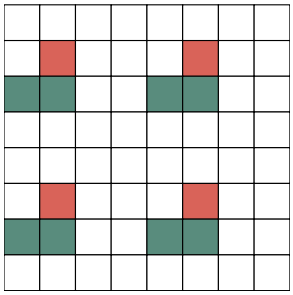
Figure 6.2: Configuration of the Readout module used in the example self-regulating organism



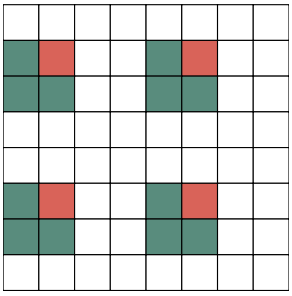
(a) Step 0



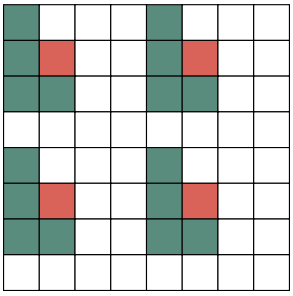
(b) Step 1



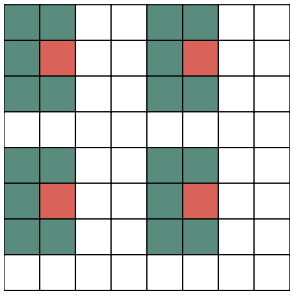
(c) Step 2



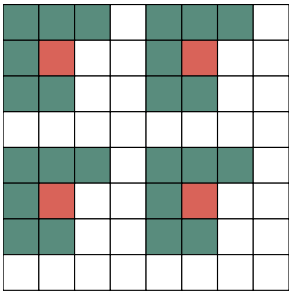
(d) Step 3



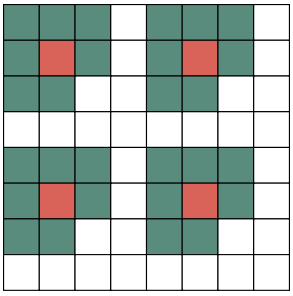
(e) Step 4



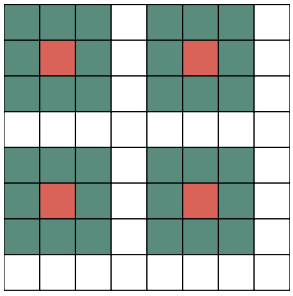
(f) Step 5



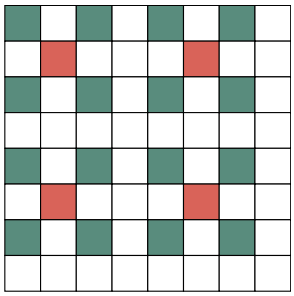
(g) Step 6



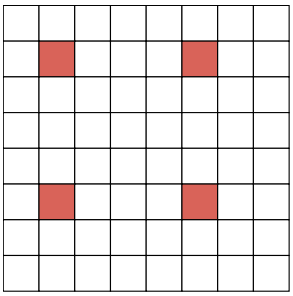
(h) Step 7



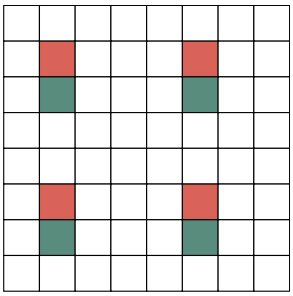
(i) Step 8



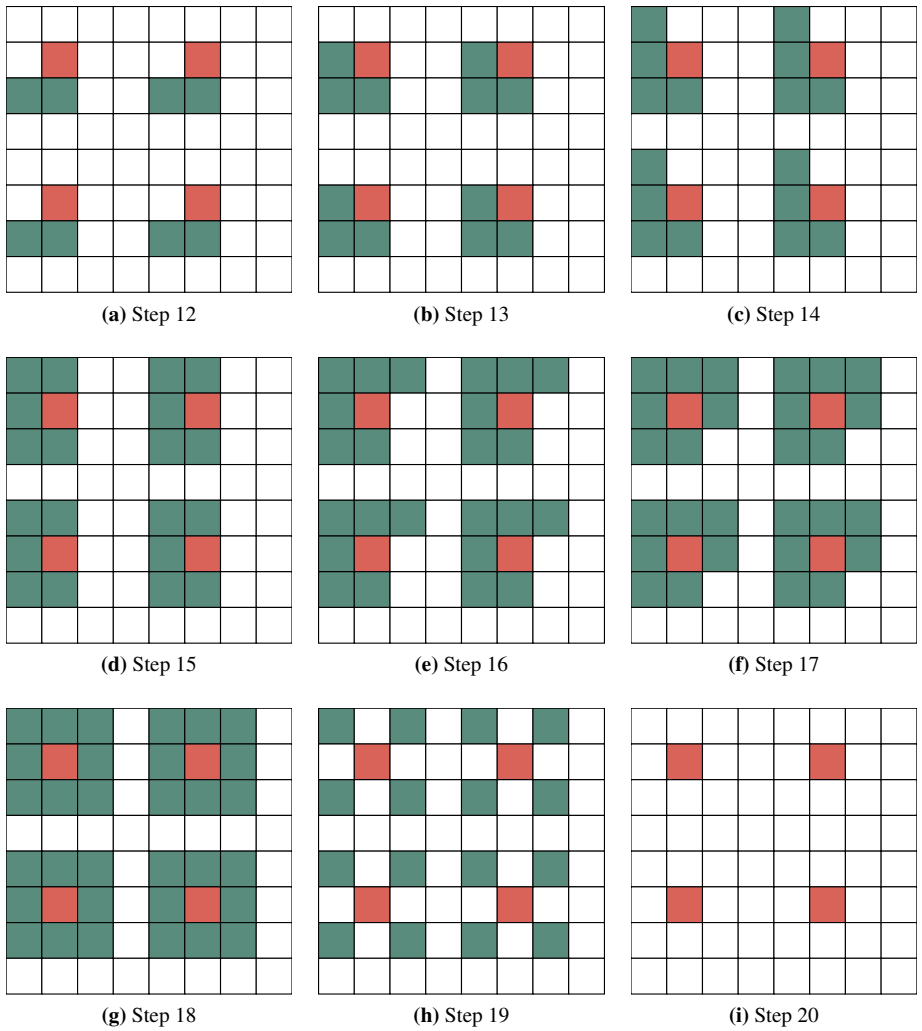
(j) Step 9



(k) Step 10



(l) Step 11



(a) Step 12 **(b)** Step 13 **(c)** Step 14

(d) Step 15 **(e)** Step 16 **(f)** Step 17

(g) Step 18 **(h)** Step 19 **(i)** Step 20

Figure 6.3: Development steps of a self-regulating cellular organism simulated on the CARP platform. At steps 9 and 19, the Readout module classifies the organism as overpopulated. The next development step, the organism reacts to this by killing off cells. The growth cycle subsequently restarts.

Chapter 7

Discussion

The proof of concept experiment in Section 6.2 gives a small glimpse into the possibilities opened up by adding the Readout module to the CARP system. While the example only showcases how processing the dynamics of the CA allows for adaption in the developmental process, more complex configurations would, hopefully, allow for cellular organisms that not only are capable of advanced computations, but that are able to adapt and learn from environmental feedback in ways not previously possible.

7.1 Resource Usage

As shown in Table 7.1, the Virtex-7 FPGA on the Convey coprocessor gives the CARP system quite a bit of room to grow. A design with an SBlock Matrix with dimensions $96 \times 96 = 9216$ cells uses $\sim 20\%$ of LUTs and $\sim 24\%$ of BRAMs available. In terms of flat quantities, the number of LUTs used has increased significantly in the new design. By comparing designs with similar SBlock Matrices, it becomes apparent that this is due to the added overhead of the Convey PDK Application Engine design into which the CARP design is placed.

7.2 Challenges

During testing, it was discovered that the Xilinx FPGA flow crashes during synthesis of large SBlock Matrices. A 96×96 large design completes without problems, a 128×128 design however stalls while synthesising SBlocks and receives a segmentation fault after a few hours. Due to time constraints and the fact that this issue did not hinder the implementation of new functionality, it has not been investigated thoroughly. To fully utilize the resources available on the Convey coprocessor it will have to be resolved however.

Cell Count	LUTs Total	LUTs %	Registers Total	Registers %	BRAMs Total	BRAMs %
8×8	194624	15.93	208690	8.54	227.5	18.91
64×64	224442	18.37	221061	9.05	280.0	23.28
72×72	230094	18.84	223355	9.14	286.5	23.82
96×96	251219	20.56	231616	9.48	308.5	25.64
$4 \times 4 \times 4$	195360	15.99	209171	8.56	232.5	19.33
$16 \times 16 \times 16$	249320	20.41	218109	8.93	261	21.70

Table 7.1: FPGA Resource usage of the CARP system

The modules that remain implemented in VHDL use custom types extensively. This makes it difficult to simulate the CARP hardware system as a whole, as no available simulation tools support mixed-language Verilog/VHDL simulation of designs using non-standard types. Being unable to simulate the system end-to-end means that a full synthesis of the system is required in order to test even the smallest of changes. A full synthesis takes upwards of two hours, even for small matrix dimensions, resulting in a very long feedback cycle. The optimal way to resolve this, would be to port the remaining modules to Chisel, and implement end-to-end test utilizing the C++ simulator generated by the framework. A second strategy would be to implement custom simulation modules for all modules blackboxed by Chisel. This would also allow for end-to-end tests in the Chisel simulator, but in terms of workload required be comparable to actually porting the system completely. A third solution would be to remove all custom types from the VHDL parts of the design. This would certainly be quicker than a full port, but having a fragmented codebase is definitely less desirable in the long term.

7.3 Future Work

To further extend the platform along the epigenetic axis, it is necessary to allow for other inputs and perturbartions to the CA than the feedback from the Readout module. A possible solution would be to implement a generic type of InputCell that, similar to the FeedbackCell used in this thesis, has state determined entirely by an outside source. That outside source could potentially be a bitstream stored on the on-board memory on the Convey coprocessor.

In the example presented in Section 6.2, configuring the SNN in the Readout module was trivial to do by hand in order to illustrate the potential capabilities. In order to effectively utilize the Readout module in more advanced experiments, an efficient training algorithm will have to be implemented. In general, training spiking neural networks is a hard task. In the specialization project report in Appendix C, an evolutionary algorithm is used to guide a search through the space of possible network configurations. This technique is both

slow and it does not converge reliably. A more efficient solution would be to implement a version of the SpikeProp algorithm [4] adapted to the simplified spiking neural networks modeled in the Readout module. Another possible approach is fully on-line training based on spike-timing-dependent plasticity [25]. Here, weight tuning happens locally inside each neuron, based on the timing of incoming spikes relative to when the neuron fires. This approach is used in IBMs Truenorth computer [31], discussed in Section 2.5.1.

Chapter 8

Conclusion

In this thesis, a physical realisation of a developmental, cellular automata-based dynamical system with dynamical structure $(DS)^2$ has been explored in a reservoir computing (RC) context. A system implemented with information encoded temporally as spiketrains from end to end has been implemented and tested. This is based on inspiration from biology and nature, wherein systems such as the human nervous system operates entirely within the spiking domain.

To achieve this, the Cellular Automata Research Platform (CARP) has been augmented with new functionality. A flexible, reconfigurable Readout module has been added to the system, designed to perform real-time classification of temporal behavior in a dynamical system with dynamical structure $(DS)^2$. The Readout module serves as an efficient way to allow the behavior of cellular computing systems to be interpreted as output. The output from the Readout module is also fed back into the dynamical system being simulated, as a form of environmental feedback that both the behavior and structure of the system can utilize to self-regulate.

With this new functionality, the CARP systems is opened up to being used for research that encompasses all three axis of the POE-ontology. Systems develop and adapt over time (Ontogeny) based on evolved developmental rules (Phylogenesis) and feedback from the environment (Epigenesis).

The platform has also been ported to run on new state-of-the-art hardware. It has also been partially ported to a more modern implementation language, Chisel. The new platform has been extensively tested through both unit tests of simulated modules and functional tests of the entire system end-to-end. The new hardware should allow the current implementation to scale up to $\sim 192 \times 192$ CA cells in 2D configurations and $\sim 24 \times 24 \times 24$ in 3D.

All in all, this thesis provides a flexible and powerful framework for research into artificial development and evolution, artificial life and bio-inspired $(DS)^2$ RC systems.

Bibliography

[1] Kjetil Aamodt. Kunstig utvikling: Utvidelse av FPGA-basert SBlock-plattform. Technical report, 2005.

[2] T Bäck, D B Fogel, and Z Michalewicz. Handbook of Evolutionary Computation. *Evolutionary Computation*, 2:1–11, 1997.

[3] Eva Bianconi, Allison Piovesan, Federica Facchin, and Et al. An estimation of the number of cells in the human body. *Annals of Human Biology*, 2013.

[4] Sander M. Bohte, Joost N. Kok, and Han La Poutré. Error-backpropagation in temporally encoded networks of spiking neurons, 2002.

[5] Robert Connelly, Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. Winning Ways for Your Mathematical Plays. *The American Mathematical Monthly*, 93(5):411, 1982.

[6] Charles Darwin. *On the Origin of Species*. 1859.

[7] Asbjørn Djupdal. Konstruksjon av maskinvare for kjøring av sblokkbaserte eksperimenter. Technical report, 2004.

[8] Chrisantha Fernando and Sampsa Sojakka. Pattern Recognition in a Bucket. *Advances in Artificial Life*, pages 588–597, 2003.

[9] Jeannie M Fitzgerald, Conor Ryan, David Medernach, and Krzysztof Krawiec. An Integrated Approach to Stage 1 Breast Cancer Detection. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1199–1206, New York, NY, USA, 2015. ACM.

[10] Dario Floreano, Phil Husbands, and Stefano Nolfi. Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines. *Evolutionary Robotics*, pages 1423–1451, 2000.

[11] F. A. Gers and J. Schmidhuber. LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

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

[13] Pauline C. Haddow and Gunnar Tufte. An evolvable hardware FPGA for adaptive hardware. In *Proceedings of the 2000 Congress on Evolutionary Computation, CEC 2000*, volume 1, pages 553–560, 2000.

[14] Brian K. Hall, Roy D. Pearson, and Gerd B. Müller. *Environment, Development, and Evolution: Toward a Synthesis*. 2003.

[15] Barbara Hammer and Jochen J. Steil. Tutorial: Perspectives on learning with rnns. *Proc. ESANN*, (April):357–368, 2002.

[16] Herbert Jaeger. The "echo state" approach to analysing and training recurrent neural networks. *GMD Report*, 148:1–47, 2001.

[17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-Luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon Mackean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, pages 1–17, 2017.

[18] Sergei L. Kosakovsky Pond, David Posada, Michael B. Gravenor, Christopher H. Woelk, and Simon D W Frost. GARD: A genetic algorithm for recombination detection. *Bioinformatics*, 22(24):3096–3098, 2006.

[19] John R Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[20] Chris G. Langton. Computation at the edge of chaos: Phase transitions and emergent computation. *Physica D: Nonlinear Phenomena*, 42(1-3):12–37, 1990.

[21] Steve Lawrence, CL Lee Giles, S Fong, Steve Lawrence, and Ah Chung AC Tsoi. Noisy Time Series Prediction using Recurrent Neural Networks and Grammatical Inference. *Machine Learning*, 44(1):161–183, 2001.

[22] Per Thomas Lundal. The Cellular Automata Research Platform: Revised, Rebuilt and Enhanced. Technical report, 2015.

[23] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997.

[24] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.

[25] H. Markram, W. Gerstner, and P. J. Sjöström. Spike-timing-dependent plasticity: A comprehensive overview. *Frontiers in Synaptic Neuroscience*, 4(JULY):2010–2012, 2012.

[26] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[27] Melanie Mitchell, Peter Hraber, and James P. Crutchfield. Revisiting the Edge of Chaos: Evolving Cellular Automata to Perform Computations. *arXiv preprint adap-org/9303003*, 7:38, 1993.

[28] S Nichele and G Tufte. Evolution of Incremental Complex Behavior on Cellular Machines. *Ecal 2013*, pages 63–70, 2013.

[29] S Nichele and G Tufte. Evolutionary growth of genomes for the development and replication of multicellular organisms with indirect encoding. In *2014 IEEE International Conference on Evolvable Systems*, pages 141–148, dec 2014.

[30] Robert Preissl, Theodore M. Wong, Pallab Datta, Myron Flickner, Raghavendra Singh, Steven K. Esser, William P. Risk, Horst D. Simon, and Dharmendra S. Modha. Compass: A scalable simulator for an architecture for cognitive computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[31] J s. Seo, B Brezzo, Y Liu, B D Parker, S K Esser, R K Montoye, B Rajendran, J A Tierno, L Chang, D S Modha, and D J Friedman. A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In *2011 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4, sep 2011.

[32] Kirk Saban. Interconnect Technology Delivers Breakthrough FPGA Capacity , Bandwidth , and Power Efficiency. Technical report, 2012.

[33] Eduardo Sanchez, Daniel Mange, Moshe Sipper, Marco Tomassini, Andres Perez-Uribe, and Andre Stauffer. Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1259, pages 33–54, 1997.

[34] Moshe Sipper. Emergence of cellular computing. *Computer*, 32(7):18–26, 1999.

[35] Moshe Sipper. Evolution of Parallel Cellular Machines: The Cellular Programming Approach. 2004, 2004.

[36] David Snyder, Alireza Goudarzi, and Christof Teuscher. Computational capabilities of random automata networks for reservoir computing. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 87(4), 2013.

[37] Antoine Spicher, Olivier Michel, and Jean-louis Giavitto. A Topological Framework for the Specification and the Simulation of Discrete Dynamical Systems, 2004.

[38] Jochen J. Steil. Backpropagation-Decorrelation: Online recurrent learning with O(N) complexity. In *IEEE International Conference on Neural Networks - Conference Proceedings*, volume 2, pages 843–848, 2004.

[39] Ola Martin Tiseth Støvneng. 2D and 3D On-Chip Development of Cellular Automata Machines. Technical Report May, 2014.

[40] Gunnar Tufte. Evolution, Development and Environment Toward Adaptation through Phenotypic Plasticity and Exploitation of External Information. In *Artificial life XI The Eleventh International Conference on the Simulation and Synthesis of Living Systems*, volume 11, pages 624–631, 2008.

[41] Gunnar Tufte and Pauline C. Haddow. Towards development on a silicon-based cellular computing machine. *Natural Computing*, 4(4):387–416, 2005.

[42] Gunnar Tufte and Odd Rune Lykkebø. Evolution-in-Materio of a dynamical system with dynamical structures. *Proceedings of the Artificial Life Conference 2016*, 2016.

[43] D. Verstraeten, B. Schrauwen, M. D'Haene, and D. Stroobandt. An experimental unification of reservoir computing methods. *Neural Networks*, 20(3):391–403, apr 2007.

[44] John Von Neumann and Michael D. Godfrey. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[45] Zhizheng Wu and Simon King. Investigating gated recurrent networks for speech synthesis. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, volume 2016-May, pages 5140–5144, 2016.

[46] Ozgur Yilmaz. Reservoir Computing using Cellular Automata. *arXiv preprint*, pages 1–9, 2014.

Appendix A

Interface Specifications

A.1 Advanced Peripheral Bus (APB)

APB is a bus specification by ARM designed for low bandwidth communication between register interfaces. Table A.1 lists its constituent buses and signals.

Signal	Source	Description
PCLK	Clock Source	Clock. The rising edge of PCLK times all transfers on the APB.
PRESETn	System bus equivalent	Reset. The APB reset signal is active LOW. This signal is normally connected directly to the system bus reset signal.
PADDR	APB bridge	Address. This is the APB address bus. It can be up to 32 bits wide and is driven by the peripheral bus bridge unit.
PPROT	APB bridge	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access.
PSELx	APB bridge	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a PSELx signal for each slave.
PENABLE	APB bridge	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
PWRITE	APB bridge	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
PWDATA	APB bridge	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when PWRITE is HIGH. This bus can be up to 32 bits wide.
PSTRB	APB bridge	Write strobes. This signal indicates which byte lanes to update during a write transfer. There is one write strobe for each eight bits of the write data bus. Therefore, PSTRB[n] corresponds to PWDATA[(8n + 7):(8n)]. Write strobes must not be active during a read transfer.
PREADY	Slave interface	Ready. The slave uses this signal to extend an APB transfer.
PRDATA	Slave interface	Read Data. The selected slave drives this bus during read cycles when PWRITE is LOW. This bus can be up to 32-bits wide.
PSLVERR	Slave interface	This signal indicates a transfer failure. APB peripherals are not required to support the PSLVERR pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this pin then the appropriate input to the APB bridge is tied LOW.

Table A.1: Signals and buses in the Advanced Peripheral Bus protocol.

A.2 MemPort Bus

The MemPort Bus is a generic memory interface with a request/response style protocol. It consists of two MemBuses wrapped in ready-valid interfaces, one for requests and one for responses. A MemBus has data, address and write enable fields. A full list of signals is given in table A.2.

Signal	Source	Description
req_addr	Requester	Request address.
req_data	Requester	Request data. Only used for write requests.
req_write	Requester	Request type. High for write, low for read.
req_ready	Responder	Responder ready. Asserted by responder when it is ready to receive a new request.
req_valid	Requester	Request valid. Asserted by requester when a new request is dispatched.
resp_addr	Responder	Response Address.
resp_data	Responder	Response data.
resp_write	Responder	Response type.
resp_ready	Requester	Requester ready. Asserted by requester when it is ready to receive a new response.
resp_valid	Responder	Responder valid. Asserted by responder when a new response is dispatched.
flush	Requester	Flush memory.
flush_complete	Responder	Flush complete.

Table A.2: Signals and buses in the MemPort protocol.

Appendix B

Instruction Set Architecture

Cellular Automata Research Platform Instruction Set Architecture

Revision 1.1

2017-05-28

Contents

1 Introduction

This document is a complete specification of the instruction set for the Cellular Automata Research Platform. It documents all effects and possible side effects of every instruction.

Unless otherwise stated, an instruction completes in one cycle. However, keep in mind that multi-word instructions require multiple cycles to send over PCI Express.

When a bit vector is broken into multiple words, the least significant part is always listed first.

Instructions

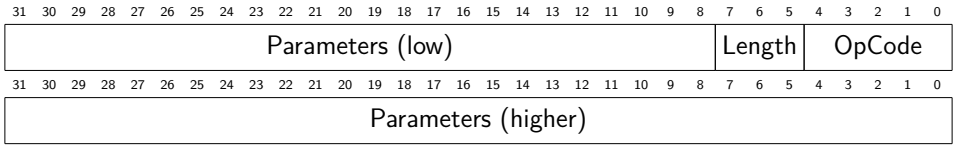
Each instruction is 256 bits and consists of a 5-bit operation code, a 3-bit length field and up to 248 bits of parameters.

The operation code specifies what kind of instruction it is, and how the parameters should be parsed.

The length field is used to improve communication speed by only transmitting the necessary parts of an instruction; It is zero-extended back to 256 bits by the fetch module. The field directly specifies the number of words after the first that are sent.

The parameters are of different types and lengths for each instruction. Please see the individual instruction pages.

Instruction Format



...

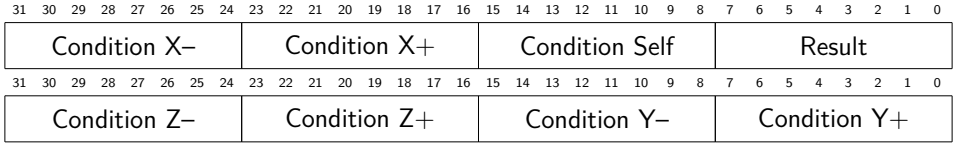
Rules

Rules consists of conditions for each cell in the neighborhood and a result that will be applied to the cell if the conditions match.

Each condition contains a type, a state and a bit for each that marks if it should be checked. The result format is identical except for that the check bits are exchanged with change bits that mark which parts of the cell should change if all conditions match.

In the formats below, [type bits] is assumed to be 5 and [states bits] 1 for the purpose of having everything nicely align to bytes.

Rule Format



Condition Format

Result Format

Notes

For a rule to be counted as a hit, all conditions must match and at least one change bit must be set.

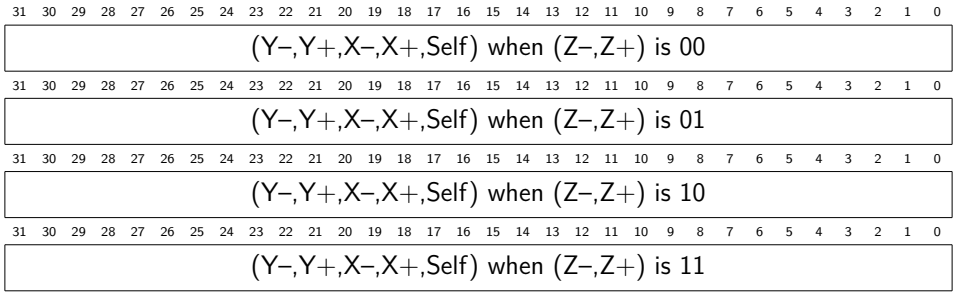
Conditions for Z are ignored when [matrix depth] is 1.

LUTs

The indexing for the look-up tables is (Z–,Z+,Y–,Y+,X–,X+,Self). For each of these indexes, the next cell state is specified. The least significant index is written first (to the right).

In the format below, [state bits] is assumed to be 1 since it is the only value currently supported. This allows the entries for (Y–,Y+,X–,X+,Self) to fit exactly within one word.

LUT Format



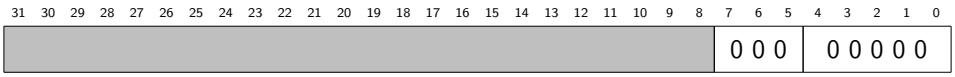
Notes

The Z parts are ignored when [matrix depth] is 1.

2 General Instructions

This section covers instructions that are not used directly or do not fit into any of the other categories.

No Operation



Format

nop()

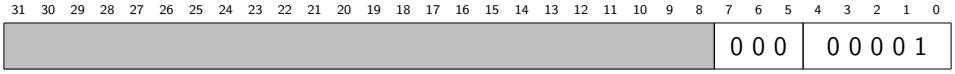
Purpose

To do nothing for one cycle.

Description

Nothing is done for one cycle.

Read Information



Format

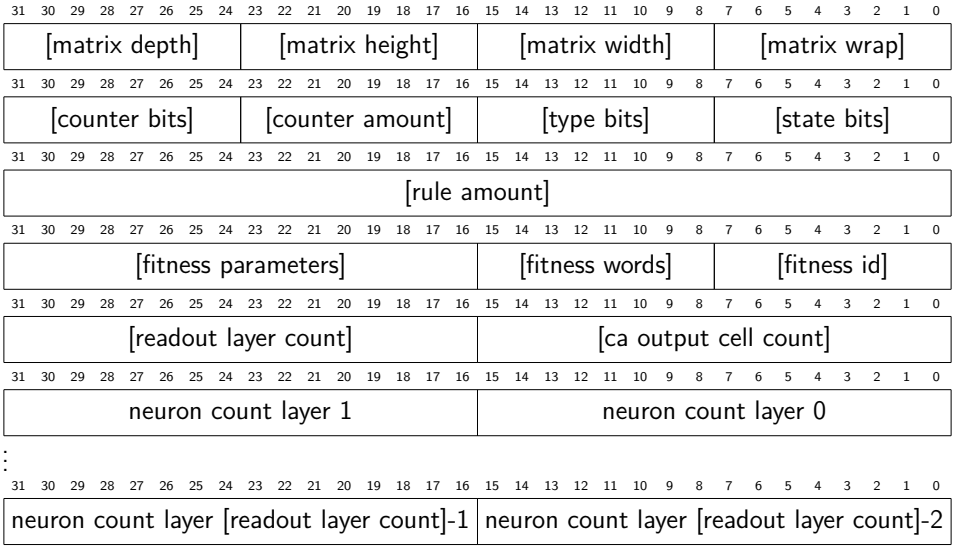
read_information()

Purpose

To retrieve information about the system.

Description

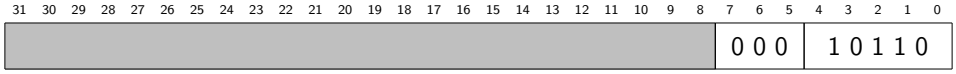
The following words are put into the Send Buffer.



Notes

This instruction takes $5 + \lceil \frac{\lceil readoutlayercount \rceil}{2} \rceil$ cycles.

Read Fitness



Format

read_fitness()

Purpose

To retrieve a fitness value.

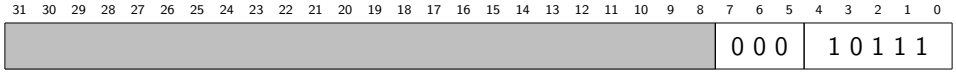
Description

[fitness words] words are transfered from the Fitness Buffer to the Send Buffer.

Notes

This instruction takes [fitness words] cycles.

Read Readout



Format

read_readout()

Purpose

To retrieve readout values.

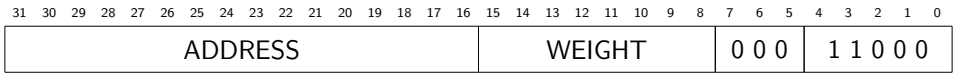
Description

32 words are transfered from the Readout Buffer to the Send Buffer.

Notes

This instruction takes 32 cycles.

Write Weight



Format

write_weight(ADDR, WEIGHT)

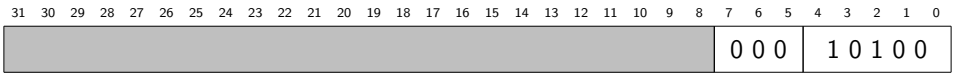
Purpose

To configure the value of a weight in the spiking neural network in the readout module.

Description

Weight at address ADDR is configured with the WEIGHT value.

Swap Cell Storage



Format

swap_cell_storage()

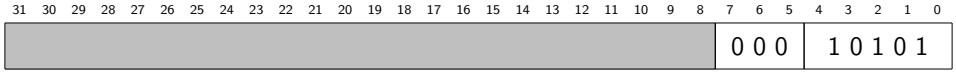
Purpose

To swap the contents of the two brams within the cell storage.

Description

Cell BRAM A and Cell BRAM B are remapped so that the contents appear to have been swapped.

Reset Buffers



Format

reset_buffers()

Purpose

To clear the Rule Vector, Live Count and Fitness Buffers.

Description

The read and write pointers of the circular FIFO buffers are set to 0. This makes them appear to be empty.

Notes

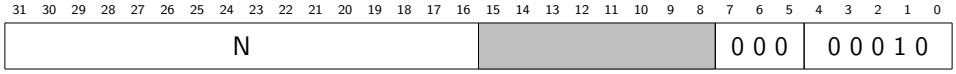
If the Fitness module is processing data, the contents of the Live Count and Fitness Buffers may become undefined.

If the Fitness buffer is full, this incruction should be called an additional time after any pending data from Fitness has been transfered.

3 Development Instructions

This section covers all instructions affecting the development module. This includes writing rules, setting active rules, running development and reading data for which rules have triggered.

Read Rule Vectors



Format

read_rule_vectors(N)

Purpose

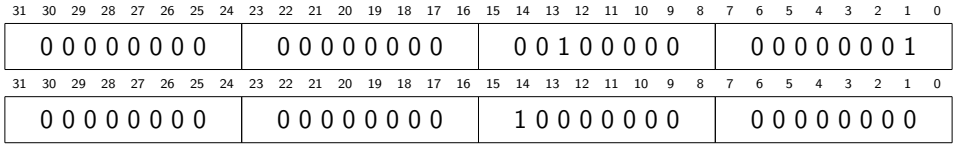
To retrieve N rule vectors.

Description

N rule vectors are placed into the Send Buffer. Each consists of [rule amount] bits, where the first bit (rule zero) is always 1. The Send Buffer is word-aligned after each rule vector by padding with 0.

Example

Assume a system with [rule amount] set to 48, where rules 13 and 47 have triggered. read_rule_vectors(1) will put the following words into the Send Buffer.

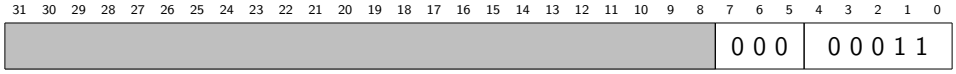


Notes

This instruction takes [words per rule vector] * N cycles.

When there are no rule vectors available and less than N have been sent, this instruction waits.

Read Rule Numbers



Format

read_rule_numbers()

Purpose

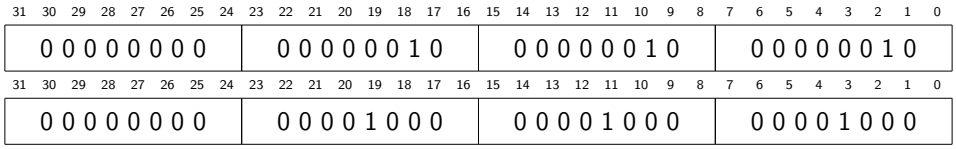
To retrieve the last rule that triggered for each cell during the previous development step.

Description

Rule numbers for the entire matrix is put into the Send Buffer. Each consists of log_2[rule amount] bits, sent in raster order (first X, then Y, then Z). A value of 0 means that no rules triggered. The Send Buffer is word-aligned after each row by padding with 0. If a rule number would be split across two words, it is instead aligned to the next word.

Example

Assume a system with [matrix depth] set to 1, [matrix height] set to 2, [matrix width] set to 3 and [rule amount] set to 256. If rule 2 triggered for all cells in the fist row and rule 8 for all in the second, read_rule_numbers() will put the following words into the Send Buffer.

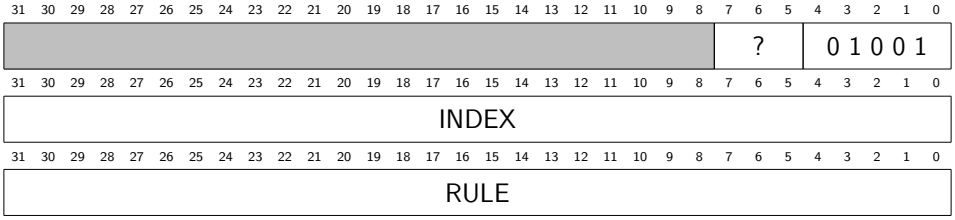


Notes

The execution time (T) for this instruction depends on [matrix depth] (M_Z), [matrix height] (M_Y), [matrix width] (M_X) and [rule amount] (R_A).

$$T = M_Z M_Y \left\lceil \frac{M_X}{\max\left(\left\lfloor \frac{32}{\lceil \log_2 R_A \rceil} \right\rfloor, M_X\right)} \right\rceil + 1$$

Write Rule



Format

write_rule(RULE, INDEX)

Purpose

To write a development rule.

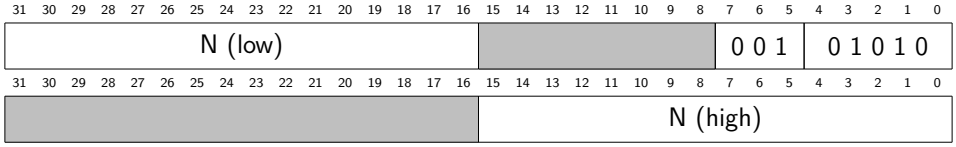
Description

RULE is written to Rule BRAM at address INDEX. The length of RULE varies depending on [matrix depth], [type bits] and [state bits]. It is sent as one continuous piece spanning multiple words. The instruction length field is adjusted accordingly.

Notes

INDEX is cropped to the number of bits in [rule amount].

Set Active Rules



Format

set_rules_active(N)

Purpose

To set the number of rules that are currently active, so others can be skipped to reduce development time.

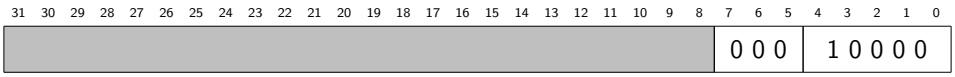
Description

Rules 1 to N is set to active (rule 0 is reserved). If N is 0, no rules will be set to active.

Notes

N is cropped to the number of bits in [rule amount]. If this is 16 or less, the second word can be discarded (and instruction length field set to 0).

Develop



Format

develop()

Purpose

To perform development on all cells.

Description

The cells in Cell BRAM A are fetched and tested against all active rules. If a rule matches a cell, the state and/or type of the cell is changed based on the rule. Rules of higher index override those of lower index. The developed cells are then stored in Cell BRAM B.

The lastly matched rule of each cell is stored in Rule Number BRAM, and a list of all rules with a match is stored to the Rule Vector Buffer.

Notes

An overridden rule will be listed as having a match, but all its effects are discarded.

The execution time (T) for this instruction depends on [matrix depth] (M_Z), [matrix height] (M_Y), [rules active] (R_A) and [rules tested in parallel] (R_{TIP}).

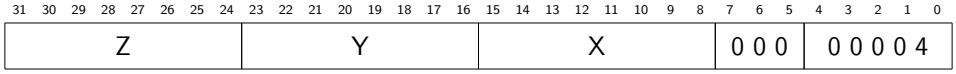
$$T_{3D} = M_Z M_Y \max\left(\frac{R_A + 1}{R_{TIP}}, 7\right) + 6$$

$$T_{2D} = M_Y \max\left(\frac{R_A + 1}{R_{TIP}}, 5\right) + 4$$

4 Cell Storage Instructions

This section covers all instructions for writing and reading states and types to/from the cell storage.

Read One State



Format

read_state(Z, Y, X)

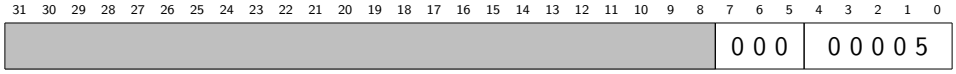
Purpose

To retrieve the state of the cell at (Z, Y, X).

Description

The state of cell (Z, Y, X) is put into the Send Buffer. The Send Buffer is then word-aligned by padding with 0. Accessing cells outside the matrix dimensions yields undefined states.

Read All States



Format

read_states()

Purpose

To retrieve the state of all cells.

Description

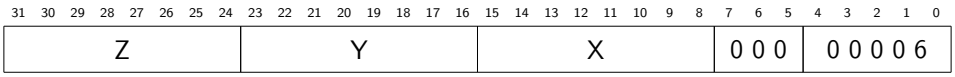
The states of all cells are put into the Send Buffer in raster order (first X, then Y, then Z). The Send Buffer is word-aligned after each row by padding with 0. If a state would be split across two words, it is instead aligned to the next word.

Notes

The execution time (T) for this instruction depends on [matrix depth] (M_Z), [matrix height] (M_Y), [matrix width] (M_X) and [state bits] (B_S).

$$T = M_Z M_Y \left\lceil \frac{M_X}{\max\left(\left\lfloor \frac{32}{B_S} \right\rfloor, M_X\right)} \right\rceil + 1$$

Read One Type



Format

read_type(Z, Y, X)

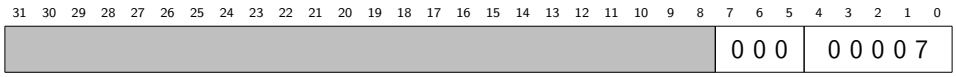
Purpose

To retrieve the type of the cell at (Z, Y, X).

Description

The type of cell (Z, Y, X) is put into the Send Buffer. The Send Buffer is then word-aligned by padding with 0. Accessing cells outside the matrix dimensions yields undefined types.

Read All Types



Format

read_types()

Purpose

To retrieve the types of all cells.

Description

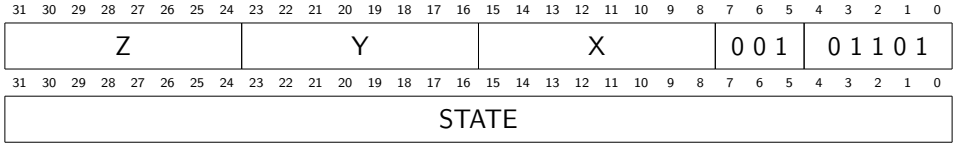
The types of all cells are put into the Send Buffer in raster order (first X, then Y, then Z). The Send Buffer is word-aligned after each row by padding with 0. If a type would be split across two words, it is instead aligned to the next word.

Notes

The execution time (T) for this instruction depends on [matrix depth] (M_Z), [matrix height] (M_Y), [matrix width] (M_X) and [type bits] (B_T).

$$T = M_Z M_Y \left\lceil \frac{M_X}{\max\left(\left\lfloor \frac{32}{B_T} \right\rfloor, M_X\right)} \right\rceil + 1$$

Write One State



Format

write_state(Z, Y, X, STATE)

Purpose

To write one state.

Description

State (Z, Y, X) in Cell BRAM A is set to STATE.

Notes

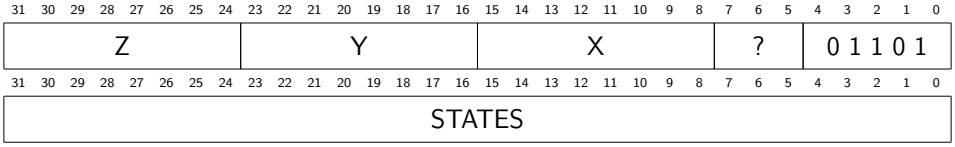
Each coordinate is cropped to the bits in its matrix dimension.

STATE is cropped to [state bits].

If row (Z, Y) is outside the defined matrix, data will still be written but only accessible by read instructions.

If X is outside the defined matrix, nothing will happen.

Write Row of States



Format

write_states(Z, Y, X, STATES)

Purpose

To write one row (or as many can fit an instruction) of states.

Description

STATES is a list of states in little-endian order. It is either [matrix width] or as many can fit 224 bits in length. Each state is [state bits] long.

The states are written to Cell BRAM A at row (Z, Y). They are offset so the first state is written to position X within the row. States offset to [matrix width] or more are discarded.

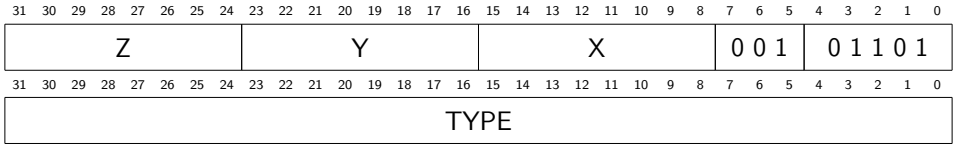
The length of STATES varies depending on [matrix width] and [state bits]. It is sent as one continuous piece spanning multiple words. The instruction length field is adjusted accordingly.

Notes

Each coordinate is cropped to the bits in its matrix dimension.

If row (Z, Y) is outside the defined matrix, data will still be written but only accessible by read instructions.

Write One Type



Format

write_types(Z, Y, X, TYPE)

Purpose

To write one state.

Description

Type (Z, Y, X) in Cell BRAM A is set to TYPE.

Notes

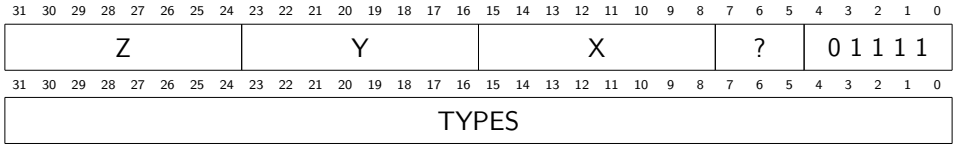
Each coordinate is cropped to the bits in its matrix dimension.

TYPE is cropped to [type bits].

If row (Z, Y) is outside the defined matrix, data will still be written but only accessible by read instructions.

If X is outside the defined matrix, nothing will happen.

Write Row of Types



Format

write_types(Z, Y, X, TYPES)

Purpose

To write one row (or as many can fit an instruction) of types.

Description

TYPES is a list of types in little-endian order. It is either [matrix width] or as many can fit 224 bits in length. Each type is [type bits] long.

The types are written to Cell BRAM A at row (Z, Y). They are offset so the first type is written to position X within the row. Types offset to [matrix width] or more are discarded.

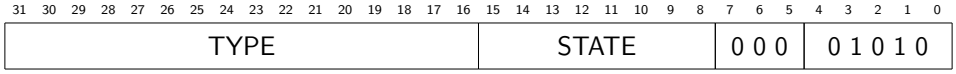
The length of TYPES varies depending on [matrix width] and [type bits]. It is sent as one continuous piece spanning multiple words. The instruction length field is adjusted accordingly.

Notes

Each coordinate is cropped to the bits in its matrix dimension.

If row (Z, Y) is outside the defined matrix, data will still be written but only accessible by read instructions.

Fill Cells



Format

fill_cells(STATE, TYPE)

Purpose

To set the state and type of all cells.

Description

STATE and TYPE is written to each cell in Cell BRAM A.

Notes

STATE is cropped to [state bits].

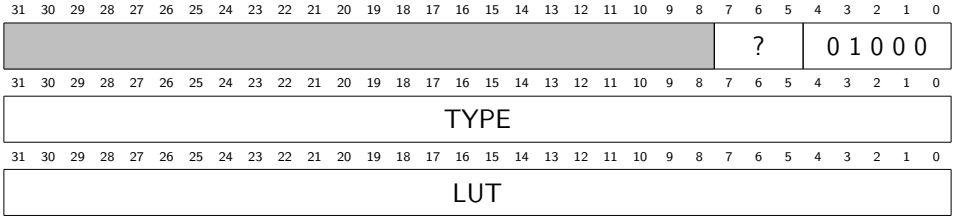
TYPE is cropped to [type bits].

This instruction takes [matrix depth] * [matrix height] cycles.

5 Cellular Automaton Instructions

This section covers all instructions affecting the Cellular Automaton. This includes writing look-up tables, configuring the CA, running the CA, and reading back the new states.

Write LUT



Format

write_lut(LUT, TYPE)

Purpose

To write a type to lookup table conversion entry.

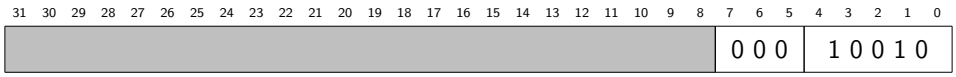
Description

LUT is written to LUT BRAM at address TYPE. The length of LUT varies depending on [matrix depth]. It is sent as one continuous piece spanning multiple words. The instruction length field is adjusted accordingly.

Notes

TYPE is cropped to [type bits].

Configure



Format

config()

Purpose

To configure the sblock matrix.

Description

The cells in Cell BRAM B are fetched along with the LUTs corresponding to each of their types. The LUTs and states are then written to the sblocks.

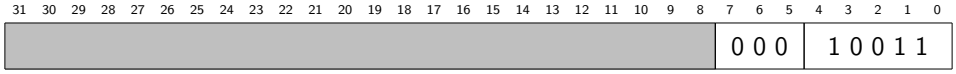
Notes

The execution time (T) for this instruction depends on [matrix depth] (M_Z), [matrix height] (M_Y) and [lut configuration bits] (LUT_{CB}).

$$T_{3D} = M_Z M_Y \frac{128}{LUT_{CB}} + 2$$

$$T_{2D} = M_Y \frac{32}{LUT_{CB}} + 2$$

Readback



Format

readback()

Purpose

To read back cell states from the sblock matrix.

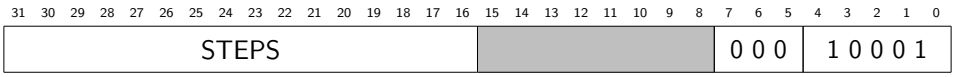
Description

The states of all sblocks are written to Cell BRAM B. Types in Cell BRAM B are preserved.

Notes

This instruction takes [matrix depth] * [matrix height] cycles.

Step



Format

step(STEPS)

Purpose

To perform updates of the sblock matrix.

Description

The sblock matrix is updated STEPS times. After each step, the number of live cells (state equals 1) are counted and stored in the Live Count buffer.

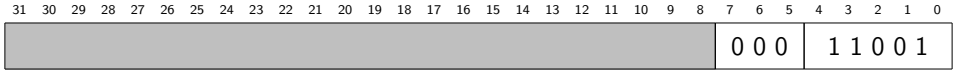
Notes

This instruction takes STEPS + 1 cycles.

6 Control Flow Instructions

This section covers all instructions that are related to the program memory. This includes those for storing, starting and exiting programs, in addition to control flow within the programs.

Break



Format

break_out()

Purpose

To break out of a running program and restore control to the host.

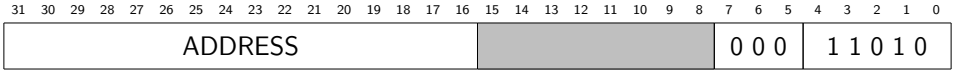
Description

The Fetch module exits [read from memory] mode and enters [read from communication] mode.

Notes

This has no effect if the Fetch module is already in [read from communication] mode.

Store



Format

store(ADDRESS)

Purpose

To begin storage of a program to internal memory.

Description

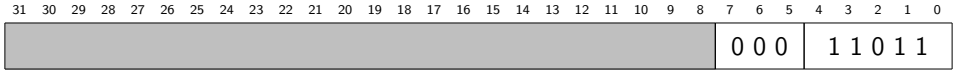
The Fetch module exits [read from communication] mode and enters [save to memory] mode. The next instruction will be saved at address ADDRESS, and then each address thereafter.

Notes

This will be saved as a nop if the Fetch module is already in [save to memory] mode.

ADDRESS is cropped to [program counter bits].

End



Format

end()

Purpose

To end storage of a program to internal memory.

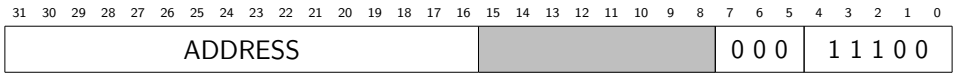
Description

The Fetch module exits [save to memory] mode and enters [read from communication] mode.

Notes

This will be parsed as a nop if the Fetch module is already in [read from communication] mode.

Jump



Format

jump(ADDRESS)

Purpose

To begin execution of or jump within a program stored to internal memory.

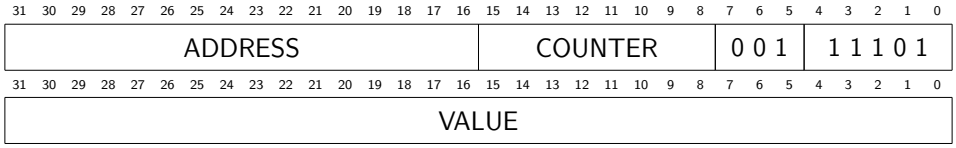
Description

If the Fetch module is not in [read from memory] mode, it exits [read from communication] mode and enters [read from memory] mode. The program counter is then updated so the next instruction is the one at address ADDRESS.

Notes

ADDRESS is cropped to [program counter bits].

Jump Equal



Format

jump_equal(ADDRESS, COUNTER, VALUE)

Purpose

To begin execution of or jump within a program stored to internal memory if a counter matches a value.

Description

If counter COUNTER is equal to VALUE, this instructions is exactly like jump(ADDRESS). Otherwise, it is discarded.

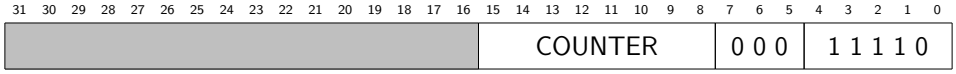
Notes

Accessing counter [counter amount] or higher yields undefined behavior.

ADDRESS is cropped to [program counter bits].

VALUE is cropped to [counter bits].

Increment Counter



Format

counter_increment(COUNTER)

Purpose

To increment a counter.

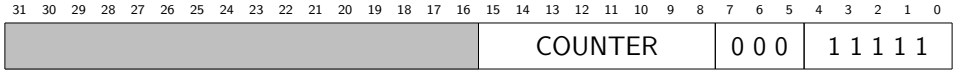
Description

Counter COUNTER is incremented by 1. If counter COUNTER is at maximum, it instead becomes 0.

Notes

Accessing counter [counter amount] or higher yields undefined behavior.

Reset Counter



Format

counter_reset(COUNTER)

Purpose

To reset a counter.

Description

Counter COUNTER is set to 0.

Notes

Accessing counter [counter amount] or higher yields undefined behavior.

Appendix C

Specialization Project Report

Reservoir Computing with Cellular Automata and Spiking Neural Networks

Øyvind Robertsen
Department of Computer and Information Science
Norwegian University of Science and Technology
Sem Sælands Vei 9, 7491 Trondheim, Norway
oyvinrob@stud.ntnu.no

***Abstract*—Reservoir Computing (RC) is a new and interesting approach to machine learning in which temporal input is imposed as perturbations on a dynamic reservoir and output is read out by performing a linear classification of reservoir state some time after the initial perturbation. While Recurrent Neural Networks are a common choice of reservoir, any dynamical system exhibiting the ability to let perturbations "echo" through the system over time and to respond distinctly to different inputs could be used. In this paper we are interested in systems where the output is spiking in nature, such as biological neurons and cellular automata (CA). Implementations of RC-systems with spiking reservoirs typically convert spiking data to analog values before performing classification. This conversion can introduce unwanted biases to the system, as well as being potentinally expensive to implement in hardware. In this paper, we introduce a linear classifier operating in the spiking domain, the Simplified Spiking Neural Network (SSNN), that avoids this conversion. To show that this model is viable, we implement a software simulation of an RC-system consisting of a one-dimensional, uniform cellular automata and a small SSNN and train the network so that is able to distinguish between the dynamics emerging in the reservoir from different initial states.**

We also implement a novel RC-system on an FPGA, using a Dynamical System with Dynamical Structure $(DS)^2$ as a reservoir and an SSNN as a readout layer. $(DS)^2$ systems are interesting in the context of reservoir computing because they have the ability to self-regulate and adapt their own dynamics based on their environment. This allows for agents that are capable learning on their own, as opposed to being trained. We implement the $(DS)^2$ reservoir as a non-uniform CA in which the state transition function in each cell is subject to development over time.

I. Introduction

In recent years, research into computation using non-traditional physical mediums and paradigms, so called unconventional computing, has seen increased interest. With challenges currently facing traditional architectures, such as the von Neumann bottleneck and ensuring continued scalability and reliability, unconventional computation presents possible solutions from a new perspective. Instead of designing architectures top-down, by composing complex units and orchestrating their interaction, a bottom-up approach is employed, where complex behaviour emerges from local interactions between simple units. Cellular computing, introduced by Sipper in [1], is an example of one such paradigm. These same principles are also used to explore computational capabilities in unconventional materials, as shown in [2].

Reservoir computing (RC) is a novel approach to machine learning and intelligent systems, in which input data is imposed as perturbations on a dynamic system (the reservoir) and output is generated by performing a linear classification of the reservoir state [3]. With RC as a field originating from the study of Recurrent Neural Networks (RNNs), these are a common choice of reservoir in RC implementations [3]. The focus of this paper however, is the use of developmental systems [4] as a reservoir and how to perform readout from such a system. Specifically, we are interested in developmental systems based on non-uniform Cellular Automata (CA) [5]. Where traditional CAs have fixed transition functions throughout a simulation, determining how the state of a cell should update based on the state of its neighbors and itself, these functions are subject to change in developmental systems. Based on growth rules and environment feedback, developmental systems change their dynamical behaviour. This ability is particularly interesting in the context of unconventional computation, as it allows for architectures with self-organizing, self-repairing and self-scaling properties. A major advantage of CA-based reservoirs over ones based on RNNs, is that they allow for efficient implementation in hardware, since no expensive floating-point operations are required.

In RC systems where the reservoir output is spiking in nature, such as (developmental) cellular automata and biological neurons, spiketrains from the reservoir is typically converted to analog values before being processed by a linear readout layer. This is usually done by resampling spiketrains and applying exponential filtering before classifying the output with linear/ridge regression [3]. Converting to, and performing classification in the analog domain, introduces expensive floating-point operations to the RC pipeline. In the interest of achieving efficient end-to-end simulation of a CA-based RC system in hardware, a readout layer that can process spiking data without need for conversion is needed.

In this paper we propose a simplified spiking neural network model (SSNN) suitable for use as a readout layer in an RC system. This model is capable of classifying reservoir dynamics based on unconverted spiketrain data. As a proof of principles, we implement a software simulation of an RC system consisting of a one-dimensional, homogenous CA and a SSNN, and show that the system can compute XOR.

We also implement a RC-machine based on a cellular

developmental system on reconfigurable hardware, a field-programmable gate array (FPGA), where the readout layer is implemented using the above mentioned SSNN model.

This paper is organized as follows: Section II gives relevant background and Section III describes the Simplified Spiking Neural Network model. In Section IV the cellular, developmental RC-machine implemented in hardware is presented. Section V describes the software and methodology used to carry out the experiment described in Section VI. Finally, Section VII offers a short conclusion, followed by an overview of future work in Section VIII.

II. Background

A. Reservoir Computing

Artificial Neural Networks (ANNs) are a commonly used computational model in machine learning and bio-inspired computing. Simple, feed forward ANNs lend themselves well to problems were data can be spatially correlated, such as classification. Many real world problems however, are temporal in nature. Recurrent neural networks (RNNs) have been shown to be powerful tools for solving temporal problems such as stock market prediction [6], learning context free/sensitive languages [7] and speech synthesis [8]. Training RNNs is computationally expensive and often requires application specific adaptions of generalized training algorithms in order to reliably converge [9]. Several techniques have been proposed that circumvent problems related to training, such as Echo State Networks [10] (ESNs), Liquid State Machines [11] (LSMs) and Backpropagation Decorrelation learning [12]. These all share the common feature of only training weights of the output layer of the network, while leaving the hidden layers of the network untrained or simply subject to weight scaling. In [13], Verstraeten et al. propose that systems based on this idea should be unified under the term reservoir computing (RC).

In general, reservoir computing as a term describes any computational system where a dynamic reservoir is excited by input data and output is generated by performing classification/regression over reservoir state. Figure 1 shows the basic architecture of any reservoir computing system. With its origins in research on various types of recurrent neural networks and training thereof, the reservoir in RC systems is often represented as an RNN [13]. However, any dynamic system capable of eventually forgetting past perturbations and of responding distinctly to different perturbations, can in principle be used. Snyder et al. [14] investigate using Random Boolean Networks, Yilmaz uses Cellular Automata [15] and Fernando et al. use a bucket of water [16].

B. Cellular Automata and EvoDevo Systems

John von Neumann introduced cellular automata (CA) as a discrete computational model based on local interaction of cells on a grid of finite dimensionality [17]. At any timestep t during the simulation, each cell in the grid is in one of a finite number of states. The state of any cell at time $t+1$ is computed as a function of the cells and its neighboring cells current states. Under this simple scheme, von Neumann showed that advanced properties such as autonomous self-reproduction is possible. In later years, much research has gone into both qualitative and quantitative analysis of the capabilities of CAs as an abstract model of computation [18] [19].

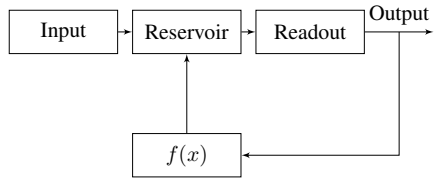


Fig. 1. Basic overview of an RC system.

Artificial evolution is an important tool in biologically inspired computing. Inspired by Darwin's theories of evolution, artificial evolution is often used to solve problems by encoding potential solutions as individuals in a population and applying mechanisms such as reproduction, recombination, mutation and selection on said population to breed better solutions with regard to some fitness function [20]. Of particular interest with regards to the work being presented in this paper, is the use of evolution to explore the rule-space of CAs [21]. In [22], Tufte describes how artificial evolution and developmental techniques can be combined to grow multicellular systems for which both structure and behaviour are emerging properties, so called Dynamical Systems with Dynamical Structures ($(DS)^2$) [23]. Where dynamical systems, such as Random Boolean Networks and CAs, have fixed structural topology and state transition functions, $(DS)^2$ systems allow these properties to undergo a developmental process. Through this process, these systems can self-regulate and adapt their dynamics and change their possible trajectories through state space. In this paper and in future work, we are interested in exploring the capabilities of these kinds of systems when used as reservoir.

C. Spiking Neural Networks

Artificial neural networks can be grouped into three generations, based on the characteristics of their base computational unit, the neuron. The first generation, based on McCulloch-Pitts neurons [24], simple threshold gates, allows for universal computation on digital input/output values. In the second generation, neurons apply a non-linear, continuous activation function on the weighted sum of their inputs.

The third generation of networks bases itself on spiking neurons, which model the interaction between biological neurons more closely. In this model, a neuron v fires when its potential P_v exceeds a threshold θ_v. The potential is, at any time, the sum of the postsynaptic potentials, resulting from firing of presynaptic neurons. The contribution of a spike from presynaptic neuron u at time s to the potential P_v of postsynaptic neuron v is given by $w_{u,v} \cdot \varepsilon_{u,v}(t-s)$, where $w_{u,v}$ is a weight representing the strength of the synapse connecting u and v, and $\epsilon_{u,v}(t-s)$ models the response of the

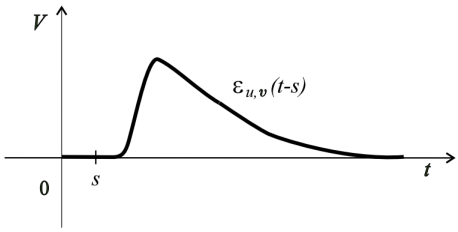


Fig. 2. Common spike response function shape, figure taken from [25].

spike as a function of time passed since the spike occurred. A synapse can be both excitatory and inhibitory, meaning that its contribution to the total potential P_v can be both positive and negative. A biologically plausible response function is shown in figure 2. From a machine learning perspective, the trainable part of a spiking neural network, is the weight $w_{u,v}$, determining to what degree spikes from a neuron u influences the potential of neuron v.

In [25], Maass shows that spiking neurons are at least computationally equal to the models used in generation one and two, and that they can also be more efficient in terms of neurons required to compute a function. SNNs also have the required attributes to be used as a reservoir in an RC system [26]. In order to be able to create an efficient implementation of a RC-machine based on a cellular developmental system as described in IV, being able to perform classification in the spiking domain is essential.

III. Simplified Spiking Neural Network

In this section, we present a simple neural model inspired by the SNN model outlined in Section II-C. The basic architecture of each neuron is shown in Figure 3.

Similarily to their biological counterparts, the potential for a neuron to fire increases as more spikes come in. For each incoming edge/synapse, a neuron has a separate counter/weight-pair. Counters are incremented every time a spike comes in via the corresponding synapse. Weights act as thresholds. For each synapse, they represent the minimal number of incoming spikes required for the neuron to fire. An incoming spike to a counter that is already equal to its weight, will not cause the counter to increment. When all counters are equal to their weights, the neuron fires, and the counters reset.

Counters are also susceptible to decay over time. If at any timestep a spike is not occurring for some incoming edge, the corresponding counter is decremented.

IV. $(DS)^2$ RC-Machine

An FPGA-prototype of a $(DS)^2$ RC-machine with a SSNN readout layer was implemented as a part of this project. In this section, we provide an overview of the functionality and implementation of the reservoir and readout components of the prototype, as well as numbers on resource usage and scalability.

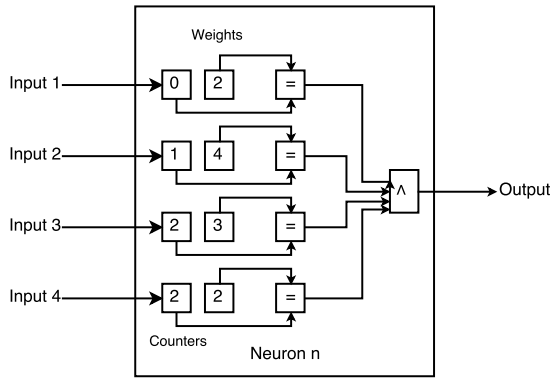


Fig. 3. Block diagram for the base computation occuring in the simplified spiking neurons.

A. Developmental Reservoir

The implemented reservoir is a developmental system based on the work presented by Tufte in [27]. In this model, the functional behavior of each cell on a regular grid of finite dimensionality is decided by its type. To model emergence of both structure and behaviour, the rules governing growth and structural development, the genome, is applied to all cells every so called Development Step (DS). These rules are expressed as a condition and a result, with the condition being dependent on both the current type and state of the cells in the von Neumann neighborhood [1] of the cell being developed. Growth is in other words encoded in the genome in much the same way as transition functions are in traditional CAs. It is important to note that the genome is not a blueprint describing exactly how the final result should look, but rather a description of how to build a system.

Each DS consists of a set number of State Steps (SS). Every state step, the state of each cell is updated. The type of the cell determines its behavior, or which transition function to use to evaluate cell states every state step. As in traditional CAs, the state of cells in the von Neumann neighborhood is used as input to the transition function.

A system operating under the regime described in this section, will essentially be a heterogeneous CA where the transition function used in each cell changes over time. Since both the type and state (environment) is taken into account during development, the system can adapt to and self-regulate its own behavioral dynamics. This property is of particular interest in the context of machine learning, as it allows for agents that are capable of learning, as opposed to the current norm, agents that can be trained.

Figure 5 shows the development of a reservoir starting from a single green cell, under the growth rule from Figure 6. For each growth rule, the label underneath the neighborhood

[1]The cells directly north, east, south and west of a cell, as well as the cell itself.

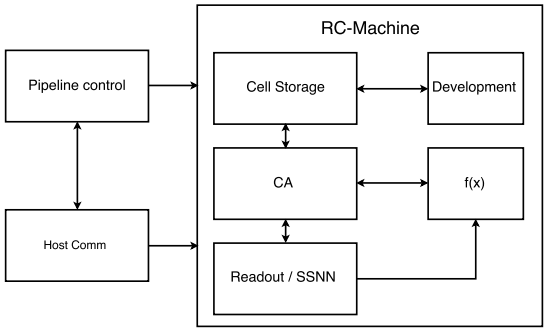


Fig. 4. High-level system architecture for the $(DS)^2$ RC-machine

indicates the direction from which the cell should grow. For instance; Gw, Grow west, indicates that the rightmost cell in the neighborhood should be copied into the center cell. Analogous rules apply for Grow north and east, as well as a special case for Grow south. A cell with a green northern neighbor will turn into a red cell, while one with a red northern neighbor will turn green. The functional difference between the red and green cell type is that they use different state transition functions. For the neighborhoods not enumerated here, the center cell remains unchanged. A subset of statesteps for DS 1 and DS 2 is shown to illustrate how cell type affects behavior. For the sake of simplicity, this rule does not take the state of neighboring cells into account when performing a development step, only their type.

B. FPGA implementation

An FPGA-based implementation of a developmental system as described here was completed and verified by Lundal in [28]. Figure 4 shows a high-level overview of the system architecture. The system is implemented as a three-stage interlocked pipeline, consisting of Fetch, Decode and Execute stages. Cell states and types is stored in On-chip Memory, so called Block RAM (BRAM) in the Cell Storage module. At each execute stage, one of two modules, either the Development or the CA module, operates on the data. Development steps happen in the development module, where growth rules are stored in BRAM. Whenever a DS instruction occurs, all cells are fetched from the Cell Storage and tested against all rules. Similarily for state steps, cells are moved to the CA module, a two- or three-dimensional grid of Sblocks [29], allowing for fully parallel execution of the behavior of the system. Each sblock is connected the sblocks in the von Neumann-neighborhood around it, receiving their states as input. The type of a cell determines the functionality configured in the sblock LUT.

New hardware, equipped with Convey Wolverine WX-2000 coprocessors (see micron.com) based on the Xilinx Virtex-7 XC7V2000T FPGA (see xilinx.com), has since been aquired and the system has been ported to work on this new platform. In an effort to further modernize the codebase, the system has been partially ported from VHDL to Chisel, a Hardware Description Domain Specific Language implemented in Scala (see chisel.eecs.berkeley.edu).

The architecture of the developmental system as implemented by Lundal remains largely unchanged. To allow the readout layer to process data in real time, without the need for an intermediate instruction between state steps, support has been added for marking sblocks as output blocks. The state from all output blocks is wired directly to the readout layer. This happens at synthesis time, as the amount of FPGA resources required for making it possible to reconfigure which blocks are used as output blocks at runtime is immense. For large reservoirs, the routing of state-signals from sblocks to the readout layer will require a considerable amount of resources in itself, making it necessary to limit the number of output sblocks. While this will limit the readout layer's "view" into the dynamics of the reservoir, the hope is that the reservoir will adapt around this, based on the feedback it receives from the readout layer.

C. Readout

Based on the basic description of each neurons computational functionality in Section III, a parameterized FPGA-implementation was created to perform realtime classification of the readout from the developmental reservoir described in the previous section. Given a description of a network topology (i.e. the number of neurons in each layer and the number of output cells in the reservoir), and weights for all neurons, hardware is generated accordingly, as shown in Figure 7. For each layer, a module encapsulating the neurons in that layer is synthesised, with each neuron being modeled as a separate entity within the layer. Neurons are implemented very similar to how they are depicted in Figure 3. Incoming spikes are counted in a set of registers, one for each neuron in the preceeding layer, which are used to determine wether or not the neuron should fire by comparing register values with statically configured weight values. In addition, circuitry for resetting the counters after a neuron fires has been added.

Layers in the SSNN form a pipeline, through which data propagates in sync with the state steps of the reservoir. For two layers in the SSNN, A and B, where B directly follows A in the pipeline, the output from neurons in layer A at time t is used as input for the neurons in layer B at time $t+1$. As mentioned, all neurons are synthesised with one counter per neuron in the preceeding layer. This means that all networks are implemented fully connected. That is, all neurons in one layer are connected to all neurons in the next. Connections can be pruned by setting a weight value of zero.

D. Resource Usage

Table I shows resource usage for different configurations of the developmental system in the RC-machine. The percentage utilization is based on the total amount of resources available on the Virtex-7 XC7V2000T FPGA. We see that for the configuration with $96 * 96 = 9216$ cells, $\sim 20\%$ of LUTs

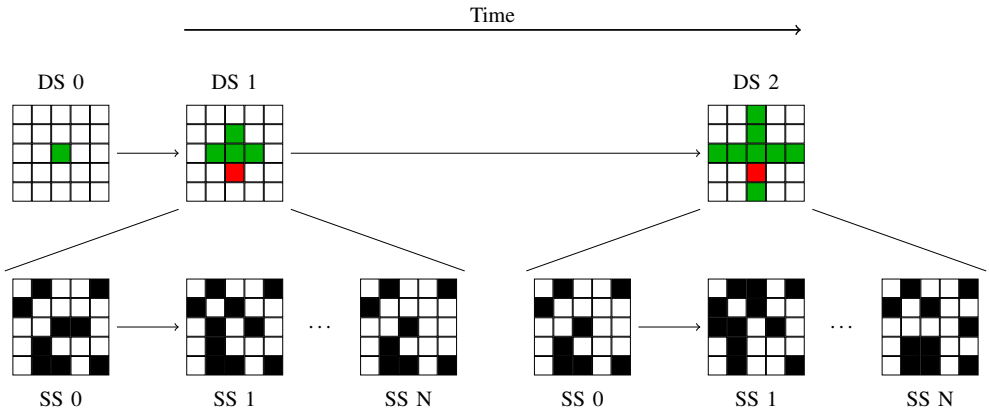


Fig. 5. Development starting from a single green cell using the growth rule in Figure 6.

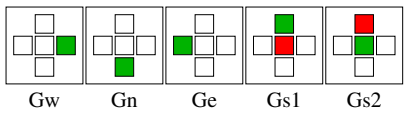


Fig. 6. Growth rules for a cellular developmental system where cells are either empty or of the green type.

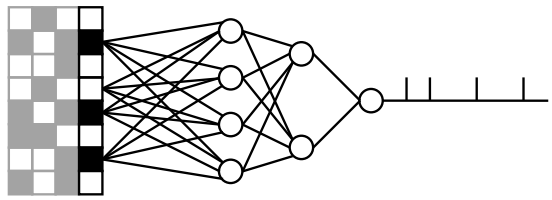


Fig. 8. At each timestep, the state of the CA is fed to the input-layer of the SSNN, which propagates updates through the network layers in a pipelined fashion.

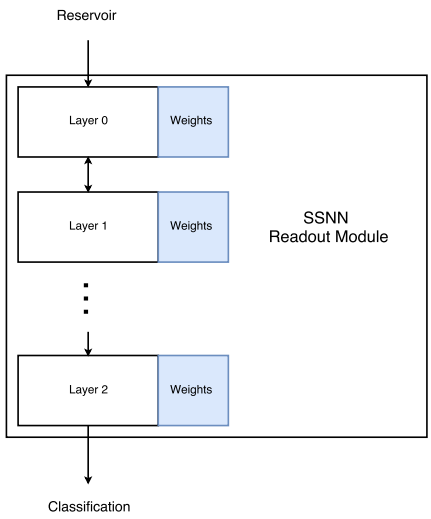


Fig. 7. HW architecture for the SSNN readout module.

and $\sim$ 24% of available BRAMs are utilized. It should, in other words, be possible to scale the developmental system up to approximately 36000 cells on this hardware. In a full RC-system, some resources will also need to be used by the SSNN, limiting the maximum size of the reservoir further.

V. Methodology

To demonstrate the viability of a SSNN as a readout-layer in a RC-system, a simple software simulation was implemented in the Python programming language. The implementation consists of three modules; a one-dimensional, homogenous, binary CA, an SSNN implementation and a genetic algorithm. Figure 8 shows how the CA and the SSNN is connected.

A. Cellular Automata Module

The CA consists of an array of 0s and 1s, and a step function that, given a CA-array, performs a single transition step on it. As the CA is uniform, the same development rule is applied to each cell every development step. Since the state of the CA is stored in such a simple format, perturbing the system with input is as simple as replacing the array as a whole or modifying parts of it betweeen timesteps.

For the work presented in this paper, CAs 32 cells wide was used, with rule 90 (see Table II) as the transition function. To

TABLE I
$(DS)^2$ RC-MACHINE FGPA RESOURCE USAGE.

Cell Count	LUTs Total	LUTS %	Registers Total	Registers %	BRAMs total	BRAMs %
8×8	194624	15.93	208690	8.54	227.5	18.91
64×64	224442	18.37	221061	9.05	280.0	23.28
72×72	230094	18.84	223355	9.14	286.5	23.82
96×96	251219	20.56	231616	9.48	308.5	25.64
$4 \times 4 \times 4$	195360	15.99	209171	8.56	232.5	19.33
$16 \times 16 \times 16$	249320	20.41	218109	8.93	261	21.70

TABLE II
RULE 90

Neighborhood	111	110	101	100	011	010	001	000
New state	0	1	1	0	1	1	1	0

TABLE III
GA HYPERPARAMETERS

Population size	50
Crossover rate	0.5
Mutation rate	0.4
Crossover function	Braid
Mutation function	Per Genome

accurately model the FPGA implementation, only a subset of the cells are used as input to the readout layer.

B. SSNN Readout Module

The model described in III is implemented as an array of neurons for each layer in the desired network topology. Each neuron is represented as an array of counters and an array of weights, one weight and one counter for each neuron in the previous layer. As in the hardware implementation, networks are fully connected by default, allowing for connections to be pruned by setting a weight value of zero. The simulation also imitates the pipelined dataflow used in the FPGA implementation.

C. Genetic Algorithm

A genetic algorithm was used to search the space of possible weight configurations of the SSNN. The hyperparameters used are shown in Table III.

Each individual in the population represents one possible weight configuration for the topology being optimized. The genotype of each individual is simply a matrix of numbers, each one representing one weight. Crossover of two parent individuals is done by braiding their layers, so that the child has layer one from parent one, layer two from parent two, layer three from parent one, et cetera. Individuals are subject to random mutations performed per genome with some probability p. An individual chosen to be mutated, will have a randomly chosen weight changed, with equal probability of incrementing or decrementing the weight.

Fitness for an individual is calculated by comparing spiketrains output by the network when fed with test data, with the expected spiketrains for that data.

VI. EXPERIMENTS AND RESULTS

Using the software simulation setup described in Section V, we wish to show that the SSNN model is viable to use as a readout layer in an RC system. For this to be the case, it needs to be possible to train the network to produce distinct spiketrains given different sequences of readouts from the reservoir. In other words, given desired output spiketrains for different input perturbations to the reservoir, it needs to be possible to train the SSNN to correctly classify the temporal dynamics arising from the perturbations and to produce the corresponding output spiketrain.

To verify that this is possible, we encode the four different input cases of a two-bit XOR-operation, $00, 01, 10, 11$ as four distinct initial states for the CA and let the presence or absence of a spike from the SSNN output node at given timesteps after some number of initial number of timesteps indicate result of the XOR operation. Specifically, for each of the initial states, the CA/SSNN was allowed to run for 200 timesteps before the output from the SSNN was sampled ten times at 25 timestep intervals. In other words, an optimal weight configuration for the SSNN is one whose output samples are ten zeroes, ten ones, ten ones and ten zeroes for the four initial states respectively. No specific mapping scheme was used to map from XOR-inputs to their respective initial states. Rather at the start of each experiment, four random 32-wide bitstrings are chosen to represent the inputs.

For this experiment, the SSNN was configured with two layers, a two-neuron input-layer fully connected to 8 of the 32 cells in the CA, and an output layer with a single neuron. As described in Section V-C, a genetic algorithm (GA) is used to search through the space of possible weight configurations. Being an optimization algorithm, the GA does not guarantee that an optimal solution will be found. For several of the configurations run during this experiment, optimal solutions were however found, showing that SSNNs are viable as readout layers in RC-systems.

Depending on how large a subsection of the CA's cells' states are fed as input to the SSNN, the convergence rate for the search varies. By using fewer cells, the size of the search space is reduced, along with the amount of dynamic behavior captured, making it harder/potentially impossible to distinguish the dynamics emerging from one initial state from the others. Use too many, and the search space size increases, making it more likely that the search gets stuck in a local maximum. Optimal solutions were found most frequently when 8 of the total 32 cells in the CA were used as output.

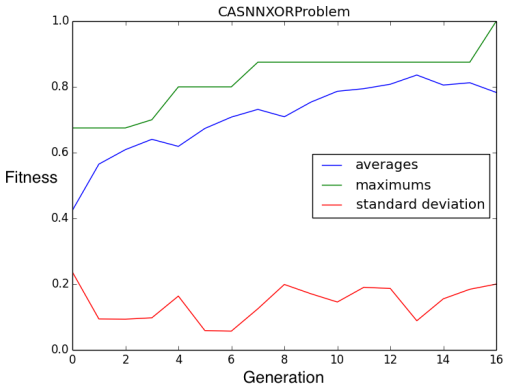


Fig. 9. Average, maximum and standard deviation of fitnesses for one run of the XOR experiment.

A plot of average and maximum fitnesses as well as fitness standard deviation for one run where an optimal solution was found is shown in Figure 9.

VII. Conclusion

In this paper, we present a Simplified Spiking Neural Network model, with the goal of using it as a readout layer in RC systems where the reservoir output is spiking in nature. Through experiments simulated in software, we show that it is possible to train a network based on this model to correctly classify temporal dynamics emerging in a cellular automaton. We also review work done towards an FPGA-implementation of a reservoir computing system consisting of a $(DS)^2$ cellular reservoir and a spiking neural network as a readout layer.

VIII. Future Work

While this paper shows that the SSNN is suitable for use in an RC system, using a genetic algorithm for training is less than desireable, and alternative training schemes should be investigated. Two possible alternatives are SpikeProp [30], a version of the backpropagation training algorithm adapted for spiking neurons, and spike-timing-dependent plasticity [31], an on-line training scheme where synapses over which spikes arrive right before the neuron fires are strengthened, while those over which spikes arrive right after thee neuron has fired are weakened.

The FPGA-based implementation of the SSNN is very simple so far, and needs more work to be practical in use. As mentioned in Section IV-C, weights are set statically at synthesis-time. To allow for easy reconfigurability, they should instead be stored in BRAM. As it is implemented now, the output from the SSNN is not stored anywhere, nor fed back into the reservoir. To facilitate analysis of the results, the output should be buffered for some time, allowing them to be transferred to the host program. It should also be possible to mark specific cells as input cells, into which the output from the SSNN is fed continuously. Since the reservoir is a $(DS)^2$ system, this will allow it to adapt according to the performance of its own dynamics.

Scalability is another issue with the current SSNN implementation. For a configuration with 4000 output cells in the reservoir and 64 neurons in the first layer of the SSNN, $4000 \times 64 = 256000$ registers will be inferred by the current implementation. To allow for flexible experimentation with reservoir sizes, number of output cells and SSNN topologies, the implementation should be changed so that it no longer generates one circuit per neuron per layer, but instead generates a limited number of neuron-circuits per layer. This way, computation in each layer happens over several clock cycles, but more FPGA-resource can be used by the reservoir.

Regarding the RC-machine described in Section IV, while work remains in order to make both the reservoir and readout components complete, much of the foundation is now in place. An interesting use case of the system is to use it to explore the space of growth rules for $(DS)^2$ systems. Rules that allow systems with interesting properties, such as the ability to self-reproduce or robustness, are of particular interest. These rules can potentially be used to guide the growth and development of reservoirs based on biological neurons, such as those used in the NTNU Cyborg project (see https://www.ntnu.edu/cyborg).

References

[1] M. Sipper, "Emergence of cellular computing," *Computer*, vol. 32, no. 7, pp. 18–26, 1999.

[2] J. F. Miller, S. L. Harding, and G. Tufte, "Evolution-in-materio: evolving computation in materials," *Evolutionary Intelligence*, vol. 7, no. 1, pp. 49–67, 2014.

[3] B. Schrauwen, D. Verstraeten, and J. Van Campenhout, "An overview of reservoir computing: theory, applications and implementations," *Proceedings of the 15th European Symposium on Artificial Neural Networks*, no. April, pp. 471–82, 2007.

[4] S. Kumar and P. Bentley, eds., *On Growth, Form and Computers*. Elsevier, 2003.

[5] G. Tufte and P. C. Haddow, "Towards development on a silicon-based cellular computing machine," *Natural Computing*, vol. 4, no. 4, pp. 387–416, 2005.

[6] S. Lawrence, C. L. Giles, S. Fong, S. Lawrence, and A. C. A. Tsoi, "Noisy Time Series Prediction using Recurrent Neural Networks and Grammatical Inference," *Machine Learning*, vol. 44, no. 1, pp. 161–183, 2001.

[7] F. A. Gers and J. Schmidhuber, "LSTM recurrent networks learn simple context-free and context-sensitive languages," *IEEE Transactions on Neural Networks*, vol. 12, no. 6, pp. 1333–1340, 2001.

[8] Z. Wu and S. King, "Investigating gated recurrent networks for speech synthesis," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 2016-May, pp. 5140–5144, 2016.

[9] B. Hammer and J. J. Steil, "Tutorial: Perspectives on learning with rnns," *Proc. ESANN*, no. April, pp. 357–368, 2002.

[10] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks," *GMD Report*, vol. 148, pp. 1–47, 2001.

[11] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: a new framework for neural computation based on perturbations.," *Neural computation*, vol. 14, no. 11, pp. 2531–2560, 2002.

[12] J. J. Steil, "Backpropagation-Decorrelation: Online recurrent learning with O(N) complexity," in *IEEE International Conference on Neural Networks - Conference Proceedings*, vol. 2, pp. 843–848, 2004.

[13] D. Verstraeten, B. Schrauwen, M. D'Haene, and D. Stroobandt, "An experimental unification of reservoir computing methods," *Neural Networks*, vol. 20, pp. 391–403, apr 2007.

[14] D. Snyder, A. Goudarzi, and C. Teuscher, "Computational capabilities of random automata networks for reservoir computing," *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, vol. 87, no. 4, 2013.

[15] O. Yilmaz, "Reservoir Computing using Cellular Automata," *arXiv preprint*, pp. 1–9, 2014.

[16] C. Fernando and S. Sojakka, "Pattern Recognition in a Bucket," *Advances in Artificial Life*, pp. 588–597, 2003.

[17] J. V. Neumann, *Theory of Self-Reproducing Automata*. Champaign, IL, USA: University of Illinois Press, 1966.

[18] C. G. Langton, "Computation at the edge of chaos: Phase transitions and emergent computation," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1-3, pp. 12–37, 1990.

[19] S. Wolfram, *Cellular automata and complexity: Collected papers*, vol. 45. 2003.

[20] J. H. Holland, *Adaptation in Natural and Artificial Systems*, vol. Ann Arbor. 1975.

[21] M. Mitchell, J. P. Crutchfield, and P. T. Hraber, "Evolving cellular automata to perform computations: mechanisms and impediments," *Physica D: Nonlinear Phenomena*, vol. 75, no. 1-3, pp. 361–391, 1994.

[22] G. Tufte, "The discrete dynamics of developmental systems," in *2009 IEEE Congress on Evolutionary Computation, CEC 2009*, pp. 2209–2216, 2009.

[23] G. Tufte and O. R. Lykkebø, "Evolution-in-Materio of a dynamical system with dynamical structures," *Proceedings of the Artificial Life Conference 2016*, 2016.

[24] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[25] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[26] G. Pipa and R. Cao, "Extended Liquid Computing in Networks of Spiking Neurons Supervisor Spiking Neurons," 2010.

[27] G. Tufte, "From Evo to EvoDevo: Mapping and Adaptation in Artificial Development," in *Evolutionary Computation*, ch. 12, 2009.

[28] P. T. Lundal, "The Cellular Automata Research Platform: Revised, Rebuilt and Enhanced," *Master Thesis*, no. June, 2015.

[29] P. Haddow and G. Tufte, "An evolvable hardware FPGA for adaptive hardware," in *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, vol. 1, pp. 553–560, 2000.

[30] S. M. Bohte, J. N. Kok, and H. La Poutré, "Error-backpropagation in temporally encoded networks of spiking neurons," 2002.

[31] H. Markram, W. Gerstner, and P. J. Sjöström, "Spike-timing-dependent plasticity: A comprehensive overview," *Frontiers in Synaptic Neuroscience*, vol. 4, no. JULY, pp. 2010–2012, 2012.

Appendix D

Project Readme

plan9

plan9 is an implementation of the Cellular Automata Research Platform (CARP) developed at NTNU targeting Convey WX coprocessors. It consists of a hardware component implemented using Chisel, Verilog and VHDL (residing in /hardware) and a software API written in C (/software). For instructions on synthesizing and deploying the hardware component, see `hardware/README.md`. Similarily for the API, `software/README.md`.

plan9-hardware

Synthesizing and building the plan9-hardware component requires the Convey PDK to be installed. The design is implemented in Chisel, with some VHDL blackboxes down the module hierarchy. All configuration and parameterization happens in CarpParameters.scala. The `ChiselParameters` trait is an interface specifying the parameters required for the design to synthesize. It also provides semi-sane defaults. Instead of changing the values in the trait directly, it is recommended to create a new object implementing the trait and change only the necessary values on a per-need basis, as shown below.

```
package plan9

object SpecificExperimentParams extends CarpParameters {
  SNNTopology = Array(4,2,1)
}
```

To have the new parameters be used during synthesis, `SpecificExperimentParams` has to be passed as a parameter when instantiating the Carp-module in Plan9.scala.

```
val carpParams = SpecificExperimentParams
...
val carp = Module(new Carp(carpParams))
```

Make targets

`make gen_verilog`

`make gen_verilog` invokes the Scala Build Tool and generates Verilog based on the Chisel code.

`make all`

Invokes the `gen_verilog` target before starting the Xilinx flow, incorporating the plan9-design into the Convey Application Engine wrapper and generating a bitfile. This takes a while.

`make release`

Moves the bitfile resulting from the previous run of `make all` to the `hardware.released` directory. Useful for archiving separate versions of the design.

`make install`

Deploys the most recently synthesized bitfile to the Convey personalities folder. This target requires the following variables to be set in phys/Makefile:

- `PERSONALITY`: Personality ID.
- `RELEASE_DIR`: Path to the folder into which `make release` moves bitfiles.
- `CNY_DIR`: Path to the root Convey directory, usually `/opt/convey`.

`make test`

Run all simulation tests in `src/test/scala/`.

Setup on NTNU servers

`moog.idi.ntnu.no` has been used for synthesis. The Convey cards are installed on `lobo.idi.ntnu.no`. This means that the bitfile resulting from `make all` will have to be manually moved between the two, instead of using the `make install` target. This can be done with the following commands:

```
$ scp moog.idi.ntnu.no:/home/<your_user>/<path_to_plan9>/\
    hardware.released/<release_tag>/ae_fpga.tgz ./
$ scp ae_fpga.tgz lobo.idi.ntnu.no:/opt/convey/\
    personalities/65002.0.0.4.0/
```

plan9-software

The plan9 software API is implemented in C. It depends on the Convey SDK, specifically `"wdm_user.h"`.

Make targets

`make all`

Build the library and all programs residing in `programs/`.

`make test`

Executes all programs in `programs/` whose filename start with `test_`.

`make clean`

Delete build artefacts.