

TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

---

**Development of a CAD tool and  
Hardware Design in Order to Execute  
Cellular Automata on a Reconfigurable  
Platform by non-FPGA-Conversant Users**

---

*Author:*

Emmanouil MYLONAKIS

*Thesis Committee:*

Prof. Apostolos DOLLAS

Assoc. Prof. Sotirios IOANNIDIS

Prof. Michael ZERVAKIS



*A thesis submitted in fulfillment of the requirements  
for the diploma of Electrical and Computer Engineer*

*in the*

**School of Electrical and Computer Engineering  
Microprocessor and Hardware Laboratory**

February 23, 2024



TECHNICAL UNIVERSITY OF CRETE

# *Abstract*

School of Electrical and Computer Engineering

Electrical and Computer Engineer

**Development of a CAD tool and Hardware Design in Order to Execute  
Cellular Automata on a Reconfigurable Platform by  
non-FPGA-Conversant Users**

by Emmanouil MYLONAKIS

Cellular Automata (CA) are Turing-Complete, discrete, computational models, invented by John Von Neumann and Stanislaw Ulam. It is a powerful mathematical tool, finding application to numerous scientific fields. Field-Programmable Gate Array (FPGA) Technology has been used for decades to speed up CA computations. In previous work, Nikolaos Kyparissas designed in his Technical University of Crete (TUC) M.Eng. Diploma Thesis a customizable framework and an architecture to accelerate CA computations, with neighborhoods as large as  $29 \times 29$ . In Kyparissas' work the initialization of the machine and the customization of the framework have to be manually re-defined for every different CA model, and the design placed and routed with the CAD tools of the FPGA vendor, Xilinx. In the present thesis we extend that work so that the user does not need to write code for the hardware implementation or go through the Xilinx CAD tools for placement and routing. A re-programmable structure of the framework has been introduced, while a new CAD tool, developed in the present thesis, drives the design at the software level. Finally, a Graphical User Interface (GUI) environment has also been developed to help the user define CA neighborhoods without having to enter one-by-one the as-many-as 841 ( $29 \times 29$ ) weights.



TECHNICAL UNIVERSITY OF CRETE

# *Abstract*

School of Electrical and Computer Engineering

Electrical and Computer Engineer

**Development of a CAD tool and Hardware Design in Order to Execute  
Cellular Automata on a Reconfigurable Platform by  
non-FPGA-Conversant Users**

by Emmanouil MYLONAKIS

Τα κυψελωτά αυτόματα (cellular automata - CA) είναι μία δομή των διακριτών μαθηματικών με σημαντικές υπολογιστικές ιδιότητες (Turing complete). Εφευρέθηκαν από τον John Von Neumann και τον Stanislaw Ulam και αποτελούν ένα σημαντικό μαθηματικό εργαλείο για μοντελοποίηση πληθώρας προβλημάτων σε πολλά και διαφορετικά επιστημονικά πεδία. Η τεχνολογία αναδιατασσόμενης λογικής (Field Programmable Gate Array - FPGA) έχει χρησιμοποιηθεί επί δεκαετίες για να επιταχύνει υπολογισμούς κυψελωτών αυτομάτων. Σε προγενέστερη εργασία, ο Νικόλαος Κυπαρισσάς στην Διπλωματική του Εργασία στο Πολυτεχνείο Κρήτης δημιούργησε ένα παραμετροποιήσιμο πλαίσιο εργασίας και μία αρχιτεκτονική για επιτάχυνση υπολογισμών CA με γειτονίες έως  $29 \times 29$ . Στην εργασία αυτή η αρχικοποίηση και η παραμετροποίηση του επιταχυντή πρέπει να οριστούν εκ νέου για κάθε διαφορετικό μοντέλο CA, και η σχεδίαση να περάσει από τα εργαλεία CAD του κατασκευαστή Xilinx για τοποθέτηση και διασύνδεση πόρων (Place and Route) της FPGA. Στην παρούσα διπλωματική εργασία επεκτείνουμε τα παραπάνω αποτελέσματα ώστε ο χρήστης να μην χρειάζεται να γράψει κώδικα ή να περάσει την σχεδίασή του μέσα από τα εργαλεία της εταιρίας Xilinx για τοποθέτηση και διασύνδεση πόρων. Αυτό επιτυγχάνεται μέσω αλλαγών στην υφιστάμενη αρχιτεκτονική που επιτρέπουν την χρήση της για διαφορετικά μοντέλα του χρήστη, και μέσω ενός γραφικού περιβάλλοντος που αλληλεπιδρά με το υλικό (hardware) του συστήματος για να φορτώνει νέες σχεδιάσεις. Μία επί πλέον γραφική διεπαφή (Graphical User Interface - GUI) έχει δημιουργηθεί στα πλαίσια της παρούσας διπλωματικής, ώστε ο χρήστης να μην χρειάζεται να ορίζει τα έως και 841 ( $29 \times 29$ ) βάρη στις γειτονίες ένα-ένα, αλλά με γραφικό τρόπο και μάλιστα αυτόματα όπου υπάρχουν συμμετρίες.



# *Acknowledgements*

First and foremost, I would like to express my deepest gratitude to my supervisor professor, Apostolos Dollas. He showed me trust by proposing the present subject to me in the first place. He was always by my side for support, beyond his role as a professor. He was compassionate during a difficult period of my life. Without him, I couldn't have carried through the present Thesis.

I would like to show my appreciation to my committee, Prof. Sotirios Ioannidis and Prof. Michael Zervakis.

I wish to acknowledge the efforts of the Technical University of Crete, which provides students with support, knowledge and numerous opportunities all of these years.

My sincere thanks go to Nickolas Kyparissas for his assistance and advises that he provided on technical issues.

My thanks are extend to all of my best buddies, in alphabetic order: Antonis Maragoudakis, George Exarchakos, George Koutroumpas, Giannis Koutroumpas, Ilias Vamvakas, Konstantinos Zacharopoulos, Konstantinos Stavroulakis, Konstantinos Koulaouzidis and Mixalis Charalampakis. Truly thank you fellas for all the support.

Heartfelt thanks to George Saltaris, my piano teacher and vocal coach. He showed me an alternative way of decompression, while it is a pleasure making conversations with him.

Special thanks to other friends and noteworthy persons, in alphabetic order: Andreas Chourdakis, Danae Rafti, George Gavrilakis, Maria Babila, Mary Tsourounaki, Mihalis Mylonakis, Niki Manioudaki, Paris Argyropoulos, Petros Pantelakis and Stella Migadaki. Additionally, to my basketball and bicycle teams.

Last but not least, all the words of the English dictionary are not enough, to express my deepest gratitude to my mother Litsa and grandmother Chrysi. Without the sacrifices that they made and the support that they offered, I wouldn't be able to make my dreams come true !



*"The Nature is a book written in the language of mathematics."  
~ Galileo Galilei*



# Contents

<b>Abstract</b>	<b>iii</b>
<b>Abstract</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>Contents</b>	<b>xi</b>
<b>List of Figures</b>	<b>xv</b>
<b>List of Algorithms</b>	<b>xvii</b>
<b>List of Abbreviations</b>	<b>xix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Thesis Contribution . . . . .	2
1.3 Thesis Outline . . . . .	2
<b>2 Theoretical Background</b>	<b>5</b>
2.1 Cellular Automaton Model . . . . .	5
2.1.1 Cellular Automata Classes . . . . .	7
2.1.2 The Game of Life . . . . .	8
2.2 FPGA Technology . . . . .	10
2.2.1 FPGA vs. Other Devices . . . . .	12
<b>3 Related Word and Motivation</b>	<b>15</b>
3.1 CAM Architecture (1984~2000) . . . . .	15
3.2 CEPRA Architecture (1994~2000) . . . . .	18
3.3 SPACE Architecture (1996) . . . . .	19
3.4 Kobori, Maruyama and Hoshino (2001) . . . . .	22
3.5 Phepls' and Islam's Framework (2023) . . . . .	23
3.6 Other Significant Approaches. . . . .	24

3.7	Thesis Approach and Motivation . . . . .	25
<b>4</b>	<b>The Baseline Hardware Architecture</b>	<b>27</b>
4.1	Top Level and System Specifications . . . . .	27
4.2	Memory Controller and Grid Representation . . . . .	30
4.3	System and Memory Initialization . . . . .	32
4.4	Grid Lines Buffer . . . . .	33
4.5	CA Engine . . . . .	35
4.6	Frame Extraction . . . . .	36
4.7	The Remaining Modules . . . . .	37
4.7.1	Graphics Controller . . . . .	37
4.7.2	Graphics Feeder . . . . .	38
4.7.3	Write Back . . . . .	38
4.7.4	Memory Access Arbitrator . . . . .	39
<b>5</b>	<b>Design of the re-programmable Framework</b>	<b>41</b>
5.1	Overview of Extended Architecture . . . . .	41
5.2	Frame Extraction and Speed Control . . . . .	42
5.3	CA Engine's Adjustments . . . . .	43
5.3.1	Supporting Totalistic Rules . . . . .	44
5.3.2	Expanding to Outer-Totalistic Rules . . . . .	46
5.4	Protocol Buffers . . . . .	48
5.4.1	Deserializing Data . . . . .	49
5.4.2	Serializing Data . . . . .	51
5.5	Assembling the complete picture . . . . .	52
<b>6</b>	<b>The CAD Tool to Drive the FPGA-based Accelerator</b>	<b>55</b>
6.1	Overview Of The Tool . . . . .	55
6.2	TCL Scripting . . . . .	56
6.3	CA Description Language (CDL) . . . . .	58
6.3.1	Interpreting the CDL . . . . .	60
	Totalistic Rules . . . . .	60
	Outer-Totalistic Rules . . . . .	61
6.4	Serializing/Deserializing Data . . . . .	62
6.5	Image Conversion . . . . .	63
<b>7</b>	<b>The Graphical User Interface CAD Tool to Describe the CA Model</b>	<b>65</b>
7.1	The GUI Environment . . . . .	65
7.2	User Options . . . . .	66

7.3	Configuring Weights . . . . .	68
7.3.1	Neighborhood Types . . . . .	70
7.3.2	Mirror Mode . . . . .	73
7.4	The Remaining Configurations . . . . .	74
<b>8</b>	<b>System Verification, Examples of Use, Evaluation, and Results</b>	<b>77</b>
8.1	Artificial Physics . . . . .	78
8.2	The Game Of Life . . . . .	80
8.3	The Hodgepodge Machine . . . . .	81
8.3.1	Experiments . . . . .	82
8.4	Discussion . . . . .	87
<b>9</b>	<b>Conclusions and Future Work</b>	<b>89</b>
9.1	Conclusions . . . . .	89
9.2	Future Work . . . . .	89
9.2.1	Application Examples . . . . .	90
	The Greenberg-Hastings . . . . .	90
	Anisotropic Rules . . . . .	91
	Hardware Approach . . . . .	92
9.2.2	Globalizing Accessibility and Improve Experience . . . . .	94
	<b>References</b>	<b>97</b>



# List of Figures

2.1	(a)-1D, (b)-2D, (c)-3D grid depiction. . . . .	6
2.2	Neighborhood types: (a) Non-weighted Moore, (b) Weighted Moore, (c) Non-weighted von Neumann, (d) Weighted von Neumann, (e) Non-weighted Custom, (f) Weighted Custom. . . . .	7
2.3	Game of Life transition rules. . . . .	9
2.4	FPGA's Hardware Architecture. The basic components. . . . .	10
2.5	Basic structure of Control Logic Blocks . . . . .	11
2.6	n-MAC operation. CPU vs FPGA . . . . .	13
3.1	CAM's hardware architecture. Basic computational loop with solid lines. Source [22] . . . . .	15
3.2	The form of the "gap" that should be filled by the transition function. Phase and channel parameters are non-local arguments. (see figure 3.1). Source [22] . . . . .	16
3.3	CAM's-8 hardware architecture. (a) A single module. (b) Each module is connected to the nearest ones. Source [24] . . . . .	17
3.4	CAM's-8 programming environment alongside a sample of a sound pulse. (Source: [24]) . . . . .	18
3.5	CEPRA's-1X environment alongside the derivative hardware. Source [28] . . . . .	19
3.6	One SPACE board. Source [30] . . . . .	20
3.7	Top level view of a lattice gas automaton. Source [30] . . . . .	21
3.8	Overview of hardware architecture. Case of $8 \times 16$ ( $k \times n$ ) PEs. $k$ and $n$ depend on the rule. Source [31]. . . . .	22
3.9	System overview. Source [31]. . . . .	23
4.1	Top Level View of the hardware architecture. Modules within dashes were modified in this Thesis. . . . .	28
4.2	Supported Grid Types: (a) Rectangular, (b) Cylindrical, (c) Toroidal. (Source: [12]) . . . . .	29
4.3	The Customizable Framework of the top level. <i>Generic</i> values are inherited by the sub-modules. . . . .	30

4.4	Handshake Mechanism for accessing Memory Controller. . . .	31
4.5	Grid representation in memory and burst addressing. $b_l$ is the number of bursts per line. Source: [12] . . . . .	32
4.6	Initialization procedure of the machine. . . . .	32
4.7	The FSM of <i>Memory Initializer</i> module. . . . .	33
4.8	Grid Lines Buffer Inner Architecture . . . . .	34
4.9	The <i>Reader's</i> functionality. . . . .	34
4.10	Datapath of the <i>CA Engine</i> . Case of $3 \times 3$ neighborhood. . . .	35
4.11	<i>Frame Extract's</i> FSM functionality. . . . .	37
4.12	The embedded color palettes in <i>Graphics Controller</i> . Figure designed by Kyparissas. . . . .	38
5.1	New top level View of the hardware architecture. Modules within dashes were modified. Rounded modules are the new ones. . . . .	42
5.2	Changes to <i>FRAME EXTRACT's</i> FSM along with the <i>SPEED CONTROLLER's</i> circuitry. . . . .	43
5.3	<i>CA Engine's</i> re-programmable structure for totalistic rules. Registers of neighborhood, multipliers and the adder tree was already developed. . . . .	45
5.4	<i>CA Engine's</i> final re-programmable structure for both totalistic and outer-totalistic rules. . . . .	46
5.5	The FSM diagram of <i>Deserializer Module</i> . The <b>MSB</b> of each byte concerns the continuation or stop bit. . . . .	49
5.6	Connectivity of <i>Deserializer</i> module in the design. . . . .	50
5.7	FSM of <i>Serializer</i> module. . . . .	51
5.8	Connectivity of <i>Serializer</i> in the design. . . . .	52
5.9	Implemented circuit on the device . . . . .	53
6.1	Flowchart of the back-end functionality. . . . .	56
6.2	The Greenberg-Hastings Model described in CDL. . . . .	58
6.3	An example of BRAM, given the reference example. . . . .	61
6.4	An example of BRAM, given the reference example. Case of cell size = 4bits . . . . .	61
7.1	The GUI environment . . . . .	66
7.2	Creating new project. . . . .	67
7.3	Configuring weights. (a) Widgets for shaping the neighborhood, (b) $29 \times 29$ lattice of entries. . . . .	69

7.4	(a) Setting up a $13 \times 13$ neighborhood, (b) Adding weights. . .	69
7.5	Adding 2 to every cell excluding zeros, in a von Neumann region. . . . .	70
7.6	Drop-down menu of neighborhood types. . . . .	70
7.7	(a) Moore, (b) von Neumann. . . . .	71
7.8	(a) Circular, (b) L2/Euclidean. . . . .	72
7.9	(a) Checkboard, (b) Checkerboard'. . . . .	72
7.10	(a) Hash, (b) Cross. . . . .	72
7.11	(a) Saltire, (b) Star. . . . .	73
7.12	Mirror Mode: (a) The one fourth of a von Neumann neighborhood, (b) Properly mirrored. . . . .	74
7.13	Mirror Mode with shifted center: (a) A small von Neumann within the second quadrant neighborhood, (b) Properly mirrored. Four von Neumann sub-neighborhoods were shaped. . .	74
7.14	Widgets for configuring the grid, inserting the transition rule and setting up the simulation. . . . .	75
7.15	An example of the format of a .config file. . . . .	75
8.1	The user's input inside the GUI. (a) <i>Artificial Physics'</i> configurations, (b) the neighborhood's window. . . . .	79
8.2	The evolution of <i>Artificial Physics</i> . (a) Initial State, (b) 500 generations, and (c), 60,000 generations along with a zoomed-in frame. . . . .	79
8.3	The evolution of <i>Game of Life</i> . (a) Initial State, (b) 500 generations, and (c), 15,000 generations along with a zoomed-in frame. . .	81
8.4	<i>Game of Life</i> in-GUI configuration. (a) Parameters/Transition Function (b) Neighborhood window. . . . .	81
8.5	(a) $15 \times 15$ and (b) $17 \times 17$ after the system has converged. . .	85
8.6	$21 \times 21$ neighborhood: (a) Initial State, (b) 19 and (c) 200 generations. . . . .	86
8.7	$25 \times 25$ neighborhood: (a) Initial State, (b) 22 and (c) 200 generations. . . . .	86
8.8	$29 \times 29$ neighborhood: (a) Initial State, (b) 40 and (c) 200 generations. . . . .	87
9.1	CA Engine's extended, re-programmable framework for supporting more complex models. A potential solution (blue color) attached to the already developed hardware (with black). . . .	92



# List of Abbreviations

<b>ALU</b>	<b>Arithmetic Logic Unit</b>
<b>ASCII</b>	<b>American Standard Code for Information Interchange</b>
<b>ASIC</b>	<b>Application Specific Integrated Circuits</b>
<b>BRAM</b>	<b>Block Random Access Memory</b>
<b>CA</b>	<b>Cellular Automata</b>
<b>CAD</b>	<b>Computer Aided Design</b>
<b>CPLD</b>	<b>Complex PLD</b>
<b>CUDA</b>	<b>Compute Unified Device Architecture</b>
<b>FPGA</b>	<b>Field Programmable Gate Array</b>
<b>GUI</b>	<b>Graphical User Interface</b>
<b>HIP</b>	<b>Hybrid Integration Protocol</b>
<b>IP</b>	<b>Integrated Protocol</b>
<b>LSB(s)</b>	<b>Least Significant Bit(s)</b>
<b>MPI</b>	<b>Message Passing Interface</b>
<b>MSB(s)</b>	<b>Most Significant Bit(s)</b>
<b>OpenMP</b>	<b>Open Multi-Processing</b>
<b>PLD</b>	<b>Programmable Logic Device</b>
<b>RAM</b>	<b>Random Access Memory</b>
<b>SPLD</b>	<b>Simple PLD</b>
<b>TCL</b>	<b>Tool Command Language</b>
<b>UART</b>	<b>Universal Asynchronous Receiver Transmitter</b>
<b>VHDL</b>	<b>VHSIC Hardware Description Language</b>
<b>VHSIC</b>	<b>Very High Speed Integrated Circuit</b>



*Heartily dedicated to the memory of Nikolaos  
Tampakis, the greatest mathematician and friend I  
ever had ...*



# Chapter 1

## Introduction

Cellular Automata (CA) are a Turing-complete mathematical structure invented by John von Neumann and Stanislaw Ulam in the 1940s; they comprise a discrete, deterministic, computational model [1, 2]. Over the last decades, the interest of many scientific communities ranging from physics to neuroscience has been stimulated to use the CA as a powerful tool, since complex physical phenomena can be simulated, observed, and verified by means of elegant definitions of a simple mathematical model.

### 1.1 Motivation

Cellular Automata's potential is limitless, seeing that they can model a broad range of phenomena which pertain to almost every scientific field. They can model physical processes, including molecular dynamics [3, 4], ecological theory [5] and artificial brains [6]. Moreover, the whole universe can fundamentally be described by digital information, hence, the evolution of the cosmos is possible to be simulated using this model [7]. John von Neumann created the first Cellular Automata, the Universal Constructor, in the 1940s. It is an abstract machine, that illustrates the conditions needed for self-replication [8, 9]. As a result of this creation, scientists delved into further research on the mathematical concept, and consequently, it was proven that the properties of universality and reversibility are fundamentally essential for numerous Cellular Automaton rules [10, 11], without reversibility having to apply to all CA models.

As a high school student, mathematics was my favorite subject by far. Nonetheless, the evolution of technology in combination with my interest in programming, led me to opt for studying Electrical and Computer Engineering (ECE). Given that mathematics is the dominant tool to advance science,

it became clear that ECE would pave the way to what I was looking for. By the time my supervisor proposed to me this Thesis subject, I responded positively without uncertainty. CA can fulfill someone's curiosity since it engages in myriads of scientific theories.

## 1.2 Thesis Contribution

In this Thesis, we have developed a CAD tool that aids non-FPGA conversant users to simulate CA rules in a convenient way. We use the term "simulate CA" as it is used by that community - all references to "CA simulations" refer to runs of the models on actual FPGA-based hardware. The hardware architecture has been designed by Nikolaos Kyparissas (presently a Ph.D. Student at the University of Manchester) in his TUC ECE M.Eng. Diploma Thesis [12], and published in [13, 14, 15]. Although Kyparissas' design as of the early 2020's is the largest ( $29 \times 29$  neighborhoods) and fastest (up to 100 HD frames per second) reported in literature, and it placed twice at the top tier in the AMD-Xilinx's Open Hardware Competition (in 2015 and 2018), hardware engineering knowledge is required for someone to exploit his machine. The key advantage of his design is that it is capable of simulating large sizes of neighborhoods, up to  $29 \times 29$ , at up to 100 frames per second and on various grids (cartesian, cylindrical or toroid) of  $1920 \times 1080$ .

In the present thesis a CAD tool has been developed to fully automate the initialization process of the machine, and provide users with a convenient simulation medium. Kyparissas' architecture, through this thesis, has been turned into a re-programmable framework with no need of the user to write VHDL code. An easy-to-use Cellular Description Language (CDL) has been developed to define the mathematical equations of models. Additionally, a GUI environment allows the user to enter the up-to-841 ( $29 \times 29$ ) weights. The combination of the above features allows for non-hardware-expert users to use the CA simulator.

## 1.3 Thesis Outline

This Thesis consists of 9 chapters. Following **Chapter 1**, i.e., this Introduction:

- **Chapter 2:** Contains the CA theoretical background on both Cellular Automata and FPGA technology.

- 
- **Chapter 3:** Presents relevant work on significant hardware architectures.
  - **Chapter 4:** Explains Kyparissas' design, which is essential to understand the changes and improvements made in the present thesis.
  - **Chapter 5:** Presents in detail the construction of the re-programmable framework.
  - **Chapter 6:** Analytically presents the back-end structure of the CAD tool.
  - **Chapter 7:** Has the GUI environment from a user's and from a system's perspective.
  - **Chapter 8:** Demonstrates applications, examples and experiments on the *Hodgepodge Machine* model, which were used both for the verification of the tools and architectures developed in this thesis, as well as to explore chimera states, a physical phenomenon of interest to physicists studying dynamical systems.
  - **Chapter 9:** Has conclusions and future work.



## Chapter 2

# Theoretical Background

This section provides the reader with the theoretical background regarding this Thesis. We introduce the discrete world of Cellular Automata which is important for the reader in order to understand both the hardware design and the CAD tool developed in this work. Secondly, we succinctly explain the basic principles of the FPGA technology in order to show why FPGA technology achieves such high performance vs. alternative technologies.

### 2.1 Cellular Automaton Model

Cellular Automaton is a discrete, dynamical system, characterized by the properties of locality and uniformity. In its general form, a CA theoretically consists of a multi-dimensional lattice (grid) of an infinite number of cells (see Figure 2.1), where each cell has a finite number of different states, and, a set of rules that determines the next state of a cell on the grid. Practically, however, the size of the lattice is finite, given that the resources of a computational machine are limited.

For this thesis, we focus on 2-D CA problems, and the process of calculating one is:

1. For every cell on the grid repeat the following steps.
2. Define a 2-D, relatively small region, called neighborhood, where the desired cell is placed at the center.
3. Multiply each cell in the neighborhood with the corresponding weight of a predetermined window of coefficients (weights).
4. Calculate the sum of all cells in the neighborhood, forming a dot product.
5. Decide on the next state of the central cell according to a transition rule.

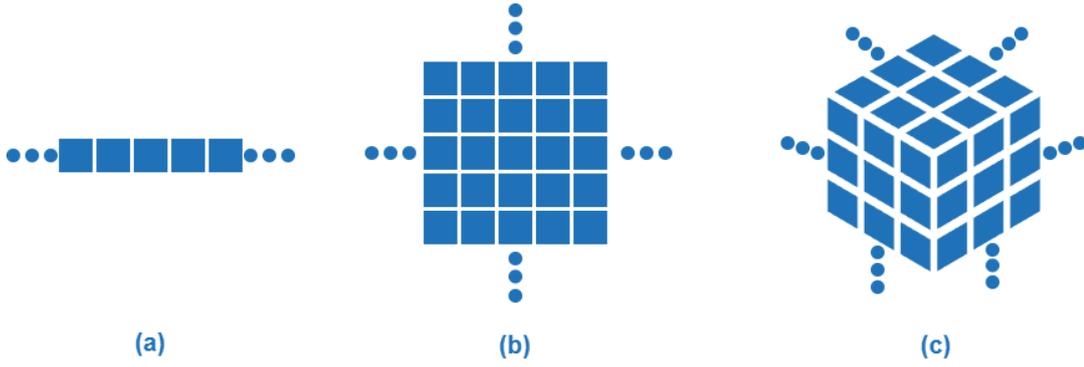


FIGURE 2.1: (a)-1D, (b)-2D, (c)-3D grid depiction.

After the grid is fully processed, the calculation of one generation has been completed. The simulation continues to the next generation and the user decides when it terminates. The neighborhood of a CA problem is shaped based on a radius and a type. The radius determines the region in which the total sum is being calculated, while the type defines the arrangement and the values of the weights. There are two mathematically defined neighborhood types, named Moore and von Neumann. In the rest of the present work we are going to refer to them as custom, as it is neither mathematically restricted to only exploit the aforementioned types, nor architecturally restricted by the CA accelerator (of Kyparissas and of this thesis).

The Moore neighborhood is a square-shaped region around the central cell that covers an  $N \times N$  area, where  $N$  is an odd number, whereas a von Neumann neighborhood is (in its simplest form) the central cell in question plus its north, south, east and west neighbors.

All types of neighborhoods can be either weighted or not, as shown in Figure 2.2. Finally, it is important to note that the length of each dimension of the neighborhood should be restricted to odd values, otherwise, a unique central cell can not be determined.

To mathematically define a Moore or von Neumann region, let  $(x, y) \in \mathbb{Z}$  be the coordinates on  $XY$  axis, where the pair  $(x_0, y_0)$  corresponds to the central cell, then, the set of cells that belongs in the Moore ( $N_{(x_0, y_0)}^M$ ) or von Neumann ( $N_{(x_0, y_0)}^V$ ) neighborhood of radius  $r$  is defined by the following mathematical expressions [16, 17]:

- $N_{(x_0, y_0)}^M = \{(x, y) : |x - x_0| \leq r, |y - y_0| \leq r\}$
- $N_{(x_0, y_0)}^V = \{(x, y) : |x - x_0| + |y - y_0| \leq r\}$

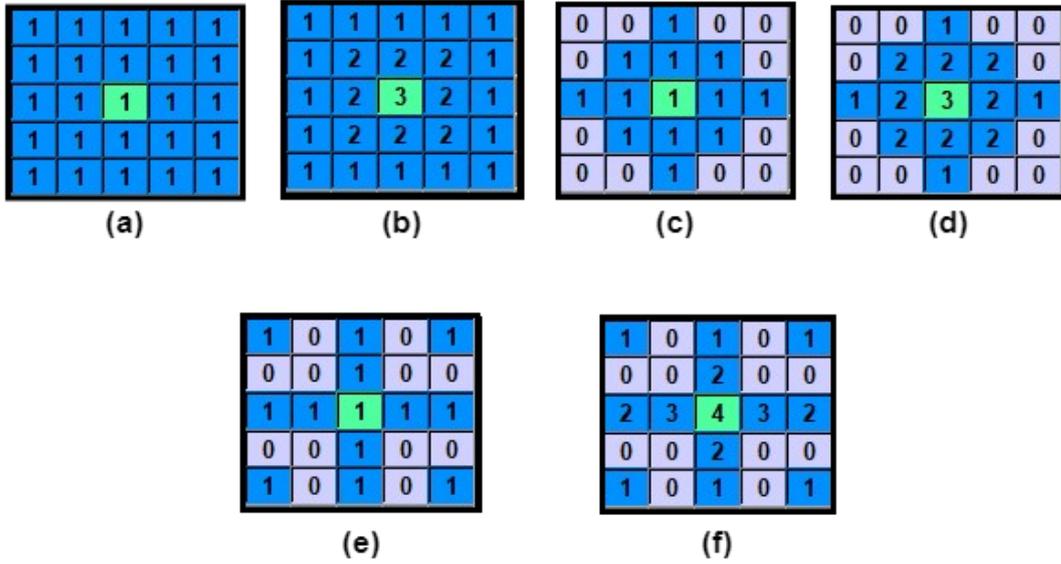


FIGURE 2.2: Neighborhood types:  
 (a) Non-weighted Moore, (b) Weighted Moore,  
 (c) Non-weighted von Neumann, (d) Weighted von Neumann,  
 (e) Non-weighted Custom, (f) Weighted Custom.

### 2.1.1 Cellular Automata Classes

The CA problems are mainly categorized into two classes: totalistic and outer totalistic. In a totalistic CA, the next state of a cell only depends on the total sum of the neighbors of the central cell, while an outer totalistic rule depends on both the total sum and the current state of the central cell [18]. Therefore, an outer totalistic CA rule is a superset of a totalistic one. To mathematically demonstrate a CA problem, let a finite set of possible cell-states be:  $S = \{s_0, s_1, \dots, s_n\} \in \mathbb{Z}^{0+}$ . The total sum of each cell using predefined weights is given by equation (1), while the transition rule is defined by equations (2) or (3).

$$\text{Sum} = \sum_{x=-r}^r \sum_{y=-r}^r w(x, y) * c_t(x, y) \quad (1)$$

$$c_{t+1}(x, y) = \begin{cases} s_0 & , \text{ if } a_0 \leq \text{Sum} < b_0 \\ s_1 & , \text{ if } a_1 \leq \text{Sum} < b_1 \\ \vdots & \\ s_n & , \text{ if } a_n \leq \text{Sum} \leq b_n \end{cases} \quad (2)$$

$$c_{t+1}(x, y) = \begin{cases} s_0 & , \text{ if } a_0 \leq \text{Sum} < b_0 \text{ and } c_t(x_0, y_0) \in [t_0, t_1) \\ s_1 & , \text{ if } a_1 \leq \text{Sum} < b_1 \text{ and } c_t(x_0, y_0) \in [t_1, t_2) \\ \vdots & \\ s_n & , \text{ if } a_n \leq \text{Sum} \leq b_n \text{ and } c_t(x_0, y_0) \in [t_n, t_{n+1}] \end{cases} \quad (3)$$

The transition rules (2) and (3) correspond to totalistic and outer totalistic CA problems respectively. This can be verified by the fact that the central cell doesn't affect the result of the former transition rule, while it does so in the latter. It should be mentioned that the above equations display a generalized picture of how a CA problem is defined, and of course, there are numerous ways to formulate a transition rule. The constant variables ( $a$ ,  $b$ ,  $t$ ) and the values of weights are mainly determined by either the mathematics that describe the model (i.e. differential equations, polynomial functions, etc.), or, after running multiple experiments and patiently observing the behavior of the rule.

### 2.1.2 The Game of Life

The Game of Life is considered to be one of the most popular two-dimensional CA problems. It was invented by the mathematician John Conway in 1970 [19]. It merely consists of two states ('0' or '1', 'dead' or 'alive') and simple four transition rules (see also Figure 2.3):

1. If a cell is alive and it has one or no neighbors, it dies because of solitude.
2. If an alive cell has four or more neighbors, it dies due to overpopulation.
3. If an alive cell has two or three neighbors, it survives.
4. If a dead cell has exactly three neighbors, it is resurrected.

The above transition rule can be represented similarly to the formulas described in the previous section:

$$c_{t+1}(x, y) = \begin{cases} 1 & , \text{ if } 2 \leq \text{Sum} \leq 3 \text{ and } c_t(x_0, y_0) = 1, \\ 1 & , \text{ if } \text{Sum} = 3 \text{ and } c_t(x_0, y_0) = 0, \\ 0 & , \text{ otherwise} \end{cases} \quad (3)$$

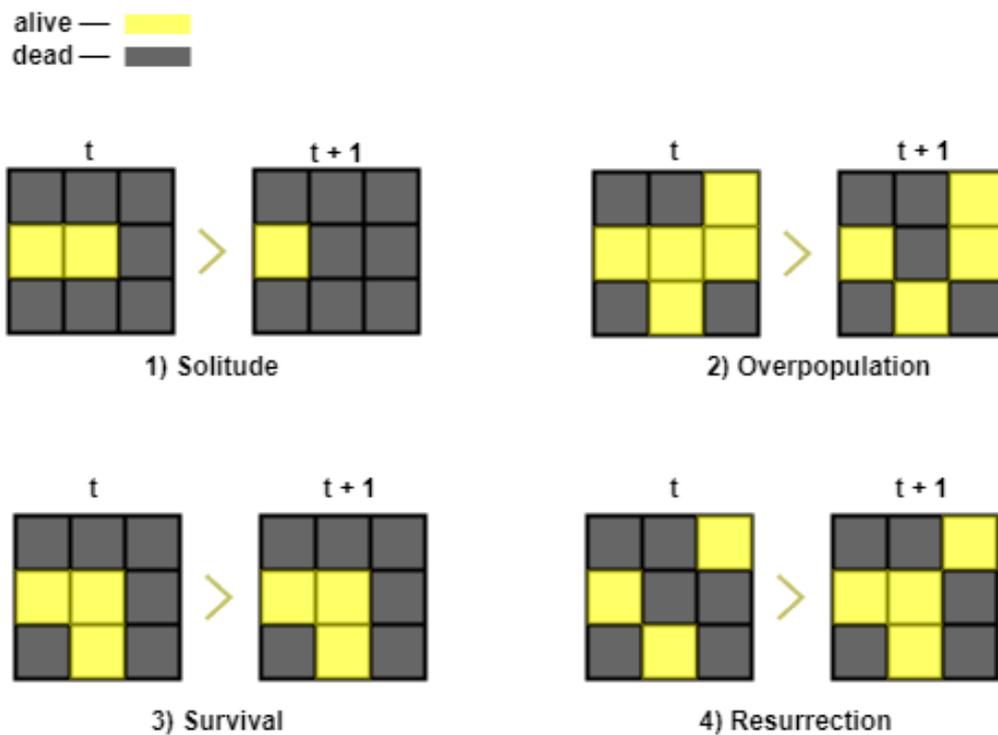


FIGURE 2.3: Game of Life transition rules.

These rules define the next state ('0' or '1') of a cell on the grid based on both the total sum of neighbors and the state of the central cell. Consequently, the Game of Life is an outer totalistic CA problem using a  $3 \times 3$ , non-weighted, Moore neighborhood. Despite its simplicity, outstanding patterns beyond human imagination can be observed during its simulation. To name but a few: stable patterns that remain unchanged through time, repeated patterns that interchange between a finite number of states (think of it like a closed loop), cells traveling through the grid while infinitely maintaining their shape, oscillators, "spaceships", "glider guns", and even logic gates, counters and Finite State Machines.

Theoretically speaking, a fully functional computer can be built inside this Cellular Automaton by finding a proper initial state for the simulation, and arguably, the Game of Life is a Turing-complete model [20, 21]. This approach of computer design, of course, is totally inefficient and unacceptably slow. Notwithstanding, all these fascinating patterns have stimulated the interest of both aficionados and researchers worldwide, because several phenomena can be recounted in the Game of Life. Such phenomena pertain to numerous scientific fields such as mathematics, computer science, biology, economy, and more.

## 2.2 FPGA Technology

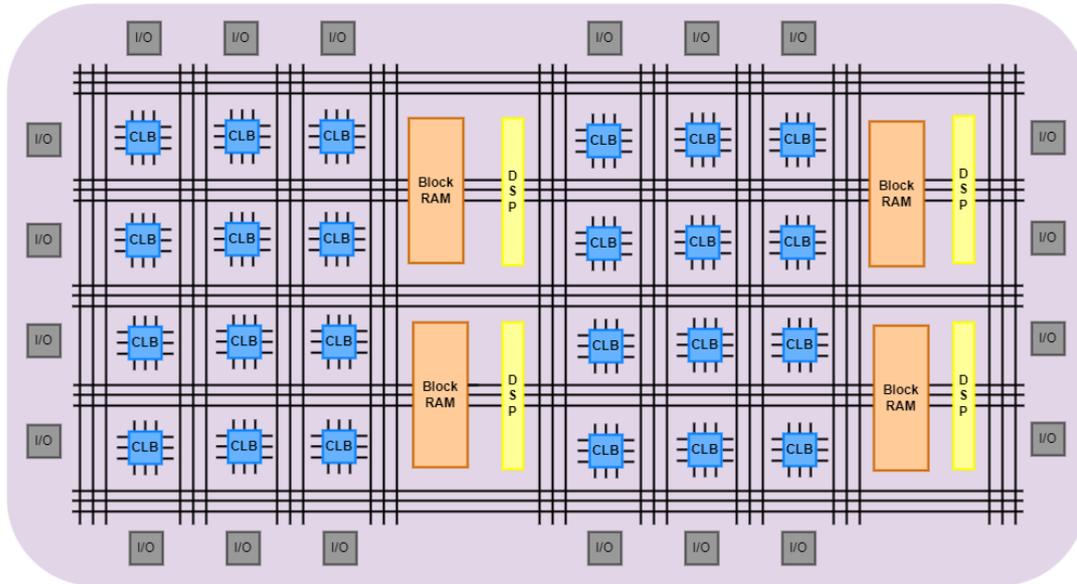


FIGURE 2.4: FPGA's Hardware Architecture. The basic components.

The term FPGA stands for **Field Programmable Gate Array** and it corresponds to a semiconductor, programmable, logic circuit. The first device for commercial purposes was first introduced to the public by Xilinx Corp. in 1985. Nowadays, Xilinx has merged with AMD under the brand of AMD-Xilinx Corp. Today's FPGAs are capable of calculating tasks, that demand very large amounts of resources (in the billions of gate equivalents), very fast (in the hundreds of MHz clock speed) and efficiently energy-wise. Thus, they are widely used in industry. That is to say, significant real-life applications - in domains such as physics, biology, astronomy, more and more - strongly rely on the performance of FPGAs, their main advantage being reprogrammability and the execution of algorithms directly on the hardware fabric (vs. running on a predefined processor architecture). This happens because FPGAs are mainly composed of fully programmable logic blocks placed in an array-like arrangement, plus individually addressable local memories, digital signal processors (DSP), local clock synchronizers, memory controllers and I/O interfaces, etc. An FPGA board is generally structured by four basic components (see figure 2.4):

1. Programmable Interconnections.
2. Configurable Logic Blocks (CLBs).
3. Input and Output (I/O) Resources.
4. Digital Signal Processing (DSP) Units.

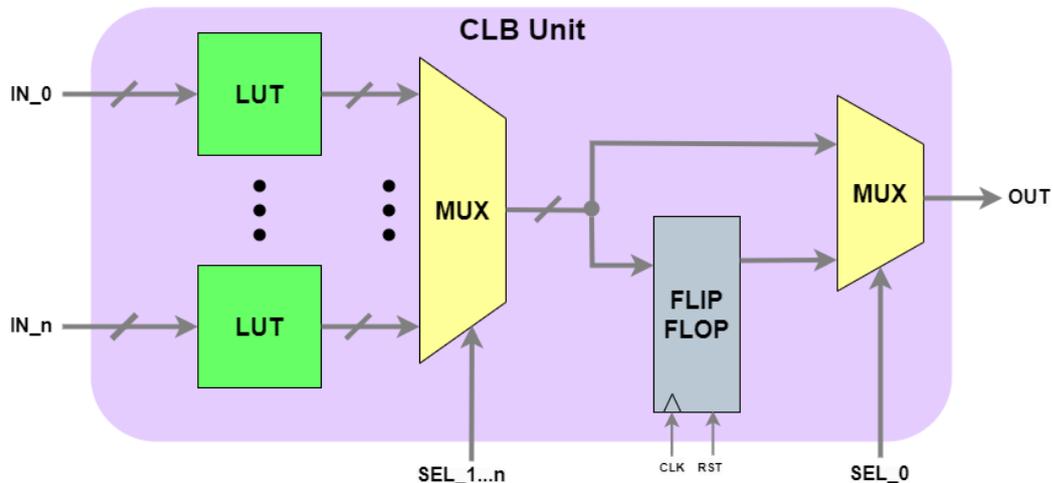


FIGURE 2.5: Basic structure of Control Logic Blocks

The CLBs are connected to each other with programmable interconnections according to the user's design. A CLB comprises of three simple logic elements: a Look-Up-Table (LUT), a Flip-Flop and Multiplexers, as shown in Figure 2.5. In the LUT, a list of results for every possible input is stored so as to be ready when needed. A Flip-flop connected with a Multiplexer provided CLB with the flexibility to be either clocked, synchronous or combinational logic circuits. The CLB units can be used to form logic components such as adders, multipliers, dividers, counters, and registers, while a combination of CLBs is capable of calculating complex tasks, including Central Processing Units ("soft core" CPUs).

The I/O resources on an FPGA board are physical structures that allow FPGAs to communicate with the outside world. These resources include interfaces for protocols such as VGA, Ethernet, HDMI, etc., and, buttons, switches and LEDs. Consequently, an FPGA is capable of exchanging information with a wide range of devices and users, as long as the proper hardware has been developed in it. Regarding the DSP units, these generally consist of adders and multipliers in a MAC (multiply-accumulate) configuration. Thus, DSP units are suitable for applications such as: image or video processing, noise filtering, sonar or radar signal processing, wireless communication, and more.

FPGA technology offers a very high amount of internal aggregate bandwidth, in the order of TBytes/sec. The internal FPGA structure allows for a very high degree of parallelism, which can be exploited in many classes of applications. As a result, FPGAs are very suitable to compute complicated

mathematical models, and nowadays are widely utilized in many areas, including Artificial Intelligence and Convolutional Neural Networks (CNN), an application of great interest to the scientific community and of great social impact.

### 2.2.1 FPGA vs. Other Devices

FPGAs are not the only logic programmable devices (PLDs) that exist. To name but a few, there are SPLDs, CPLDs, Simple- and Complex-PLDs, correspondingly. The SPLDs and CPLDs have substantially lower capacity vs. FPGAs and they are not capable of computing complicated tasks. Notwithstanding, there is practical use of these devices in applications such as encoding/decoding, data display, and for example, they could be used alongside an FPGA on a Printed Circuit Board (PCB) to execute system configurations and allow for the FPGA resources to be used on the computationally heavy work.

Aside of PLDs, CPU's as well come in variants, beyond the processors which are used in general-purpose computers. To illustrate, a Microcontroller is a "hybrid" device made up of both digital and analog components. Its digital parts mainly consist of a pipeline that operates like a CPU, Random Access Memory (RAM), Read-Only Memory (ROM) and I/O Ports for external communication. What truly distinguishes microcontrollers from conventional CPUs are the onboard analog resources, which allow for it to operate standalone. These include: Oscillators to provide clock to the circuit, Pulse Width Modulators (PWM), Analog-to-Digital/Digital-to-Analog Converters, watchdog timers, etc. All of these components allow for great versatility of Microcontrollers, rendering them suitable for numerous applications and they are embedded in other systems ranging from home devices such as printers, smartphones and microwaves to airplanes, satellites, and space-ships.

Another category of high-performance integrated circuits are the Graphics Processing Units (GPUs), which are highly pipelined processors with high performance, especially on floating point operations.

Application Specific Integrated Circuits (ASICs) constitute the best competitor of FPGAs in terms of computing power and processing speed. In contrast with FPGAs, ASICs can not be re-programmed. On the other hand, FPGA's hardware can be re-targeted over and over again towards specific

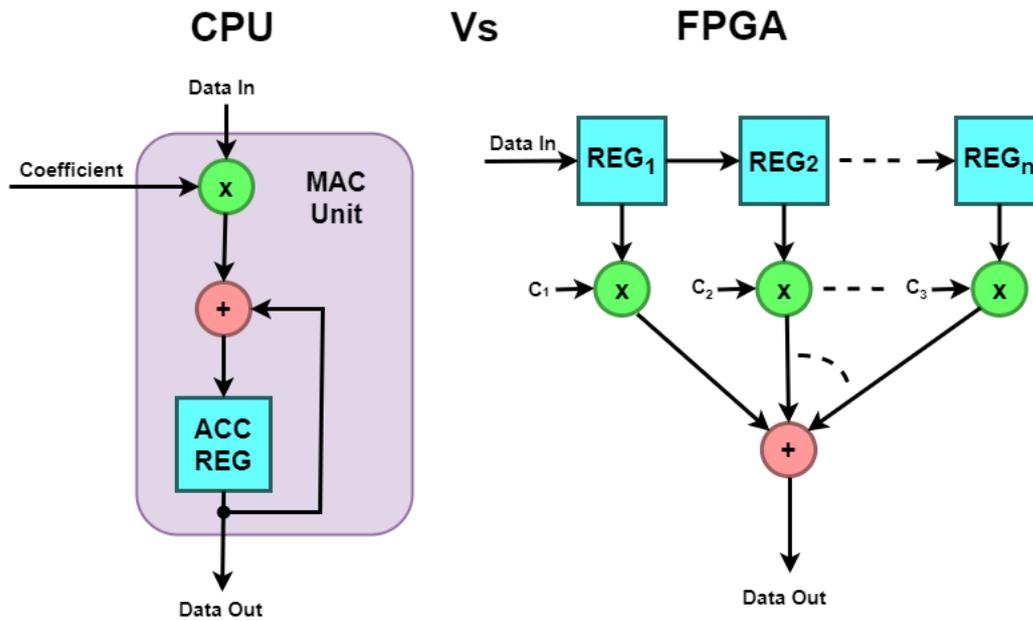


FIGURE 2.6: n-MAC operation. CPU vs FPGA

computations. Therefore, for a broad class of problems an FPGA will implement a targeted task faster than a CPU.

It should be noted that the clock frequency of a CPU or GPU is roughly one order of magnitude higher than that of a typical FPGA design. Hence, the speedup of FPGA-based designs vs. CPUs and GPUS comes from parallelism, pipelining, and the very high internal bandwidth of FPGAs.



## Chapter 3

# Related Word and Motivation

In this section, we outline significant special-purpose machines to accelerate CA problems. We are not aware of any CAD tools similar to the framework presented in Kyparissas' thesis or in the present one. Even so, the accelerators presented here were available to users.

At the last part of this chapter, the approach of the Thesis is presented in order to show the motivation for the newly-developed CAD tool, how it differs from existing methods methods, and how it complements the hardware accelerator.

### 3.1 CAM Architecture (1984~2000)

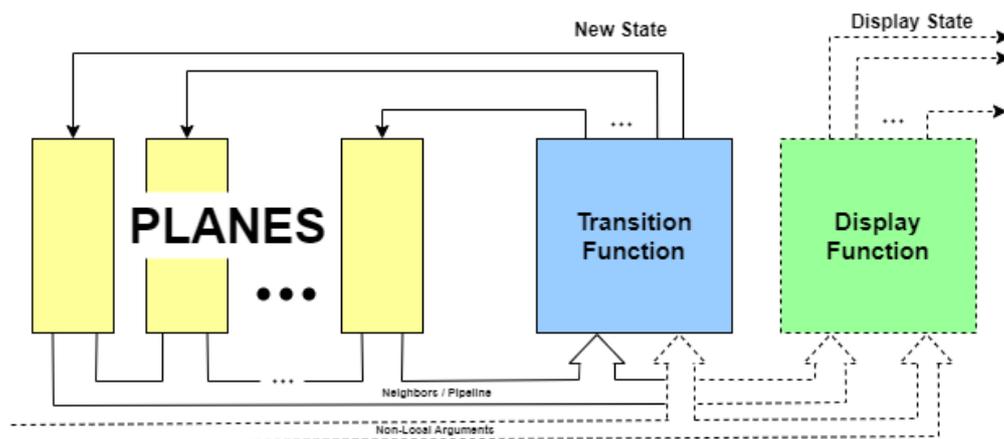


FIGURE 3.1: CAM's hardware architecture. Basic computational loop with solid lines. Source [22]

Tomasso Toffoli's and Normam Margolus' Cellular Automata Machine (CAM) was the first special-purpose computer to simulate CA. Tomasso Toffoli, also known for inventing the universal reversible Toffoli gate, developed

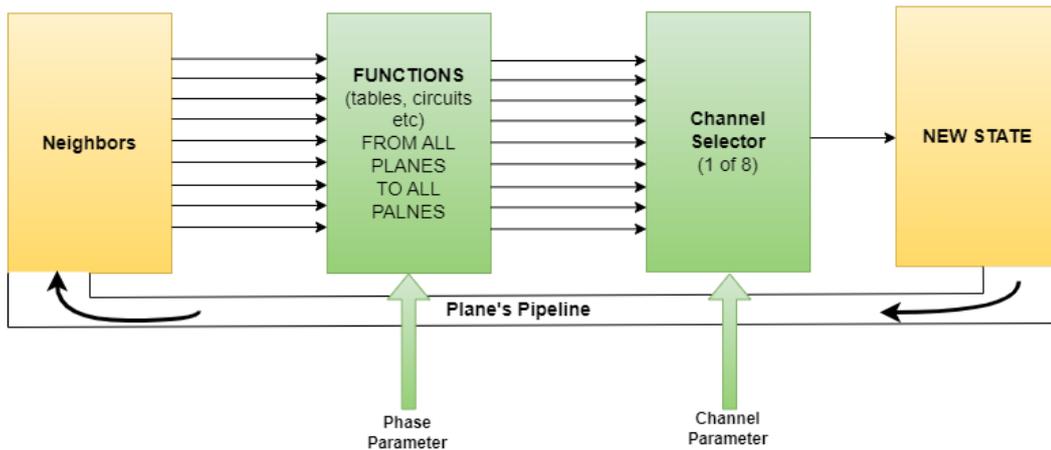


FIGURE 3.2: The form of the "gap" that should be filled by the transition function. Phase and channel parameters are non-local arguments. (see figure 3.1). Source [22]

a CA accelerator based on TTL logic, named CAM. The first version published in 1984 [22].

CAM's architecture was fully pipelined and was mainly consisted of: the memory *Planes*, the *Transition Function* and the *Display Function* (see figure 3.1). One memory *Plane* stored a  $256 \times 256$  lattice of 1-bit depth, where up to 8-bit planes were supported. The memory *Plane* delivered a  $3 \times 3$  neighborhood to the *Transition Function* per clock cycle. The *Transition Function* was entirely made up of SRAM Look-Up Tables and the simulation was evolved at 60FPS.

The *Non-Local Arguments*, as figure 3.2 depicts, are user-defined components, where their circuitry depends on the transition rule. Additionally, the CAM box consisted of a 10-slot card cage and the minimum setup required for simulations, encompasses the following components:

1. **One Interface Card:** Providing user/monitor ports, system registers, etc.
2. **One Scanner Card:** Containing timing and control logic.
3. **One Plane Card:** Including two planes.
4. **One Table Card:** Involving two separately addressable  $4 \times 1024$ -bit SRAM-LUTs.

The user could plug and unplug those cards in different combinations, adding extra features or removing the unnecessary ones. Either factory-made or custom-made cards were supported, offering: additional memory planes, more cell states, random number generators, multiple LUTs, and so on. The

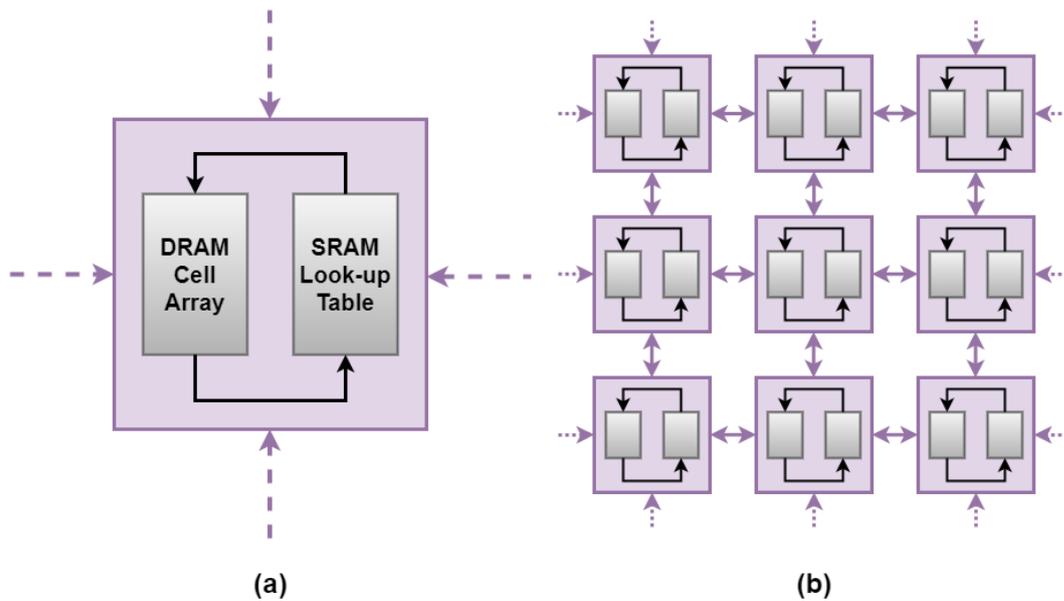


FIGURE 3.3: CAM's-8 hardware architecture. (a) A single module. (b) Each module is connected to the nearest ones. Source [24]

CAM box was driven under a low-level software tool for defining CA models, designed by Norman Margolus. Moreover, the CAM's hardware could be configured for other applications regarding parallel computing and image processing, apart from CA models.

Norman Margolus was a Ph.D. student under Toffoli's supervision and both together collaborated to further develop the CAM. They published a book about the CAM-6 version in 1986 which demonstrates the capabilities of the machine on numerous applications [23]. As a new feature in CAM-6 version, the memory planes could literally be "glued" edge-to-edge, in consequence, arbitrary large lattices, multiple dimensions and different types of grids were supported. A technique called "scooping" was utilized, so, the whole grid was stored in the host computer's RAM and the  $256 \times 256$  area was processed in the pipeline. Finally, a high level software was driven CAM-6 written in Forth and the machine offered a plug-and-play experience.

Henceforth, Margolus continued developing the machine and he introduced the CAM-8 version in 1993 [24]. This design consisted of multiple mini-processors (Figure 3.3). Each processor contained one DRAM for storing a part of a grid, plus, a SRAM LUT for deciding the next state. The computations relied on shift-based operations across sectors. Thus, CAM-8 could support arbitrary large neighborhood windows and states per cell, as many as the available mini-processors. It was capable of producing 190 generations

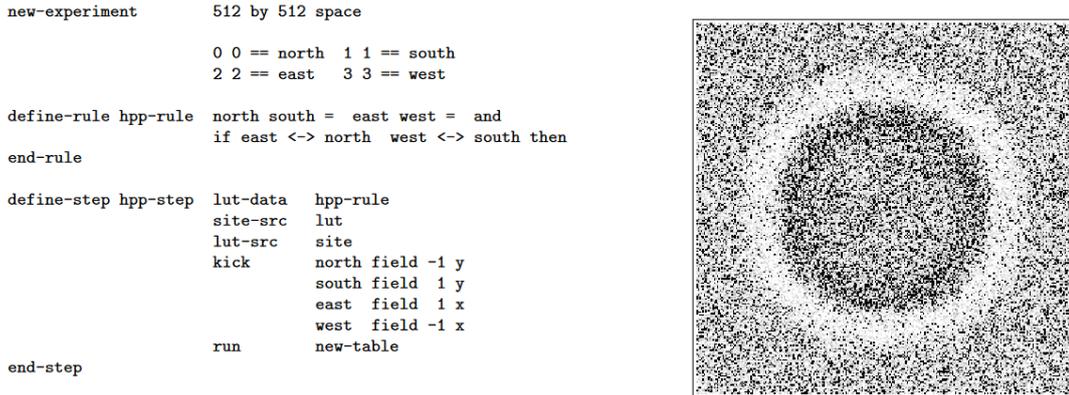


FIGURE 3.4: CAM's-8 programming environment alongside a sample of a sound pulse. (Source: [24])

per second, although, less speedup were achieved as the radius grew.

For the CAM-8 version, a new software was developed on top of the preceding version as a higher layer, offering backwards compatibly. Still, the models was determined through a programming environment (see figure 3.4) in assembly language, which was specifically targeted to the CAM's hardware.

Based on his ideas and experiences throughout CAM's development, Margolus exploited FPGA technology to put them into practice later on. Utilizing fast and high-capacity DRAMs in combination with the internal bandwidth of FPGAs, he proposed hardware architectures capable of performing systolic computations, DSP-like applications and spatial-lattice computations in general, apart from CA rules [25, 26].

## 3.2 CEPPRA Architecture (1994~2000)

CEPPRA stands for **C**ellular **P**rocessing **A**rchitecture and it was an FPGA-based architecture developed at the Technical University of Darmstadt, Germany. CEPPRA refers to a family of streaming architectures where several prototypes were released. Instead of using LUTs - as CAM did - so as to compute the transition rule of a Cellular Automaton, CEPPRA utilized pipelined arithmetic logic, capable of deciding the next state of a cell at one clock cycle. In this way, CEPPRA accomplished to simulate even more complex and probabilistic CA problems, whereas CAM had to divide the rule into cascaded LUTs.

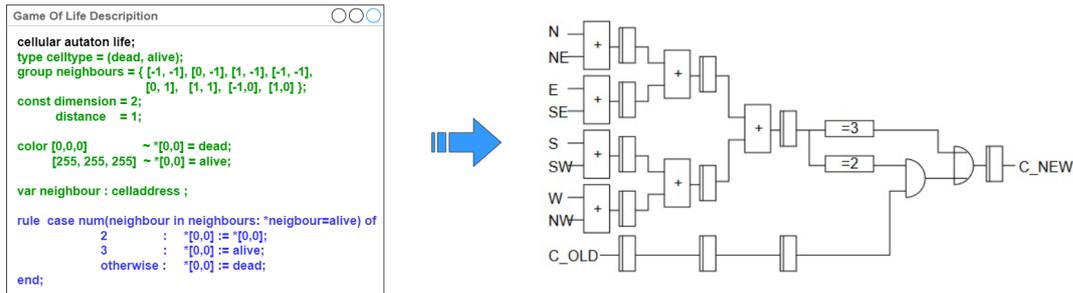


FIGURE 3.5: CEPRA's-1X environment alongside the derivative hardware. Source [28]

CEPRA-8L was the first version of the family and it was published in 1994 [27]. Eight, application-targeted, configurable FPGAs were working in unison, each one having independent access to the 8 nearest neighbors. Therefore, all of the cells of the neighborhood could be simultaneously updated to their next state. CEPRA-8L supported a  $3 \times 3$  neighborhood size on a  $512 \times 512$  lattice of 8-bit cells or 256 states, displaying 22 FPS in real-time.

Three years later, the next version CEPRA-1X was completed and published in 1998, while a Cellular Description Language (CDL) was also developed to aid user's simulations [28, 29]. CEPRA-1X supported both 2D and 3D CA problems, still,  $3 \times 3$  neighborhoods were supported. The evolution could be displayed on a  $1024 \times 1024$  grid of 16-bit cells, achieving a speedup of  $50 \times$  in comparison with a typical CPU of that era.

In this timeframe, the Cellular Automaton Description Language (CDL) was firstly introduced. An example of the description of the Game of Life is represented in figure 3.5 alongside the respective logic circuit, as the authors demonstrated it. The addition of CDL offered real advantages in terms of usability vs. the previous versions.

### 3.3 SPACE Architecture (1996)

Scalable Parallel Architecture for Concurrency Experiments was an FPGA-based architecture, developed by Paul Shaw, Paul Cockshott and Peter Barrie at Strathclyde University's Computer Science department [30]. The SPACE was designed to specifically target HPP models<sup>1</sup> and one SPACE board consisted of  $4 \times 4$ , array-like-arranged FPGAs, a Transputer<sup>2</sup> microprocessor

<sup>1</sup>The Hardy-Pomeau-Pazzis (HPP) model is a fundamental Lattice Gas Cellular Automaton that simulates the physics of fluids or gases.

<sup>2</sup>Transputer was the first general-purpose microprocessor for parallel computing developed in the 80s.

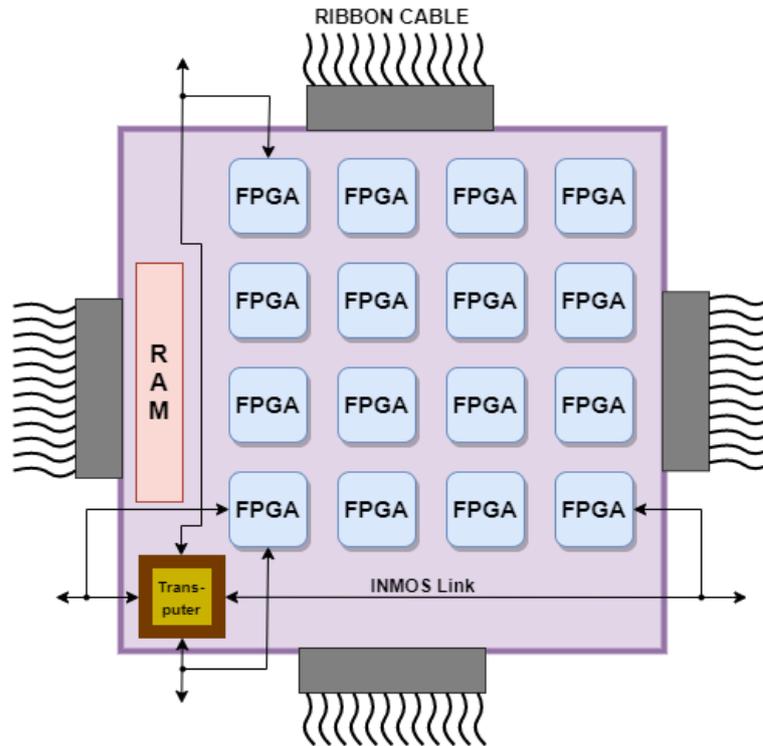


FIGURE 3.6: One SPACE board. Source [30]

and INMOS Links for converting serial to parallel and vice versa (Figure 3.6). One board could hold a  $9 \times 30$  lattice gas automaton and the scalability feature allowed to linearly expand the grid by connecting multiple boards side-by-side via ribbon cables.

The FPGAs were designed with inter-programmable, collision, logic blocks (Figure 3.7). While Look-Up tables are capable of merely updating one or a few cells simultaneously, with this approach, each Processing Element (PE) could hold every cell of the lattice separately, categorizing them into horizontal and vertical sub-populations without mixing them. The data streams or cells are being shifted from west to east, from north to south and vice versa, and the PEs decide if the cell continues its travel to the next clock cycle. Ultimately, one SPACE module with memory boards could approximately achieve a speedup of  $10\times$  over two, CAM-8 modules on simulating a  $9 \times 30$  HPP model.

The simulation was driven under a program written in C. The source code contained libraries of placement and wiring functions. The former allowed to manipulate the basic hardware components of the design, while the latter produced the netlist files which describe the hardware. Furthermore, an

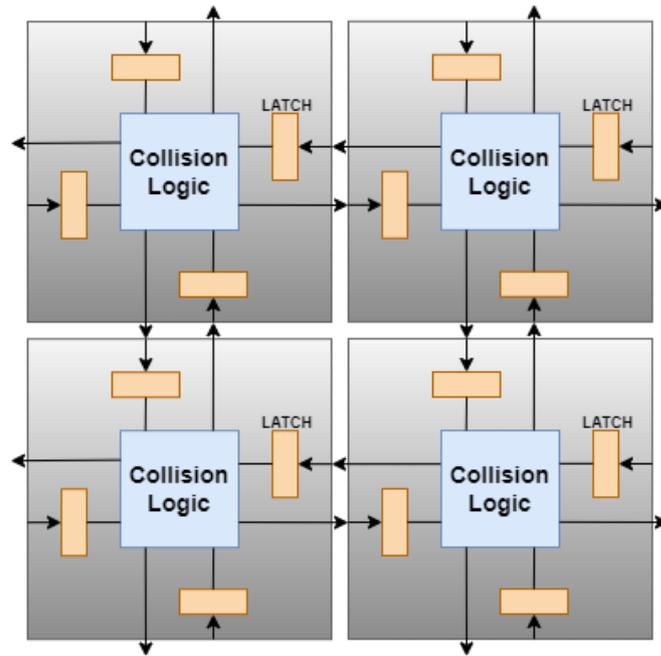


FIGURE 3.7: Top level view of a lattice gas automaton. Source [30]

optimized software tool was developed for placing and routing the logic elements in the circuit.

### 3.4 Kobori, Maruyama and Hoshino (2001)

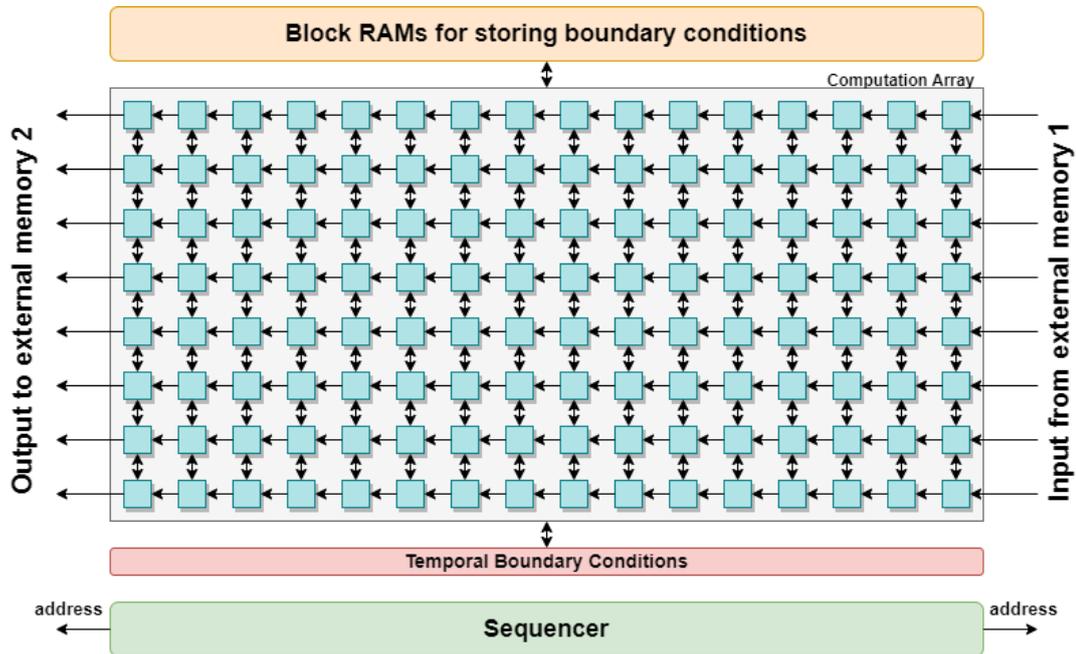


FIGURE 3.8: Overview of hardware architecture. Case of  $8 \times 16$  ( $k \times n$ ) PEs.  $k$  and  $n$  depend on the rule. Source [31].

Tomoyoshi Kobori, Tsutomu Maruyama and Tsutomu Hoshino from the University of Tsukuba in Japan proposed an FPGA-based architecture for accelerating Cellular Automata [31]. This design consisted of  $k \times n$  Processing Elements ( $k, n \in N$ ), where the data was read from and stored to two, separate, external memories (figure 3.8).

This hardware structure allows reading  $k$  cells at the same time, while  $k \times n$  cells are processed simultaneously. The  $k$  cells need  $n$  clock cycles to traverse the whole pipeline. Given that the pipeline is filled with data, the generations are growing with a time-step of  $+n$  every clock cycle. The key issue was that, due to the limited memory bandwidth, the cells at the circumference of the lattice couldn't provide the information needed to properly update the state of the central cell. To resolve this, the designers exploited the FPGA's distributed RAMs of very high bandwidth to temporarily hold such cells and handle boundary conditions.

Overall, the system consisted of one FPGA and an PCI board, the simulation was driven from the host computer (figure 3.9). The reading and writing

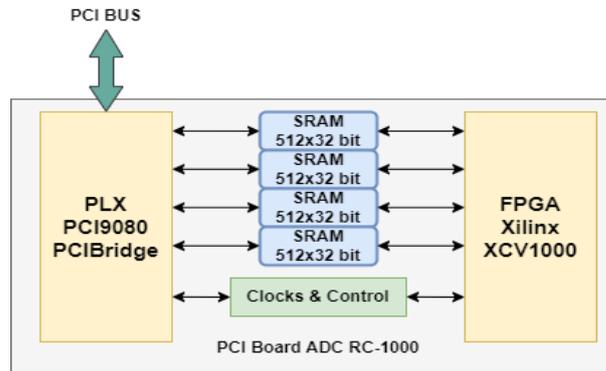


FIGURE 3.9: System overview. Source [31].

data was controlled by a DMA and traveled through the PCI bus. The Cellular Automaton rule could be described in a programming environment, which was a simplified extension of C code.

This design could achieve outstanding performances, capable of producing 400 generations per second on a  $2048 \times 1024$  FHP Lattice Gas Cellular Automaton. It accelerated the model by offering a  $155\times$  speedup over the Intel Pentium-III running at 700MHz. Notwithstanding, it was a pseudo-real-time visualization, since, most of the calculated generations were never actually stored back in the external memory.

### 3.5 Phepls' and Islam's Framework (2023)

Chase Phelps and Tanzima Islam developed A Cellular Automata Compiler (ACAC) framework in 2023, capable of simulating CA models on heterogeneous platforms [32]. As far as we are concerned, it is the latest attempt regarding CA simulators, nonetheless, it is not an FPGA-based architecture.

Their software-level framework optimizes CA computations to run on CPUs and GPUs. They utilized the respective tool-kits and APIs that the manufacturers provide, such as: CUDA framework, HIP and OpenMP. Multiple units collaborate in a cluster, where they interchange messages with MPI protocol. Several parallelization strategies were proposed by utilizing: both shared and distributed memory, double buffering and packing cell states up into groups of eight.

The ACAC framework supports up to  $29 \times 29$  neighborhood areas and 8-bit cell sizes, while the model can evolve on either 2D or 3D grids. There are four grid sizes available ranging from 2.36M to 236M cells (i.e.,  $1920 \times$

1080,  $5120 \times 2880$ ,  $10240 \times 5760$ , and  $20480 \times 11520$ ). The framework on the following platforms:

- **PHI:** The Stampede2 cluster. Contains 68 Intel Xeon Phi 7250 @ 1.4 GHz CPUs.
- **GTX:** The Maverick2 cluster. Includes 68 Intel Xeon E5-2629 @ 2.1GHz CPUs, and 4 NVIDIA 1080-TI GPUs.
- **M150:** The Penguin On-Demand (POD) cluster. Embodies 48 AMD EPYC CPU of 7001 series and 8 AMD Radeon Instinct M150 GPUs, Vega20 architecture.

They managed to achieve  $16\times$  and  $13\times$  speed up on Artificial Physics (AP) and Greenberg-Hastings (GH) models respectively, over our FPGA architecture. Nevertheless, their framework supports a limited amount of deterministic CA applications, while Kyparissas' architecture is considered a general-purpose CA machine. Furthermore, our architecture has been developing on a medium-sized FPGA, released in 2015. With a state-of-the-art FPGA board, the performance of our architecture can further increase. Last and foremost, the AP and GH models use  $21 \times 21$  and  $29 \times 29$  neighborhood sizes correspondingly. Thus, the larger the neighborhood grows, the less speedup the ACAC framework achieves, which is not the case in our FPGA implementation.

### 3.6 Other Significant Approaches.

All of the aforementioned architectures added their bit to evolve Cellular Automata accelerators. Over the last decades, dozens of designs and ideas have also contributed to the field. So, in the present section are succinctly demonstrate the ones that decided to be noteworthy.

In 1991, Bouanna et al. exploited the ArMen Machine, designed for parallel computation at Laboratoire d'Informatique de Brest, in order to accelerate Cellular Automata simulations [33]. Interprogrammable FGPA on a linear ring arrangement, with an internal 32-bits data path, collaborated alongside three parallel layers: communication system, sequential processor and synthesize operator arrays. The user specified the rules via a C program and a CCEL compiler was developed to translate it into hardware.

In 2001, G. Cappuccino and G. Cocorulla published a standalone, FPGA-base computing machine, named CAREM [34]. It was a scalable architecture,

meaning that as many as possible FPGAs could be added and increase performance, where each FPGA consisted of PEs working in parallel. However, it is not mentioned how the system was configured by the user, so, it is assumed that the respective HDL should be re-written manually.

Between 2000 and 2010, Murtaza, Hoekstra and Sloot from the University of Amsterdam developed a series of FPGA-based architectures for accelerating CA [35, 36, 37, 38]. Based on whether a specific CA rule needs to prioritize memory or computational resources, they proposed a variety of topologies, sizes and types of PEs working in parallel. Floating point arithmetic was also supported and validated by running Boltzmann fluids on FPGA clusters. To configure the desired problem, a software kit that supported a C or C++ program was running on the host computer, including drivers, headers and library files.

Last but not least, A.C. Lima and J.C.Ferreira from Porto University published a re-configurable hardware circuit for simulating CA[39]. Likewise, the FPGAs were structured with array-like arranged PEs, while, merely  $3 \times 3$  neighborhood sizes on an up to  $72 \times 72$  lattice were available. Contrary to already-discussed software tools, they developed a friendly GUI written in Java, capable of executing the design flow and parameterizing the CA. Nevertheless, the transition rule should be textually determined with Verilog HDL.

### 3.7 Thesis Approach and Motivation

The aforementioned approaches used small to medium-sized neighborhood windows, while the supported transition rules were limited. Furthermore, the majority of CAD tools developed, drove those machines by means of a low or high level programming language. This feature tho, limits the potential end-users given that they required specialized technical knowledge.

On the other hand, Kyparissas' design utilizes large  $29 \times 29$  neighborhood windows on a FHD lattice @60FPS and it is a general-purpose machine of CA models. Although the initialization procedure has to be performed manually, the generic framework must be reconfigured for every different model, and thus writing the transition rules in VHDL is unavoidable.

To address the above limitations, the CAD tool developed in the present thesis was developed. A re-programmable structure of the initial framework

has been constructed, so that circuitry of the hardware can be re-configured. The CAD tool fully automates the initialization of the machine and the extraction of snapshots. An easy-to-use Cellular Automata Description Language (CDL) has also developed to define transition rules. A CDL compiler translates user's input into the appropriate binary data that the hardware expects. Finally, a GUI environment enhances user's experience, especially when determining  $29 \times 29 = 841$  coefficients.

## Chapter 4

# The Baseline Hardware Architecture

Nikolaos Kyparissas has designed in his M. Eng. thesis a sophisticated FPGA-based architecture to simulate CA problems in real-time [12]. This high-performance system allows for large neighborhoods (up to  $29 \times 29$ ), on HD grids ( $1920 \times 1080$ ), with 4-bit weights and 8-bit states, running in real-time at 60 updates per second, and with an incorporated graphics controller for video output of the state (the hardware can process up to 100 fps but it was synchronized with the display controller). The hardware design has been described in detail, not only in Kyparissas' thesis, but also in three published papers [14, 13, 15].

One of the challenges of the present thesis was to understand Kyparissas' design in order to expand it and develop the associated CAD tool. Therefore, and for readability purposes, this section briefly describes Kyparissas' architecture and hardware design without delving into the details. This is the baseline hardware architecture upon which the present work was based, and many subsystems of it remain unchanged.

### 4.1 Top Level and System Specifications

All modules of the architecture are represented on the figure 4.1, with arrows showing data movement in the datapath. The yellow modules (Wizards, Memory Controller and FIFOs) come from Intellectual Property cores (IPs) generated by the Xilinx Vivado CAD tool suite for the targeted FPGA, while the rest are custom-made components designed in VHDL by Kyparissas.



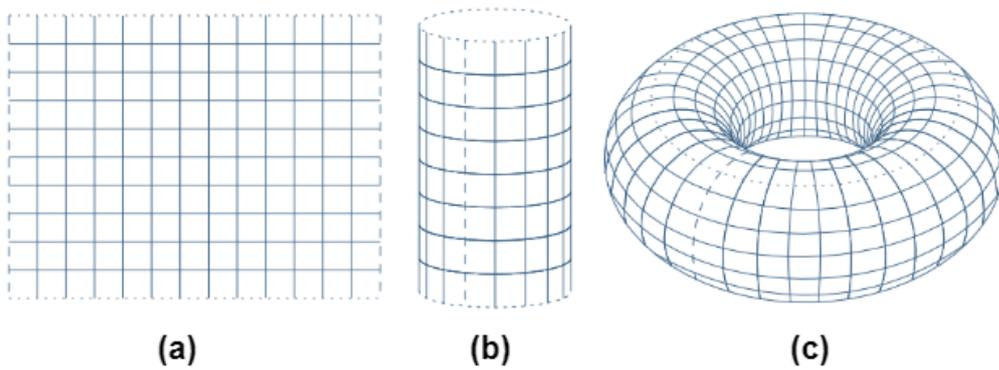


FIGURE 4.2: Supported Grid Types: (a) Rectangular, (b) Cylindrical, (c) Toroidal. (Source: [12])

Note that the *Grid Line Buffer* and *Frame Extract* are placed between two different clock domains, and they use both of the respective clock frequencies.

The design can display the evolution of the model on the  $1920 \times 1080$  grid at 60 FPS in real time. While CA theoretically evolves within an infinite space, practical constraints arise due to limited computational memory resources. Three distinct grid types are supported by this design, as illustrated in Figure 4.2. The *Rectangular* type constitutes the simplest form to be visualized, i.e. a Cartesian grid, surrounded by zero-padding so that the boundary cells can shape a complete neighborhood. A *Cylinder* is formed by connecting the vertical sides of a rectangular grid, allowing the left-most and right-most cells to belong in the same neighborhood. Finally, a *Torus* is constructed by folding the horizontal edges of a cylindrical grid, eliminating the need for boundary conditions and creating a "**periodically infinite**" lattice.

As discussed in Chapter 2, a cell can exhibit a number of different states. For example, the Game of Life requires a mere 2 states per cell, 'dead' or 'alive' ('0' or '1'). Other models demand a higher number of cell states in order to display the evolution of gradual phenomena, such as temperature changes or chemical mixtures. Furthermore, the neighborhood area can grow from small to large sizes, where, large sizes of neighborhoods can lead to a higher accuracy and reveal patterns that couldn't be emerged otherwise.

The system supports either 4-bit cells or 8-bit cells, enabling simulations with up to 16 ( $2^4$ ) or 256 ( $2^8$ ) cell states respectively, and the neighborhood size can reach up to a  $29 \times 29$  area ( $radius = 14$ ). The CA parameters (grid type, neighborhood size, states per cell, etc.) can be set within the framework (see figure 4.3), where every module inherits those settings from the top level.

```

GENERIC (
  GRID_X : INTEGER := 1920;           -- Number of cells per line
  GRID_Y : INTEGER := 1080;         -- Number of total lines
  CELL_SIZE : INTEGER := 8;        -- Cell size in bits. 4 or 8 bits.
  NEIGHBORHOOD_SIZE : INTEGER := 29; -- The NxN area of the window
  GRID_TYPE : STRING := "TOROIDAL"; -- To set the Grid Type
  BURST_SIZE : INTEGER := 128;     -- Number of bits per burst
  NUMBER_OF_BURSTS_PER_LINE : INTEGER := GRID_X/(BURST_SIZE/CELL_SIZE); -- Burst per line calculated via formula
  PALETTE : STRING := "WINDOWS";   -- Color paletter for the display
  SPEED : INTEGER := 60;           -- Simulations speed in FPS
  MEMORY_ADDR_WIDTH : INTEGER := 27 -- In bits
);

```

FIGURE 4.3: The Customizable Framework of the top level.  
Generic values are inherited by the sub-modules.

Only the *CA Engine* requires user's manual intervention for defining the transition rule, and enlarging or shrinking its sub-components according to the radius.

Adjusting even one parameter requires re-synthesizing and re-implementing the hardware design inside the Xilinx's Vivado CAD tool, a time-intensive process that demands up to an hour based on rule's complexity. This is where the re-programmable structure of the present thesis and the development of the associated CAD tool offer solutions to overcome this challenge.

## 4.2 Memory Controller and Grid Representation

The Memory Controller is a fully customizable IP, generated with the Memory Interface Generator (MIG) wizard tool and supports DDR, DDR2, and DDR3 memory interfaces. In our case, a 128MB, DDR2 memory is utilized, where the data are received and written in the form of bursts. The burst and word size of the memory are 128 and 8 bit respectively, so, 16 words are contained in a single burst.

The reading and writing operation in the memory is implemented via a handshake mechanism, as figure 4.4 depicts. Appropriate signals indicate whether a writing or reading operation occurs (command signal), the data to be read or written, the access address, acknowledgment signals, and system-reserved (DDR2 signals). The *Memory Initializer* and the *Write Back* are the only modules that access the external memory for writing, and the *Graphics Feeder* reads the data to be displayed. The same (current) frame is displayed and processed simultaneously, while the other one is being updated.

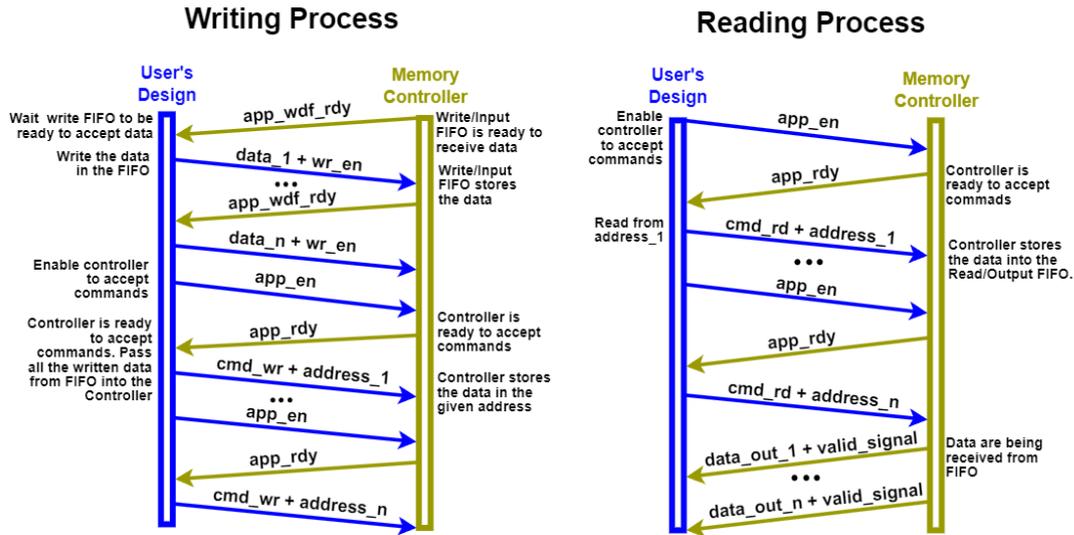


FIGURE 4.4: Handshake Mechanism for accessing Memory Controller.

The Memory Controller stores two consecutive frames of the simulation, the currently being-updated frame and the previous state of the simulation. The addressing for the two frames follows the same pattern and the access between them is switched by flipping the most significant bit from zero to one and vice versa. This technique is known as double buffering. Utilizing this technique, the old frame is not overwritten by the new one, since the whole image of the previous state is required for the calculation of the next generation.

Figure 4.5 showcases the addressing on a  $1920 \times 1080$  grid. As stated above, the burst size of memory is 128 bits and the word size is fixed to 8 bits. The cell size can be either 4 bits or 8 bits, therefore, the 4-bit cells are concatenated together in an 8-bit address. The grid is scanned horizontally by the system - from left to right and top to bottom -, an addressing pattern known as *simple horizontal scan*. Overall, the Memory's capacity is high enough to store  $2 \times 1920 \times 1080 \times 8\text{bits} \approx 2\text{MB}$  in worst case scenario (2 frames), and the number of bursts per line required is given by the following formula:

$$\text{bursts\_per\_line} = \frac{\text{Width\_of\_Grid} \times \text{cell\_size}}{\text{burst\_size}}$$

To initialize the machine, the FPGA has to be programmed with the bit file of the design, and an initial state of the Cellular Automaton must be provided to trigger the evolution. The process that the user must follow is in figure

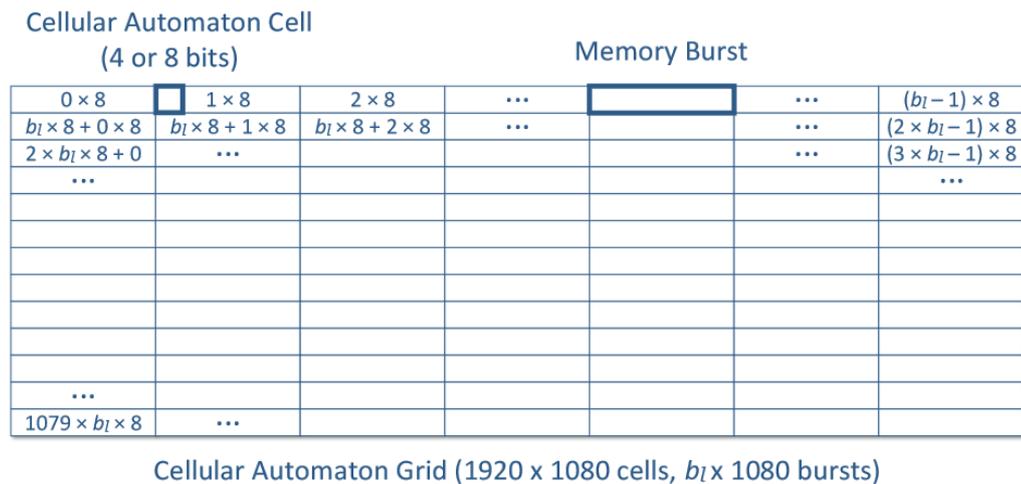


FIGURE 4.5: Grid representation in memory and burst addressing.  $b_l$  is the number of bursts per line. Source: [12]

4.6; this was a completely manual procedure in Kyparissas' thesis, prior to the development of the software CAD tool.

### 4.3 System and Memory Initialization

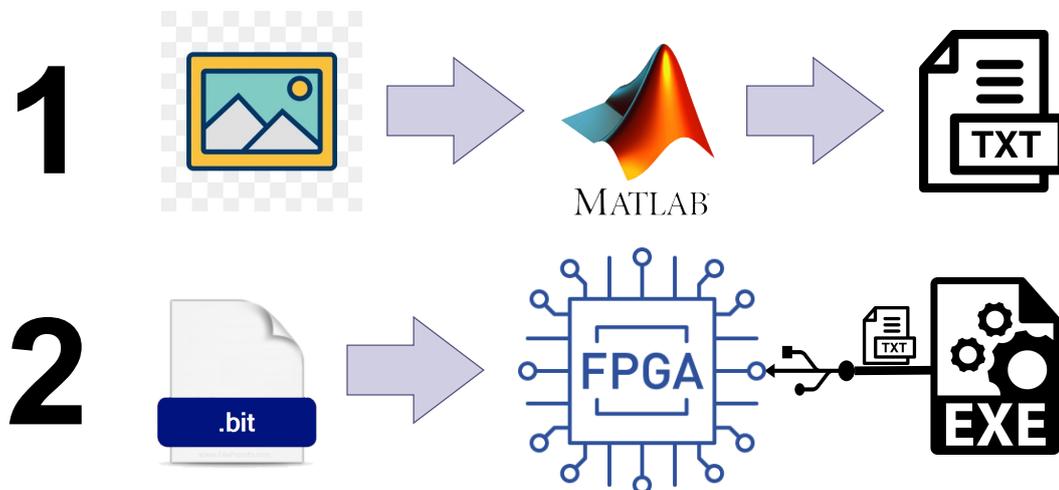
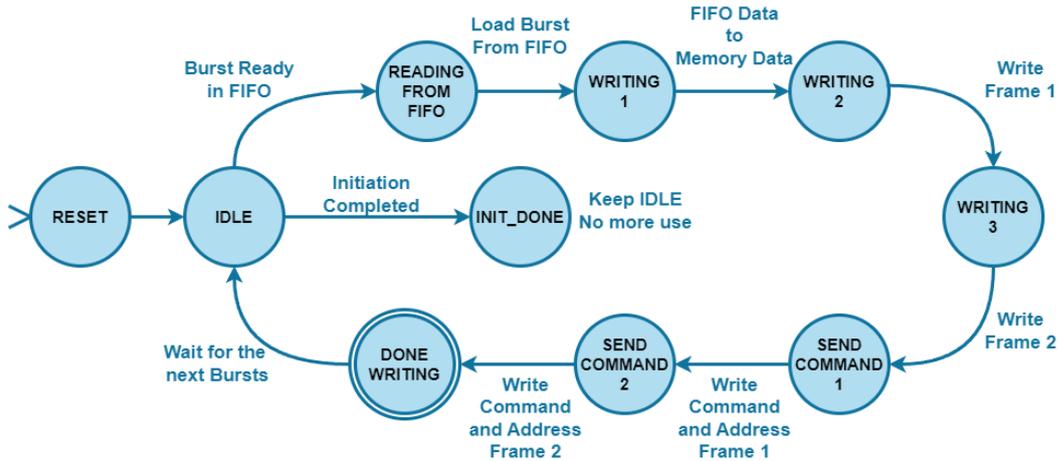


FIGURE 4.6: Initialization procedure of the machine.

The initial state of the machine (or model) is described by an image, which can be created with a drawing tool like the *Windows Paint* application. The image is converted to a text file by extracting the indices of its color palette via a *MATLAB* script. The color palette is what associates certain integer values with a color. For example, for the *Game Of Life*, a two-colored palette is required to associate the values '0' and '1' with two, different colors (mainly

FIGURE 4.7: The FSM of *Memory Initializer* module.

Back-and White, but not restricted to be so). Then, the FPGA is programmed with the already-generated bit file. Finally, by using an executable C program, the UART communication is activated, and the values of the text file are transmitted into the FPGA via a USB-A to USB-B Micro cable.

Upon the arrival of the incoming data, the *Memory Initializer* begins operating, as an FSM (see figure 4.7), and initiates handshaking with *Memory Controller*. First, the FSM delivers two identical bursts for each frame into the external memory ("WRITING" states), followed by the write commands ("SEND COMMAND" states), if the *Controller* is ready to accept them. Once the two frames have been successfully stored, *Memory Initializer* prompts the simulation to start.

## 4.4 Grid Lines Buffer

The *Grid Lines Buffer* is the most significant module of the design, rendering it capable of delivering high performance (see figure 4.8). The module consists of  $n + 1$  BRAMs ( $n$  is the diameter of the neighborhood), each one storing a whole line of the grid. It receives a burst of data from the *Memory Controller* and provides *CA Engine* with the complete neighborhood according to the selected grid type. Thus,  $(n + 1) \times c \times line\_width$  bits of BRAM resources are required. The total number of BRAMs is automatically generated by the framework, with respect to the neighborhood size and without user intervention.

The *Writer* is made up as a LUT and the *Reader* is implemented as an FSM. The re-circulation multiplexer synchronizer aids the communication

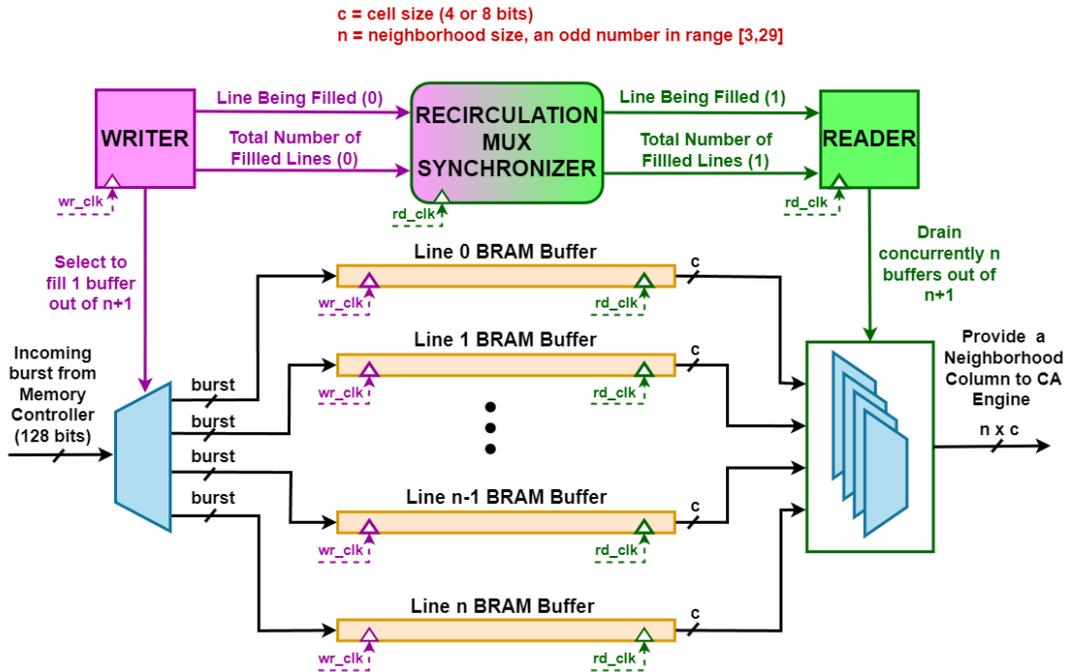


FIGURE 4.8: Grid Lines Buffer Inner Architecture

between these two sub-components, in order for the writer to not start filling a line that hasn't been drained yet by the Reader. As soon as  $n$  BRAMs have been filled, the Reader begins operating, as figure 4.9 suggests.

Preloading the neighborhood window to the CA engine before setting the valid signal, it is the key feature of the Reader in addressing vertical boundary conditions. Thus, the appropriate left-most part of the half neighborhood is shaped regarding the grid type. In case of a *Toroidal* grid, further logic and an extra amount of BRAM resources are utilized to store the upper-most and bottom-most parts of the grid. Given that the  $n$  BRAM are filled with grid lines, the neighborhood is provided to CA Engine column-by-column per clock cycle.

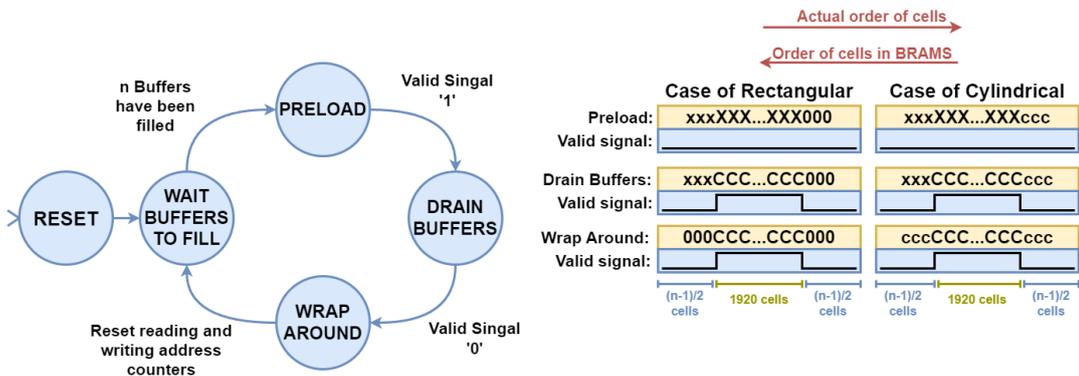


FIGURE 4.9: The Reader's functionality.

## 4.5 CA Engine

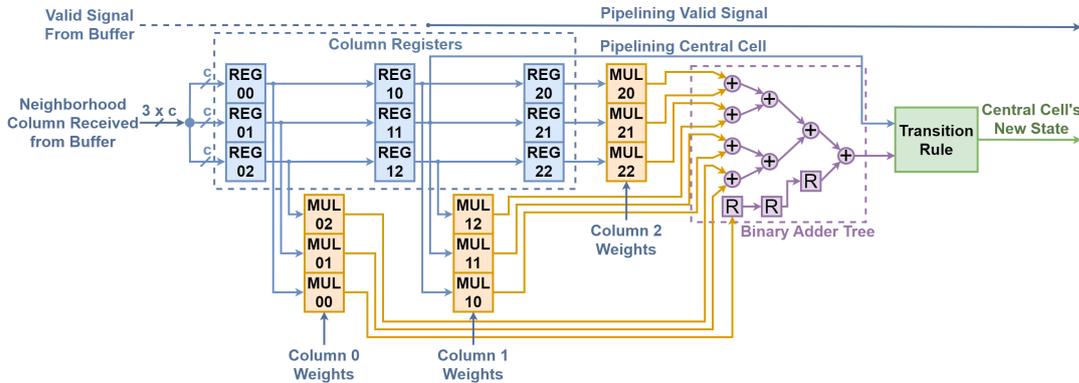


FIGURE 4.10: Datapath of the CA Engine. Case of  $3 \times 3$  neighborhood.

The *CA Engine* is a fully pipelined module running at 200MHz. It provides the next state of a cell at every clock cycle. This is possible due to its internal structure and given that the pipeline is fully filled. The *CA Engine's* datapath is represented in Figure 4.10, where the following parts can be distinguished:

1. A lattice of **Column Registers** (REGXY).
2. A set of **Multipliers** (MULXY).
3. The **Binary Adder Tree**.
4. The **Transition Rule** sub-module.

First, the lattice of **Column Registers** shifts the neighborhood columns received from *Grid Lines Buffer*. Second, each cell of the neighborhood is multiplied by the respective coefficient that the rule determines. Then, the multiplied values travel through the **Binary Adder Tree**, for calculating the total sum of the neighborhood. Finally, the total sum alongside the central cell is provided to the **Transition Rule** sub-component, which is realized as a LUT, where the next state of the central cell is decided.

The overall depth of the pipeline strongly depends on both the neighborhood size and the complexity of the rule. Given that the neighborhood is equal to  $n$  ( $n = 2 \times k + 1$ , where  $n \in [3, 29]$  &  $k \in \mathbb{N}$ ) a  $n \times n$  window of *Column Registers* is constructed, meaning that a column requires  $n$  clock cycles to cross the registers. The **Multipliers** are generated by Vivado's IP generator wizard tool and they are fully pipelined with a depth of 3 stages. The depth of the *Binary Adder Tree* also relies on the variable  $n$  and it is equal to  $\lceil \log_2(n \times n) \rceil$ . Finally, the **Transition Rule** is expressed with branch

command and requires 1 or 2 clock cycle to derive a result in terms of its complexity.

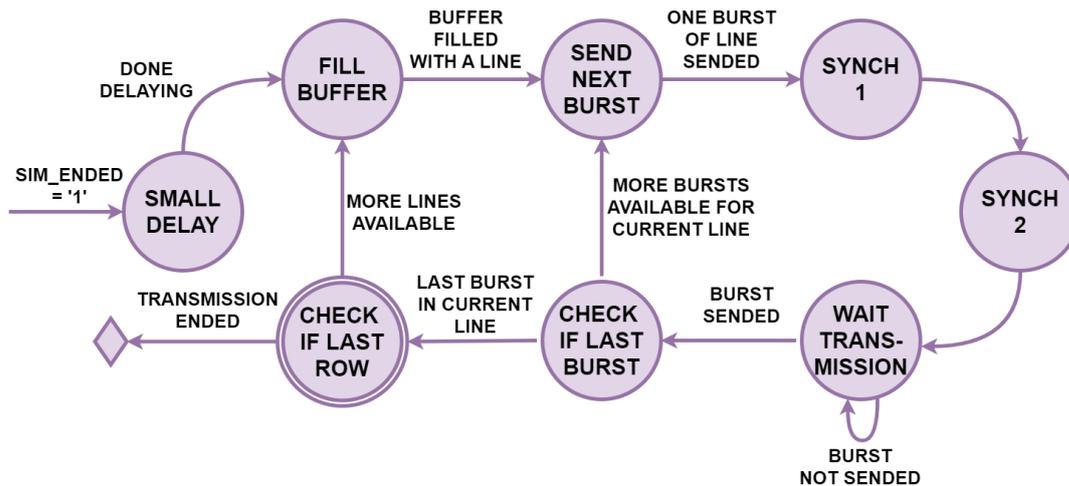
The *Central Cell* and *Valid Signal* are also pipelined. The former must arrive at the *Transition Rule* sub-component concurrently with the corresponding total sum. The latter travels in the pipeline one or two more clock cycles than the former, exactly as the *Transition Rule* demands, and enables the following FIFO for writing (see figure 4.1). The key characteristic is that, the *Valid Signal* traverses the pipeline  $(n - 1)/2$  lesser stages than the whole. As aforementioned in section 4.4, the  $(n - 1)/2$  cells are pre-loaded to shape a complete neighborhood and the valid signal is still deasserted. So, the asserted valid signal arrives at the *CA Engine*, when half of the neighborhood has already traveled in the pipeline. Thus, the *CA Engine* accepts whatever lies on the *Grid Lines Buffer's* output bus, and the valid signal is what indicates the appropriateness of incoming (from the buffer) and outgoing (to the FIFO) data.

To conclude, the *CA Engine* is the only module of the design that requires manual intervention by the user, while the others are automatically updated by the Framework. It is also written in VHDL and the user must adjust its code according to the desired CA rule. The adjustments concern, setting the appropriate number of *Column Registers*, parameterizing the weights, expanding or shrinking the *Binary Adder Tree*, redefining the transition rule, and so on. For this Thesis, the *CA Engine* has been redesigned to establish the re-programmable structure of the machine, as detailed in the next chapter.

## 4.6 Frame Extraction

The *Frame Extract* module extracts a snapshot of the simulation, as its name implies. While observing the evolution of the model in real time, a push button on the FPGA can be pressed to export the desired state. By the time the button has been pressed, the simulation seemingly stops - meaning that the same generation is being reproduced over and over again - and a signal indicates the *Frame Extract* to start operating, as the figure 4.11 suggests.

The *Frame Extract* operates as an FSM and accepts the data before they are written back into the *Memory Controller*, avoiding handshaking with it (see figure 4.1). The *Write Back* module keeps track of which line is currently being written and informs the *Frame Extract* accordingly. After a small delay,

FIGURE 4.11: *Frame Extract's* FSM functionality.

it stores all of the provided bursts of the current line into a small buffer, sends the data to the *UART* controller byte per byte, and it stops functioning when the whole state has been successfully transmitted. To convert the screenshot of the simulation into an image, the same procedure described in section 4.3 is executed, but in reverse (see Figure 4.6).

## 4.7 The Remaining Modules

In this section, we are going to briefly present the remaining modules of the design, that are yet to be introduced. This is due to the fact that, an exhaustive comprehension of these modules was not essentially required for the successful development of the CAD tool.

### 4.7.1 Graphics Controller

The *Graphics Controller* displays the evolution of the model in real-time, on a  $1920 \times 1080$  at 60Hz (FHD). It utilizes horizontal and vertical counters to keep track of which pixel is currently being displayed. Furthermore, it generates the synchronization pulses (Hs and Vs) alongside the RGB signals, as the VGA protocol prescribes.

To generate the appropriate RGB values, three different color palettes are stored in the controller (see Figure 4.12). A color palette can be realized as an array of different colors, associated with an integer value. Therefore, the integer number of the state of a cell is that that indicates which RGB values to opt for display.

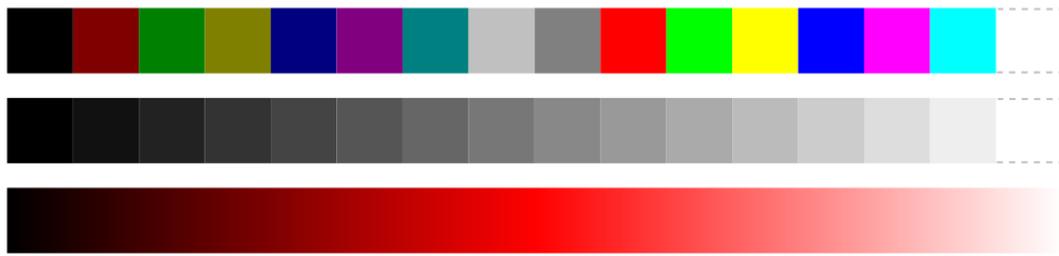


FIGURE 4.12: The embedded color palettes in *Graphics Controller*. Figure designed by Kyparissas.

The first color palette is the 16-color *Microsoft Windows palette* and it is suitable for 4-bit Cellular Automata, where the cells exhibit distinctive states. The second color palette is a 16-color, Black-and-White palette, appropriate for rules with few states that display gradual phenomena. The last palette concerns 8-bit Cellular Automata and it consists of 256 colors, which refer to Black, Red, and White shades.

### 4.7.2 Graphics Feeder

The *Graphics Feeder* loads the new frame to the Graphics Controller. When a new frame is requested by the *Graphics Controller*, the two memory segments are alternated by flipping the MSB of the address (double buffering, as described in section 4.2). Since it is the only module of the design that handshakes with the *Memory Controller* for requesting data, they are also loaded into the *Grid Lines Buffers* for processing. Thus, the  $n$ -th frame is processed and displayed at the same time, while concurrently, the  $(n + 1)$ -th frame is being updated.

### 4.7.3 Write Back

The *Write Back* receives a burst from the FIFO and writes it back into the *Memory Controller*. By the time a burst has arrived from the FIFO, it handshakes with memory, given that its access has been granted to do so. It is fully pipelined and capable of writing one burst per clock cycle, while it counts the number of already-written bursts, in order to switch between the two memory segments.

#### 4.7.4 Memory Access Arbitrator

The *Memory Access Arbitrator* regulates the access to the *Memory Controller* between *Graphics Feeder* and *Write Back* modules. The *Graphics Feeder* has priority over the *Write Back* in order to display the evolution smoothly. The *Graphics Feeder* consumes the memory bus for 25% of the time, while the rest 75% is enough to process and write a whole generation, as Kyparissas has estimated.



## Chapter 5

# Design of the re-programmable Framework

As discussed in the previous section, the user of Kyparissas' CA accelerator had to go through the Xilinx's CAD tool in order to re-configure the customizable framework. This is done with VHDL code, in order to generate the bit file of the design. Therefore, in this chapter, we demonstrate the new structure of the re-programmable framework, developed in the present thesis. Additionally, the extraction of multiple snapshots has been automated at circuit level given a time-step. Furthermore, by utilizing the *Protocol Buffer's*, we are able to distribute incoming data from software level to the appropriate hardware components. Finally, several images of the hardware (.bit files) have been generated, ready to program the FPGA for all necessary CA neighborhood sizes.

### 5.1 Overview of Extended Architecture

According to new top level view represented in figure 5.1, two new modules have been introduced, named *Deserializer* and *Serializer*. These modules concern the *Protocol Buffers* that developed for manipulating structured data in form of bytes. The former delivers the software-level-defined data into the correct hardware components, while the latter serializes the screenshot to be delivered to the outside world.

Additionally, the *SPEED CONTROLLER* was modified and implemented as a separate module, altering the speed of circuit between 0 and 60 FPS (max speed) given a time step, without relying on the push buttons of the

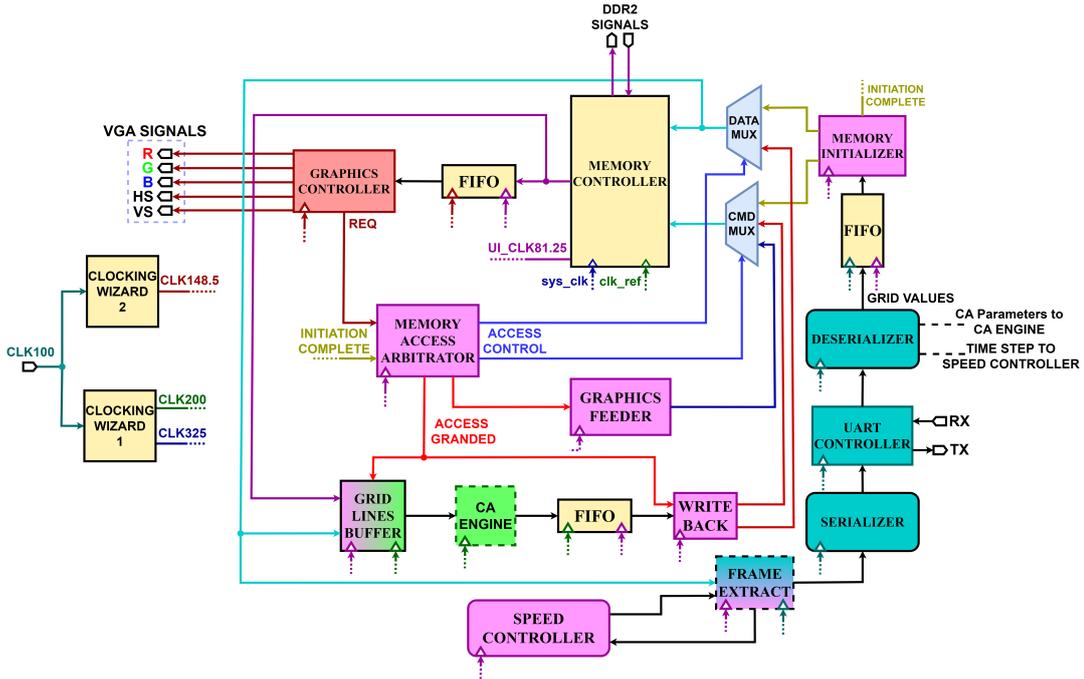


FIGURE 5.1: New top level View of the hardware architecture. Modules within dashes were modified. Rounded modules are the new ones.

FPGA. The *SPEED CONTROLLER* cooperates alongside the *FRAME EXTRACT* for extracting multiple screenshots indefinitely. Finally, the new re-programmable structure of *CA Engine* currently accepts the definition of the model from *DESERIALIZER*.

It is noted that, the data of model's definition (neighborhood, transition rule, etc.) and the time-step are determined and serialized from software level. The CAD tool which is represented in the following chapters is that, that generates these serialized data, according to user's input, and transmits/receives them to/from the architecture by means of the UART protocol.

## 5.2 Frame Extraction and Speed Control

Initially, the *FRAME EXTRACT*'s FSM arrived at a sink state IDLE after one successful extraction of a snapshot. So, the FPGA board had to be re-programmed with the same bit file to extract further screenshots. Additionally, the *SPEED CONTROLLER* accepted signals coming from the on-board buttons, in order to alter the speed of the simulation between 1 and 60 FPS, or pause it for extraction. This feature had to be abandoned, given that the access to the accelerator may be from a remote site. Therefore, the changes

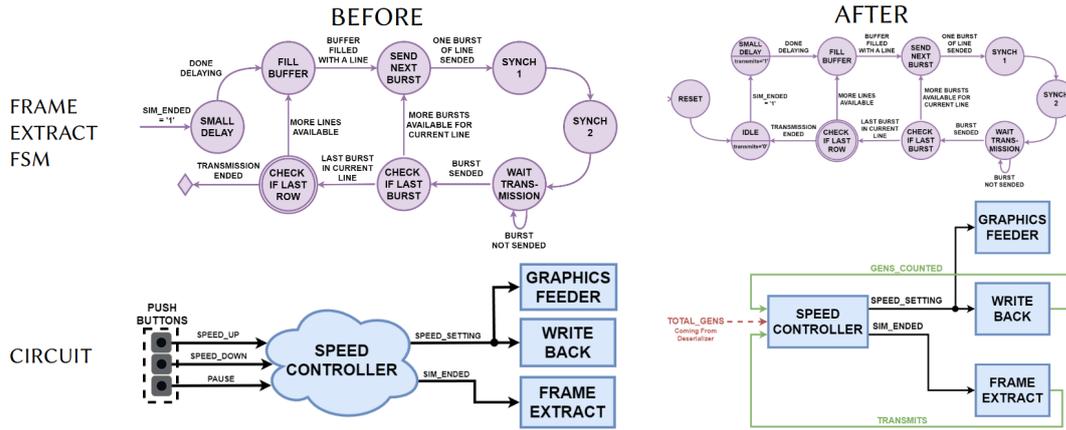


FIGURE 5.2: Changes to *FRAME EXTRACT*'s FSM along with the *SPEED CONTROLLER*'s circuitry.

depicted in figure 5.2 were developed so that the simulation can pause and continue automatically, given a provided time-step of generations.

Regarding the *FRAME EXTRACT (FE)*, a starting *IDLE* state have been added, waiting for the evolution to stop so that the start extracting phase can commence. The rest of the functionality has remained intact, while upon finishing an extraction, the FSM jumps back to the '*IDLE*' state. Concurrently, a level signal ("*transmits*") is set or unset whether *FE* is operating or not.

Regarding the *SPEED CONTROLLER (SC)*, it receives the total generations to be counted by the *DESERIALIZER*. Also, the *WRITE BACK* module informs the *SC* the total number of generations that have been counted. So, when the given time-step have been reached, the *SC* pauses the circuit and triggers the *FE*. Upon conclusion of extraction, the "*transmits*" is de-asserted and the *SC* signalizes the circuit to continue the simulation.

To conclude, the total number of snapshots to be extracted are determined from software level. In this manner, the simulation evolves and results are extracted indefinitely. Nevertheless, in the future the machine will be uploaded to the *Amazon Web Services (AWS)*, as the next step of the present Thesis. Consequently, the FPGA platform will be powered off, for example, by means of a provided Software Development Kit (SDK) from software level.

### 5.3 CA Engine's Adjustments

While the *Grid Lines Buffers* is the most important module of the design to achieve high performance due to how it loads the data from the external

memory and delivers them to the *CA Engine*, the *CA Engine*'s structure is what defines our machine as a general-purpose, Cellular Automata simulator. To create a re-programmable framework, *CA Engine* has been reconfigured in order to utilize its maximum capacity and to load the *CA*'s configurations from software level. Thus, its most general form is available in the design, and its hardware components on standby are updated with data on-the-fly, according to model's definition.

Therefore, the new structure of the *CA Engine* alongside the CAD tool (which is discussed later on) has addressed the aforementioned issues, contributing eventually to a more attractive and convenient CA simulator. The *CA Engine* always accepts  $29 \times 29$  neighborhood sizes, where smaller neighborhoods are wrapped-around with zeros. Additionally, a set of  $29 \times 29$ , fully-pipelined multipliers is available for applying the weights to each cell within the neighborhood. Finally, the *Transition Rule* component (see figure 4.10) has been replaced by a BRAM module featuring one clock cycle latency, in which, potential states are pre-stored, which are poised to replace the central cell with its next state.

As covered in the chapter 2, the CA problems are mainly categorized into two classes, totalistic and outer-totalistic rules. In the former, only the total sum decides the next state of the central cell, while in the latter case, both the total sum and the central cell determine the next state. Likewise, we are going to demonstrate the structure of the *CA Engine* step-by-step, exactly as the development process was progressed.

Before proceeding to the new adjustments, a deep comprehension of this module was essential. While studying the functionality of *CA Engine* and verifying it alongside a software level script that developed to emulate its behavior, a critical error was discovered. The neighborhood window was being horizontally mirrored (over Y'Y axis) within the pipeline during its introduction. Nonetheless, this issue didn't afflict design's correctness nor Ky-parissas' results, given the symmetrical arrangement of the already-utilized coefficients within the window.

### 5.3.1 Supporting Totalistic Rules

Starting with totalistic CA rules, the figure 5.3 showcases the hardware developed for this scenario. The principle is that when the total sum of neighborhood belongs in a certain interval, then, there is a corresponding potential

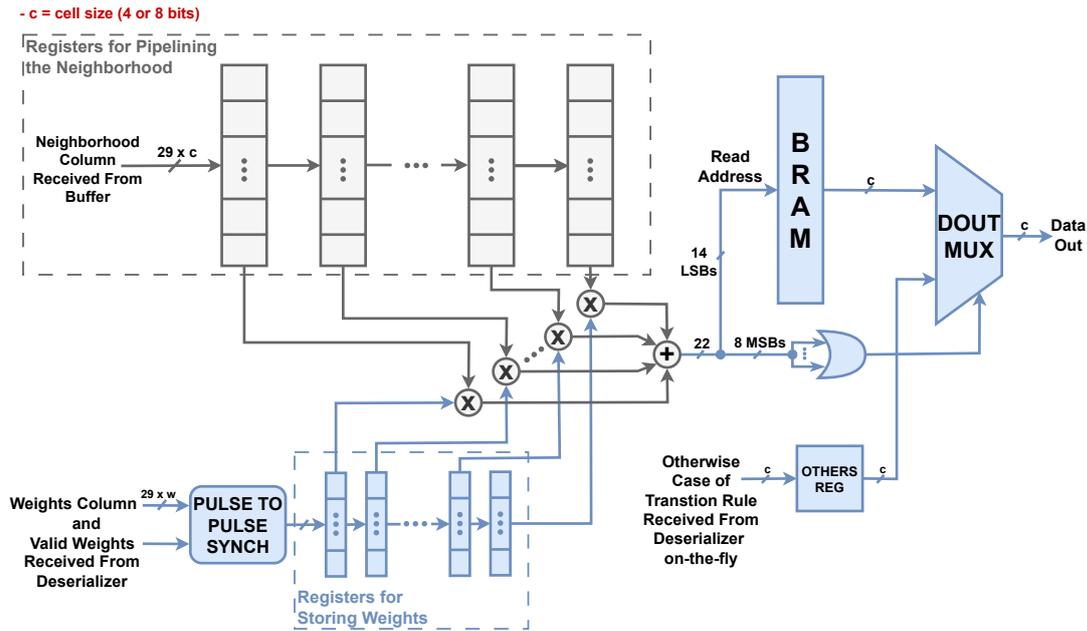


FIGURE 5.3: CA Engine's re-programmable structure for totalistic rules. Registers of neighborhood, multipliers and the adder tree was already developed.

next state. So, the limits the interval determine the addresses of the BRAM, in which the corresponding state value is stored. For example, if the next state of the cell is equal to 1, when the total sum belongs in  $[20, 132)$ , then, the BRAM addresses from 20 to 131 will contain the number 1. Thus, the total sum *per se* can be used as a pointer to access in BRAM and decide for the next state of the central cell.

Two versions of the *CA Engine* have been developed, one designed for 4-bit CA problems (allowing for up to 16 states per cell) and another optimized for 8-bit rules (enabling 256 states per cell top). Due to the limited amount of resources of our medium-sized FPGA, in this manner, the former version can support 8-bit coefficients, while the latter merely allows 4-bit weights. The product is a 12-bit integer in both version, leading to a 22-bit total sum in the worst case scenario ( $29 \times 29 \times 255 \times 15 = 3,216,825$ ). Therefore, the overall BRAM resources required to cover the worst case are:  $3,216,825 \times 4$  bits per address  $\approx 12.8$  Mbits, and,  $3,216,825 \times 8$  bits per address  $\approx 102.9$  Mbits, for 4-bits and 8-bits rules respectively, an unavailable amount to be utilized.

Instead, the total amount of BRAM resources used are:  $2^{14}$  addresses  $\times 4$  bits per address = 65,536 bits, and,  $2^{14}$  addresses  $\times 8$  bits per address = 131,072 bits, at each version. Hence, the 14 LSBs of the total sum are used for accessing into the BRAM, while the rest 8 MSBs required for overflow

detection. The overflow signal alters the selection of a 2-to-1 multiplexer (DOUT MUX). The two inputs of the multiplexer concern the data output of the BRAM and a constant value that pertain to the "otherwise" case of the transition function.

### 5.3.2 Expanding to Outer-Totalistic Rules

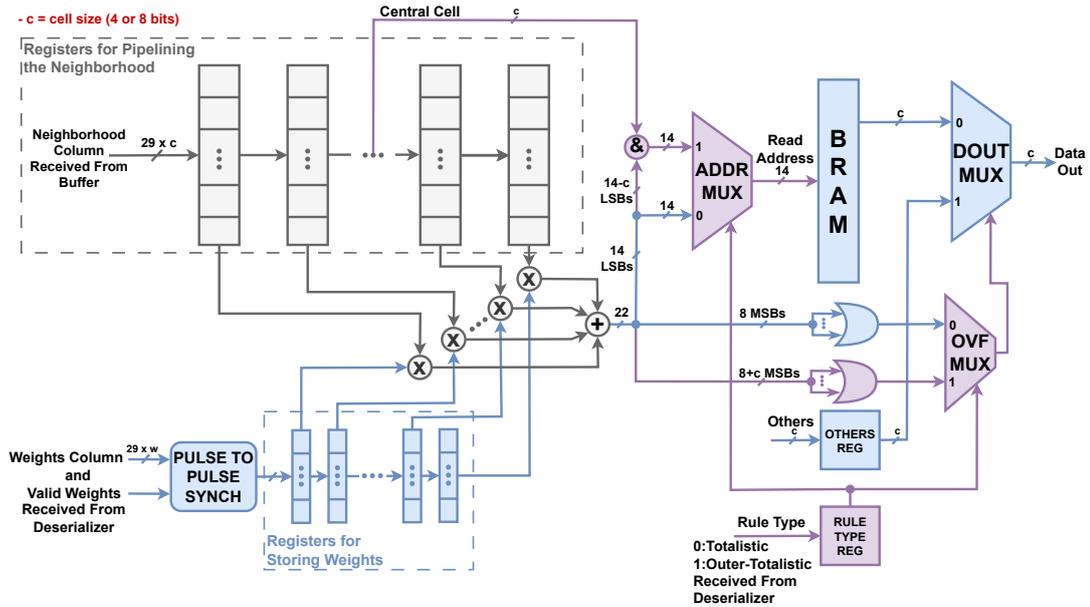


FIGURE 5.4: CA Engine's final re-programmable structure for both totalistic and outer-totalistic rules.

Regarding the outer-totalistic rules (see figure 5.4), the reading address is formed by concatenating the bits of the the central cell (current state) with the total sum. Still, the overall bits of the reading address are 14, since the same BRAM module was utilized. In case of a 4-bit CA problem, the 4 MSBs of the address refer to the central cell, while the rest concern the 10 LSBs of the total. The similar logic applies for 8-bit rules, but only the 6 LSBs appertain to the total sum. In other words, it can be imagined as the BRAM is splitted up into 16 or 256 sub-intervals according to required bits per cell. So, the central cell points which sub-interval to access to, while the total sum indicated the address within the specific sub-interval.

For example, let's consider the following, dummy, transition function:

$$c_{t+1} = \begin{cases} 2 & , \text{ if } 0 \leq \text{sum} \leq 10 \ \& \ c_t = 0 \\ 4 & , \text{ if } 0 \leq \text{sum} \leq 10 \ \& \ c_t = 1 \\ 0 & , \text{ otherwise} \end{cases} \quad (5.2)$$

If the cell size is equal to 4 bits, then there are 16 sub-intervals within the BRAM of range 1024, where each one uniquely corresponds to a cell state. The 1st sub-interval corresponds to the cell state 0, the 2nd to 1, and so on. Therefore, the addresses in range  $[0, 10]$  will contain the state 2, the addresses from 1024 to 1034 will carry the state 4, while all other addresses and the 2nd input of the *DOUT MUX* will hold the state 0 as the otherwise case claims. Unfortunately, this approach restricts the range of sub-intervals to 1024 and 64 according to cell size, nonetheless, in an high-end FPGA, these intervals can be increased by at least four orders of magnitude.

Moreover, two extra multiplexers can be distinguished in the figure 5.4, named *ADDR MUX* and *OVF MUX*. Their select signals are either unset or set for a totalistic or outer-totalistic rule respectively. Not only does the reading address differ according to the CA class, but also, the overflow bits are increased, since lesser bits of the total sum are utilized for addressing. Consequently, different signals are selected with the help of these multiplexer on a case-by-case basis.

The select signals of *ADDR MUX* and *OVF MUX*, the value of the "otherwise" case, the BRAM values and the weights are externally delivered to *CA Engine* from *Deserializer*, which is further discussed later in this chapter. The *Deserializer* operates at 100 MHz, while *CA Engine* runs at 200 MHz. In consequence, Clock Domain Crossing (CDC) techniques were utilized to properly transfer the data and avoid metastability.

Therefore, the BRAM module operates in dual-port mode, where the writing and reading ports function at 100MHz and at 200MHz correspondingly. As for the weights, we encounter numerous challenges in order to drive them properly in the *CA Engine*. Due to the excessive utilization of resources of the FPGA, many approaches led to timing constraint violations. Hence, The *Deserializer* delivers the weights column-by-column (29 weights) alongside a valid signal. The valid signal passed through a pulse-to-pulse synchronizer and the incoming column is stored in the first column of weight registers, while the rest are being shifted to the next one. By leveraging this technique, we avoided creating long datapaths, as an already developed hardware (the set of weight registers) was utilized, and the timing constraints of the design were met. Finally, a simple standard cell synchronizer proved effective for the remaining data.

## 5.4 Protocol Buffers

Before exploring how we managed to distribute the user's configurations in to the appropriate hardware components, it is essential for the reader to be aware of how the *Protocol Buffer* (or *Protobuf*) works. The *Protobuf* is an open-source, platform-neutral, data format used to serialize structured data, developed by Google in early 2001 and published on July 7, 2008. Serialization is a process, where the data are typically encoded into a sequence of bytes or characters, while serializing structured data provides us with a more convenient way to process, transmit, or store complex data more efficiently.

The *Protobuf* communication protocol utilizes **TLV** (**Tag-Length-Value**) encoding scheme for structuring the informational data. The **Tag** value indicates the kind of receiving or transmitting data according to a predetermined structure, the **Length** is pertinent to the number of bytes within the *Value* field, and finally, **Value** contains the data of the message. Let's define the following structure:

```
message Example {
    repeated uint32 msg = 1;
}
```

In the above example, a simple message has been created, called **Example**. The message definition is separated into four fields: the field label (**repeated**), the field type (**uint32**), the field name (**msg**), and the field number (**1**). The field labels can be specified as "*optional*", "*repeated*", or "*map*", where, the field can be, set or unset, repeated zero or more times, or, a pair of key-value respectively. The field type, also referred as **wire type**, has a unique ID, in our case, it is equal to 2. The **Tag** value is provided via the formula:  $(field\_number \ll 3) | wire\_type$ . Therefore, the **Tag** is calculated as follows:

$$T = ((1)_{10} \ll 3) | (2)_{10} = ((1)_2 \ll 3) | (10)_2 = (1000)_2 | (010)_2 \implies \\ T = (00001010)_2 = (0a)_{16}$$

So, the lower 3 bits of **Tag** hold the wire type and rest tell us the field numbers. Let's set *msg* equal to an array of values: [723, 234, 5], the following steps are executed to encode a number:

1.  $(723)_{10} = (1011010011)_2$  # Decimal to Binary Conversion
2. 101 1010011 # Split into group of 7-bits.
3. 00000101 01010011 # Fill with zeros (Group of bytes).
4. 01010011 00000101 # Convert to little-endian.

5. 11010011 00000101 # Add continuation bit.  
 6. D3 05 # Hexadecimal form

The same procedure is followed for the rest numbers in the array. The MSB of each byte is called continuation bit and indicates whether to continue or stop ('1' or '0') reading bytes of the present number. Consequently, this message would be encoded as: [0A05 D305 EA01 05], where  $T=[0A]$ ,  $L=[05]$ , and  $V=[D305 EA01 05]$ .

Consequently, with the use of **Protobufs**, the user's configurations are organized, and the hardware is capable of distinguishing the kind of data it receives, in order to assign them to the corresponding modules of the design.

### 5.4.1 Deserializing Data

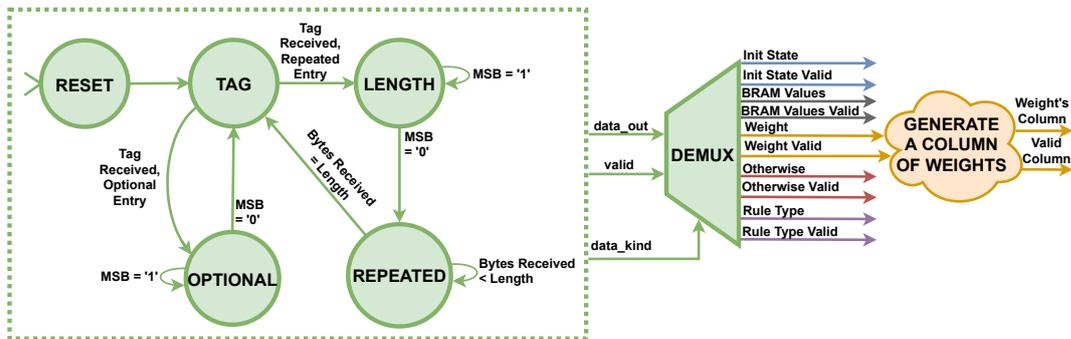


FIGURE 5.5: The FSM diagram of Deserializer Module. The **MSB** of each byte concerns the continuation or stop bit.

The *Deserializer* module, as its name encapsulates, deserializes the received data and delivers them to the proper hardware components. It accepts data from the *UART* controller byte by byte and operates as an FSM that Figure 5.5 depicts. It can be mainly realized as only having three states which follow the **T-L-V** scheme that **Protobuf** utilizes. Nonetheless, to deserialize the *Value* field, two separate states have been developed, named *OPTIONAL* and *REPEATED* (figure 5.5). In view of an optional value, the sequence of bytes concern one and only entry, where the stop bit indicates when to stop reading, thus, the *Length* field is not included in this case.

Given that the very first byte to be received is the *Tag* value, this state operates as an IDLE. When a *Tag* is being received, firstly, an internal register is updated in order to signal the type of the currently received data. Afterwards, the FSM jumps to the **LENGTH** or the *OPTIONAL* state, whether the entry is related to a repeated or an optional content. In the *OPTIONAL*

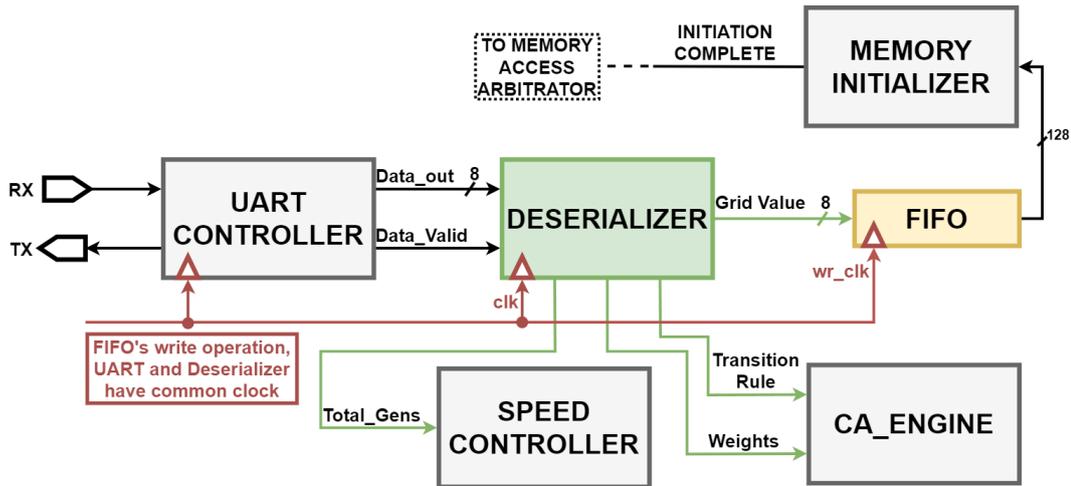


FIGURE 5.6: Connectivity of *Deserializer* module in the design.

state, the incoming bytes are being read as long as the continuation bit appears. When the stop bit is perceived, the FSM provides the data alongside a valid to a demultiplexer, and jumps back to the *TAG* state, waiting for the next data to arrive.

In case of having a repeated entry, the FSM jumps for the *TAG* to **LENGTH** state, signaling also the kind of data. The *Length* decodes the number of entries that are about to arrive within the *Value* field. Consequently, the FSM bounces to the *REPEATED* state, in which, it remains insofar as the length field claimed. When one value of the repeated entry has been successfully deserialized, the FSM outputs the data and the valid signal. Upon completion of the current entry, the FSM bounces back to *TAG* state, pending for potential data.

In order to drive the data to the appropriate hardware components, a demultiplexer has been utilized. The FSM invariably outputs a data and a valid signal. The demultiplexer is the one that separates and drives the data to the corresponding modules with the assistance of the *data\_kind* (see figure 5.5) as its select signal.

The type of data that the *Deserializer* accepts and delivers are: the initial state of the grid, the weights of the neighborhood, the transition rule, and the time step for each screenshot. As figure 5.6 demonstrates, the time step is driven to the *SPEED CONTROLLER* on-the-fly and it is stored into an internal register. The weights are transferred to the *CA Engine* in groups of 29 column-by-column. The transition rule concerns: an array of cells to be stored in BRAM, the value of the otherwise case, and type of class of the rule.

Finally, the grid values are sent to the *Memory Initializer* through a FIFO. Because the *Memory Initializer* triggers the simulation to start, the initial state is invariably arrives last from software level.

### 5.4.2 Serializing Data

The *Serializer* executes the reverse procedure of *Deserializer*. In this case however, only the state of the simulation has to be extracted to the outside world, hence, the grid values are the only data relevant to this process.

The *Serializer* module is also an FSM and it is placed in the design, as it is shown in Figures 5.7 and 5.8 respectively. The *Serializer* is connected between the *Frame Extract* and the *UART Controller*. Initially, these two modules were perfectly synchronized. The output valid data signal of *Frame Extract* enables the *Uart Controller* to transmit data, while the busy signal coming from *UART* stops the *Frame Extract* on sending data. Leveraging this interaction, the *Serializer* embedded in the design accordingly.

The FSM employs four states overall, the *Tag* and the *Length* states as they have already discussed, and two states for the *Value* field. The size of the grid will always be  $1920 \times 1080$ , and in order to know the value of the *Length* beforehand, 2 bytes per cell state are always transmitted. The worst-case scenario is to transmit the cell value 255 which requires 2 bytes in *Protobuf's* format. Cell values that are less than 127, their MSB in an 8-bit representation is '0', hence, they require 1 byte to be transmitted (in *Protobuf's* format).

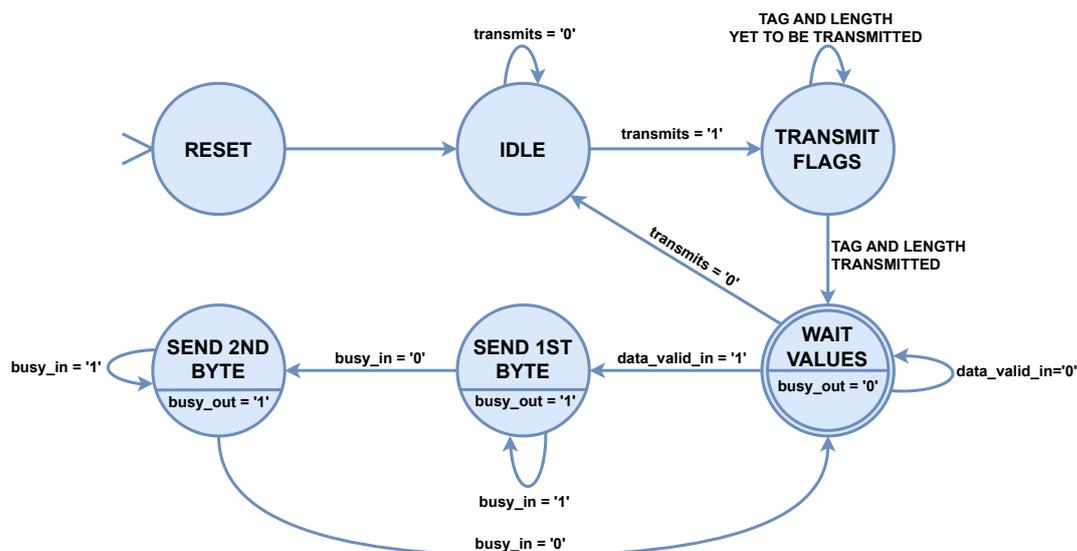
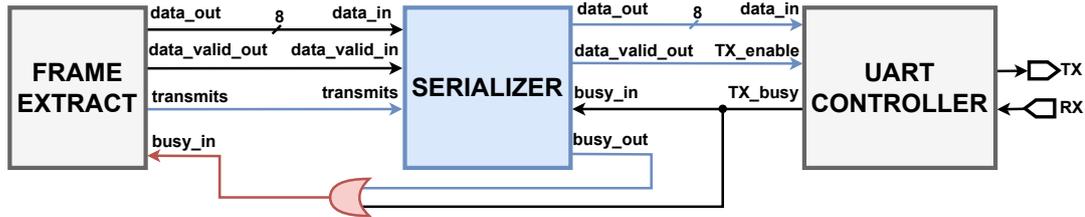


FIGURE 5.7: FSM of Serializer module.

FIGURE 5.8: Connectivity of *Serializer* in the design.

Nevertheless, the latter ones are encoded into 2 bytes, by appending 7 zeros and 1 continuation bit. For example, the number 3 will be serialized as: [10000000 00000011], and if we decode it according to *Protobuf*, the number 3 will be retrieved. Otherwise, the *Length* should be transmitted last (since we can not predict it beforehand), while the T-L-V scheme demands the order of these fields to be preserved. Consequently, the *Length* value will always be equal to  $1920 \times 1080 \times 2$  number of bytes.

To conclude, the *Serializer* module is also indicated by the *Frame Extract* that the transmission has started ("transmits" signal). Then, the well-known *Tag* and *Length* fields are immediately sent to the *Uart Controller*. Given that the *Frame Extract* is being delayed before it actually starts transmitting the cell values, there is a sufficient amount of time for the first two fields to be transferred. Additionally, because the *Serializer* requires 2 clock cycles to transfer one state, and *Frame Extract* transmits one state per clock cycle, a busy signal is also output from *Serializer* to *Frame Extract*. Furthermore, the busy signal from *Uart* to *Frame Extract* was preserved, inasmuch as the *Uart* may be busy and *Serializer* may not. Finally, when the "transmits" signal is unset, the FSM of *Serializer* returns back to the IDLE state, waiting for the next state to be extracted.

## 5.5 Assembling the complete picture

By embedding the aforementioned developed modules all-together into the initial hardware architecture, the re-programmable structure has been consolidated, ready to be driven from software level. Overall, six images of the hardware have been generated in the form of bit files. Given that 3 different bit files are required for each grid type (rectangular, cylindrical and toroidal), in addition to two extra cases of either 4 bit or 8 bit rules, their combinations resulted in a total of six distinct downloadable files.

The hardware architecture was developed using the Vivado 2019.2 tool by AMD-Xilinx. The FPGA platform used to execute the simulation was *Digilent's Nexys 4 DDR*, a medium-sized board, based on the Artix-7 FPGA series with part number *XC7A100T-1CSG324C*. The table 5.1 and figure 5.9 depicts the resources utilization, in the case of a 8-bit, toroidal grid.

TABLE 5.1: FPGA's resources utilization

Components	Availability	Consumed	Percentage
Slices	15850	15563	98.20%
LUT	63400	42581	67.16%
FF	126800	69441	54.76%
LUTRAM	19000	1890	9.94%
BRAM (Tiles <sup>1</sup> )	135	125	92.59%
DSP	240	1	0.04%
IO	210	69	32.85%

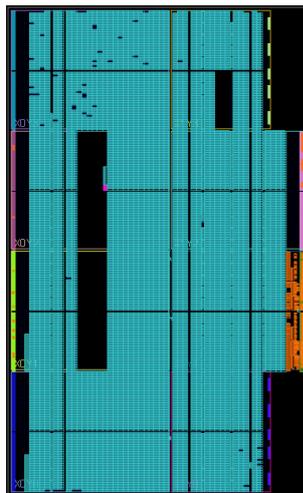


FIGURE 5.9: Implemented circuit on the device .

<sup>1</sup>Each tile is equal to 18 Kbits.



## Chapter 6

# The CAD Tool to Drive the FPGA-based Accelerator

In this chapter, we present the structure and the functionality of the tool, developed to aid in simulating Cellular Automata models and to drive the hardware design. While the user interacts with the tool by means of a GUI environment, this chapter mainly focuses on its back-end operations.

### 6.1 Overview Of The Tool

Our tool is primarily constituted by Python, a high-level, general-purpose programming language, alongside a minuscule, TCL script to run the *Vivado Design Suite* in the background. The primary purpose is to fully automate the initialization process of our machine, to load user's configuration in the hardware, and finally, to extract screenshots of the evolution given a specified time step.

The user merely has to specify the initial state in the form of an image, the parameters of the CA (weights, grid type, number of state, etc.), and the transition rule via the GUI. Inasmuch as everything is prepared and the users opts to start simulating, the following operations are executed by the tool:

- Image to array conversion and vice versa.
- Serialization and deserialization of data.
- Select one out of six bit files based on parameters
- Program the FPGA with the selected bit file.
- Transmission/Reception of data using UART protocol.

The main flow of back-end is depicted in the figure 6.1, showcasing the exact order of execution of the aforementioned operations.

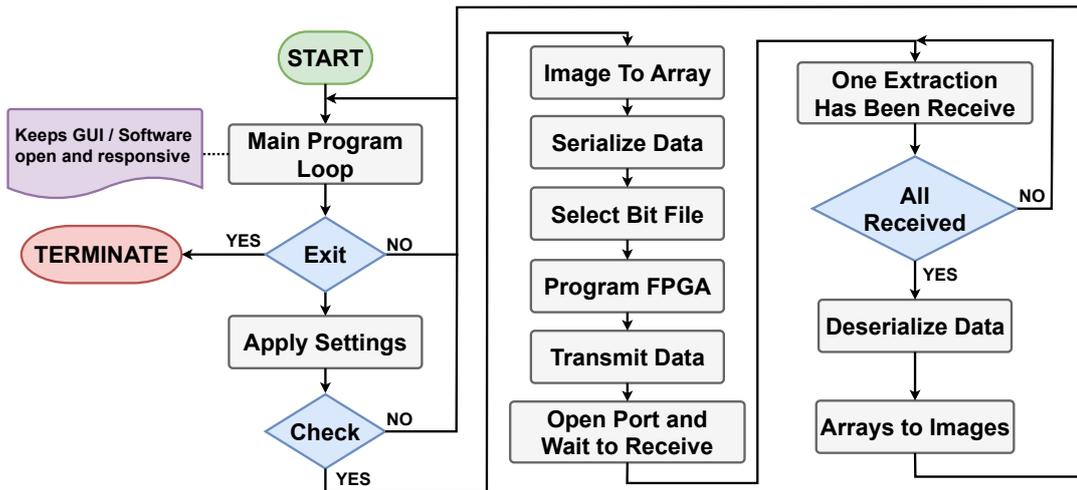


FIGURE 6.1: Flowchart of the back-end functionality.

## 6.2 TCL Scripting

The TCL stands for **T**ool **C**ommand **L**anguage and it is a high-level, interpreted, programming language. The TCL is integrated in the *Vivado Design Suite* environment and it offers the ability to control the IDE programmatically. The *Vivado* can be launched using three different modes:

1. **TCL Shell Mode:** Enter individual TCL commands or run TCL scripts in the *Vivado Design Suite* Shell, either inside or outside of the Vivado IDE.
2. **TCL Batch Mode:** The *Vivado Design Suite* shell opens, executes the specified TCL script, and then, exits when the script completes.
3. **IDE Mode:** Opens the GUI environment.

In our case, the **TCL Shell Mode** is used, since individual TCL commands are sent into the Vivado shell, as the following example represents:

```

set argv [list arg1 arg2 ...]
set argc [llength $argv]
source %path%/my_script.tcl
  
```

The arguments that the script accepts and their total number require two individual commands, plus, one extra command to run the TCL script. A sub-process spawns in the background which opens the Vivado shell and the above commands are sent through a pipe. Our script file operates in two, different, independent modes: "compile" or "program". The former performs: *Compilation* of the design sources, *Logic Synthesis*, *Implementation*, and

*Bitstream Generation*, while the latter *Programs* the FPGA with the specified bit file. In this regard, the *Vivado Suite* is driven in the background.

The compilation style of the *Vivado* varies between *Project Mode* and *Non-Project Mode*. In *Project Mode*, the *Vivado* tools automatically manage the design flow. All of the design data required by the tool (netlist files, log files, reports of the design, etc.) are auto-organized in the hard drive disk, in a specific, structured directory. On the other hand, *Non-Project Mode* performs in-memory compilation, where the design sources can be stored in a simple custom-made directory, and the files required by the *Vivado* are temporarily generated in RAM. The *Non-Project Mode* is more suitable, when someone desires to perform a simple compilation of the design and program the FPGA on the background, as it is faster and requires less storage.

Although the TCL script operates in the two aforementioned modes, "compile" and "program", the former was not incorporated in the tool. Initially, it was thought to be a good practise to generate VHDL files and then compile them to generate a new bit file, if the already developed hardware wouldn't support a user's configuration. Eventually, it was proved to be out-of-scope for this Thesis, while consuming time for developing this functionality could lead us out of the main goal. To conclude, the TCL commands used by our tool are illustrated in the table 6.1, below.

TABLE 6.1: The TCL commands that are used by our tool.

Command	Description
set_part	Specify the part number of the FPGA.
set_property	Define any property of an object in the design.
read_vhdl	Read VHDL files.
read_ip	Read IP files.
generate_target	Generate output products of an IP.
synth_ip	Synthesize an IP.
synth_design	Synthesize the design.
opt_design	Perform high-level design optimization.
place_design	Place the design.
synth_opt_design	Perform physical logic optimization.
route_design	Route the design.
write_bitstream	Generate the bitstream file.
open_hw_manager	Open hardware manager.
connect_hw_server	Connect to the hardware server.
program_hw_devices	Program the FPGA.

### 6.3 CA Description Language (CDL)

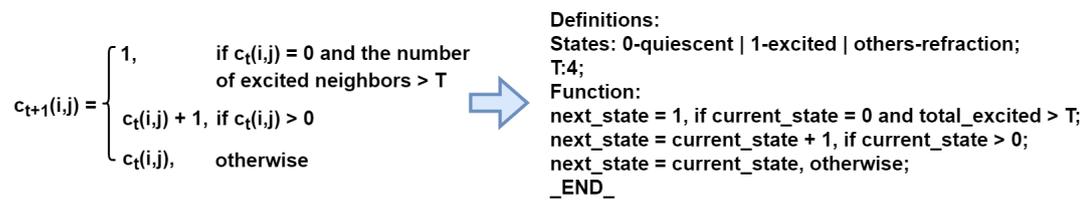


FIGURE 6.2: The Greenberg-Hastings Model described in CDL.

The Cellular Automaton Description Language, or CDL, has been developed to help the user define the transition rule of a model. The CDL is determined in the form of plain text, writing in a text editor embedded in the GUI. It has been intentionally designed to be as simple as possible, especially for individuals; unfamiliar with programming. In terms of coding, it is truly convenient to create a language for this kind of problems, because, having specific, predefined sets of syntax rules it can be interpreted to manageable structures for a program. Therefore, the *Lex* and *Yacc* tools were used to create a compiler for our CDL.

At this point, we have to explain the syntax rules of the CDL by using the example in Figure 6.2. The example represents the transition rule of an excitable model called *Greenberg-Hastings*. Our CDL is structured into two subsections: the *Definitions* and the *Function*.

The *Definitions* is an optional field, depending on the definition of the model. In this subsection, the user can entitle several states of a cell, using its value alongside a desired characterization. The syntax is restricted to the pattern "**value**<sub>1</sub> - **name**<sub>1</sub> | **value**<sub>2</sub> - **name**<sub>2</sub> ...", while the keyword **others** may be used to refer to rest (also optional), non-defined states. Additionally, it is possible to declare constant variables, if need be.

In the subsection *Function*, the mathematical equation that describes the model is determined, according to the following pattern:

```

next_state = <expression1 >, if <condition1 >;
next_state = <expression2 >, if <condition2 >;
:
next_state = <expression >, otherwise;
  
```

The initiation of each statement is restricted to the phrase "**next\_state =**", where the **otherwise** and an at-least one "non-otherwise" condition are mandatory. In order to shape an **expression** or **condition**, the following keywords or symbols are allowed to be used:

- **Arithmetic Operators:** "+", "-", "\*", "/".
- **Logical Operators:** "AND", "OR", "NOT".
- **Comparative Operators:** "=", ">", ">=", "<", "<=".
- **Keywords:** "sum", "current\_state" and "total\_name<sub>x</sub>", where name<sub>x</sub> is a entitled state in *Definitions* section.
- **Constants or Numbers:** Only the defined constants or numbers  $\in Z$ .
- **Parenthesis:** "(", ")", for prioritizing operations.

The keyword "sum" refers to the total sum of the weighted neighborhood, the "current\_state" to the state number of the central cell. The "total\_name<sub>x</sub>" counts the occurrences of certain state-value within the window. Furthermore, the user may use the expression, i.e. sum in  $[x, y)$ , where  $x, y \in Z$ , to define more conveniently the intervals, in which the total sum belongs to.

Nonetheless, the current structure of the re-programmable framework does not support all of the above functionalities of the CDL. For example, adding multiple keywords "total\_name<sub>x</sub>", the same amount of Binary Adder Trees are required to be preinstalled in the *CA Engine* as well. Due to the limited amount of resources, it wasn't feasible to further extend the hardware. Notwithstanding, since the one of the main challenges of this Thesis was the user's input, the CDL were designated to its most general form. At present, a basis core has been established for future development.

The most general form of the already supported CDL is provided to the following example of transition rule:

```
next_state = X, if sum in [Y,Z] and current_state = W;
.
.
.
next_state = G, otherwise;
```

The  $X, Y, Z, W$  and  $G$  belongs in  $Z$ , where the interval's definition supports all possible combinations of brackets and parenthesis ([, ], ( or )). The above example concerns an outer-totalistic rule, where if the part "and current\_state = W;" is omitted, then a totalistic rule has been determined,

where multiple cases of *next\_state* are allowed, as long as the sum intervals can fit in the BRAM.

It can be said that our CDL is expected to be truly convenient to the user because if we replace the "next\_state", "sum" and the "current\_state" keywords with mathematical symbols, such as  $f(t + 1)$ ,  $\Sigma$  and  $f(t)$ , correspondingly, our CDL follows an identical pattern to that of the mathematical equations adhere to. Thus, it does not demand the slightest programming knowledge whatsoever.

### 6.3.1 Interpreting the CDL

The *Lex* and *Yacc* tools have been utilized to build our domain-specific compiler. These tools implemented within the PLY library to provide compatibility with Python. The compiler ensures the syntax correctness of the user's input and translates the CDL into a structure with a specific format. Then, the tool reads this structure and generates the LUT to be stored into the BRAM. Similarly, we going to distinguish two scenarios, for totalistic and outer-totalistic rules, and explain the concept with the help of dummy examples.

#### Totalistic Rules

For this case, let's consider the following paradigm:

```
next_state = 1, if sum in [0,10];
next_state = 2, if sum in (15, 20];
next_state = 3, if sum in (23,78);
next_state = 0, otherwise;
```

Since the syntax is correct, the compiler will provide the ensuing output in a text file:

```
LUT=1:[0,10] | 2:(14,20] | 3:(23,78) | 0:others
```

According to the format, "S:[a, b]" or "S:others" (for otherwise), S is related to the next state and a,b are the limits of the interval. The delimiters ":" and "|" separate the state from the interval and each case respectively. An array with 16384 addresses, matching the total number of BRAM addresses, is filled with the next state-values in the corresponding addresses that sum indicates, as the figure 6.3 showcases.

Address	0	1	...	10	11	...	16	...	21	...	24	...	77	78	...	16383
Value	1	1	...	1	0	...	2	...	0	...	3	...	3	0	...	0

FIGURE 6.3: An example of BRAM, given the reference example.

### Outer-Totalistic Rules

As for an outer-totalistic rule, let's examine the following example:

next\_state=1, if sum in [0, 10] and current\_state=1; (1)

next\_state=2, if current\_state=1; (2)

next\_state=3, if sum in [0, 10]; (3)

next\_state=0, otherwise;

Similarly, the compiler will generate the subsequent string:

LUT=1:C1:[0,10] | 2:C1 | 3:[0,10] | 0:others

The only addition (in comparison to previous case) is the "CX" sub-string, declaring that the current state is equal to X for the specific case. In the case (1), both the sum and the current state exist in the condition, so, the LUT string contains both the current state-value and the interval (1:C1:[0,10]). In case 2, only the central cell appears in the condition, hence, the interval is omitted (2:C1), while the case (3) has already been clarified.

In coding level, tracing an at least one character 'C' in the LUT string, it is certain that an outer-totalistic rule has been provided. It is reminded that, the BRAM is divided into 16 or 256 sub-intervals for 4-bit or 8-bit rules correspondingly. The current state indicates the sub-interval to access, whereas the total sum points to a position within that specific sub-interval. So, according to the above example, the figure 6.4 depicts how the array/BRAM will be formed in case of cell size = 4 bits.

Given the figure 6.4, the addresses for 1024 to 1034 contain the value 1 owing to the condition (1). Alternatively, if the current state was equal to 9,

Address	0	10	11	...	1023	1024	...	1034	1035	...	2047	2048	...	2058	2059	...	
Value	3	...	3	0	...	0	1	...	1	2	...	2	3	...	3	0	...
	current_state=0					current_state=1					current_state=2						

FIGURE 6.4: An example of BRAM, given the reference example. Case of cell size = 4bits

then,  $9 \times 1024 = 9,216$ , so the number 1 would be stored from 9,216 to 9,226. The remaining position within the second sub-interval store the value 2 because of the condition (2). Finally, all of the 11 first positions of the remaining sub-intervals hold the value 3 due to the condition (3).

In the above example there exist overlaps of sub-intervals among the conditions, where the condition (1) has priority over (2), the (2) over (3), and so on. This does not happen due to how they were written, but due to how the rule should be defined properly. The tool properly treats the priorities over cases accordingly, without taking into account the written order, avoiding to overwrite values of cases with higher priority.

Every address of the array is initialized with the value  $-1$ . The highest priority case (1) overwrites every position, whether the  $-1$  appears or not. The limits of the sum intervals of case (1) are preserved in a list. So, it is not allowed for the case (2) to overwrite the positions that exist in this list, while it overwrites all the others. The lowest priority case (3) only updates the array whenever a  $-1$  exists in the respective positions. Finally, after the completion of this process, the remaining non-updated values of the array (still holding "-1") are being replaced with the state value of the otherwise case.

The procedure, as described above, can be generalized into a mathematical expression. If the next state  $= s \in Z$ , central cell  $= n \in Z$  and  $a, b \in Z$  are the lower and upper bounds of the sum interval respectively, it is implied that,  $c(t+1) = s$ , when  $sum \in [n \times L + a, n \times L + b)$ , where  $L$  is the length of each sub-interval and it is equal to 1024 or 64 for cell size = 4 or 8 bits correspondingly.

## 6.4 Serializing/Deserializing Data

While Protobuf's serialization and deserialization process required to be manually implemented on the hardware level, in software level this process is much more simplified. Firstly, the structure of the message to be sent is being described as plaintext; in a file with extension ".proto". Our message structure is the following:

```
message DataStruct {
    optional uint32 time_step = 1;
    repeated uint32 weights = 2;
```

```
    repeated uint32 bram_values = 3;
    optional uint32 others = 4;
    optional uint32 rule_type = 5;
    repeated uint32 initial_state = 6;
}
```

Each variable in the above message corresponds to a different kind of data to be transmitted, while their names encapsulate their content. As long as our data are finally structured, ".proto" file is being compiled and the corresponding file descriptors are automatically generated. The file descriptors outline the serialization and deserialization procedure. Using the appropriate methods provided within the protobuf's libraries, the data are converted to a sequence of bytes. These bytes are transferred and received with the UART protocol one-by-one.

## 6.5 Image Conversion

At first glance, a Bitmap image had to be converted to an ASCII file and vice versa, by means of a Matlab script which was executed manually. In order to embed this functionality in our tool, we utilized the Pillow (PIL) library, a powerful image-processing library compatible with the Python interpreter.

Bitmap images appear to be very handy for our application owing to their uncompressed format. On the other hand, a compressed image format, like JPEG or PNG, would require a decompression algorithm in order to fully retrieve the correct integer color-values. So, the tool only supports Bitmap images.

Using the PIL library, the information of an image can be extracted using *palette* mode. This way, the indexes are separated from the color palette, which are transmitted to the machine as its initial state. Conversely, when a snapshot extracted and received from the machine, the color palette (of the provided initial state) is applied to it, drawing the image of a future generation.

The user has to create the initial state properly and to include the correct indices within the image file. The tool merely extracts the indices of the input image, stores the color palette, and checks if the state-values comply with the

definition of the model. A cell state can not be greater than  $n - 1$ , where  $n$  is the overall states of the CA.

## Chapter 7

# The Graphical User Interface CAD Tool to Describe the CA Model

The majority of the CA machines, as described in the *Related Work* (Chapter 3), were configured via either a low or a high level programming language, such as *C* or *Assembly*. This characteristic; though, limits the potential number of users to mainly computer engineers, while the CA models primarily concern physicists, biologists and mathematicians. Even though CAM-8 and CEPRA-1X supported a CDL environment to enhance convenience, it was somewhat complicated to utilize (see Figures 3.4, 3.5).

This was the case with Kyparissas' baseline architecture: the user needed to write VHDL code to represent the CA model rules (generally this was a variation of some template which was already provided), subsequently the design had to go through the Xilinx CAD tool suite, and the resulting file was downloaded in the user's FPGA for execution. Given that the purpose of this thesis is to allow for remote access of non-FPGA conversant users, it was deemed appropriate to develop a CAD tool in order to facilitate the model description (e.g. selection of weights and transition rules). This CAD tool is not associated with the target technology in terms of its use (but conforms to the capabilities of the system, as described in previous chapters), and it allows for easy CA model description.

In this chapter, the GUI environment of our tool is represented, showcasing its capabilities and the convenience that provides to the end-user.

### 7.1 The GUI Environment

The Graphic User Interface (GUI) was developed by using the *Tkinter* package, the standard Python API for interacting with the Tcl/Tk GUI toolkit,

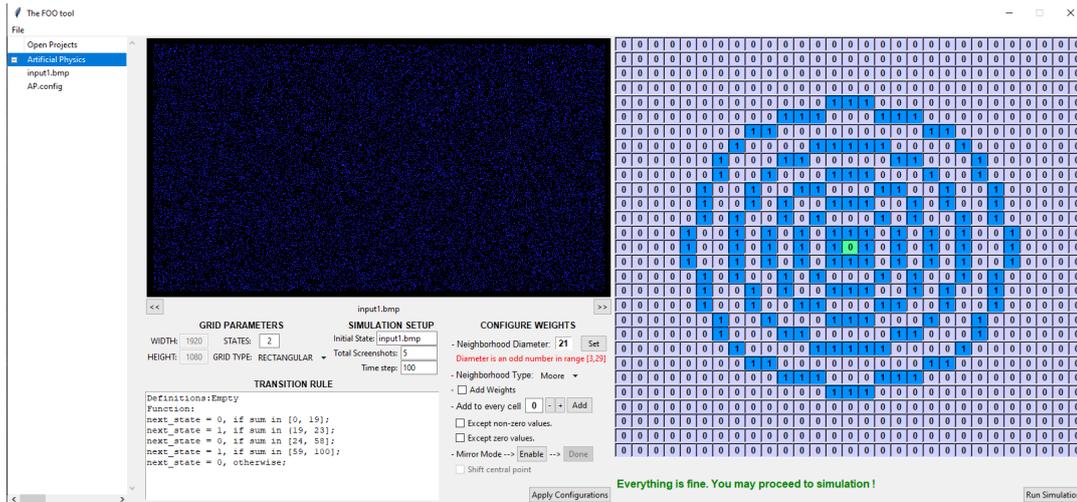


FIGURE 7.1: The GUI environment

which is available on both Unix-like platforms and Windows Systems. The primary reason that led us to develop this environment is the large window of the neighborhood. Inserting  $29 \times 29 = 841$  coefficients one by one is surely painstaking. Without the GUI, the same scripts could generate different patterns of, weighted or not, neighborhood windows. Still, the GUI offers more capabilities, as shown later in this section

Figure 7.1 displays how the environment is organized. The following parts of the GUI can be distinguished:

- **Toolbar:** Placed at the uppermost part of the window.
- **Treeview:** Placed at the rightmost section of the window.
- **Image Viewer:** At the middle of the window.
- **Configurations:** Beneath the Image Viewer.
- **Neighborhood:**  $29 \times 29$  entries at the leftmost part of the window.

## 7.2 User Options

In the toolbar, only the "File" menu is available so far, that contains a sub-menu where the user can *Create*, *Close* and *Export* a project, *Import* files or *Exit* program. If there are no opened/created projects, all the widgets of the GUI are disabled and the user can not interact with tool whatsoever. The user has to create a new project, where the steps to do so are demonstrated in Figure 7.2.

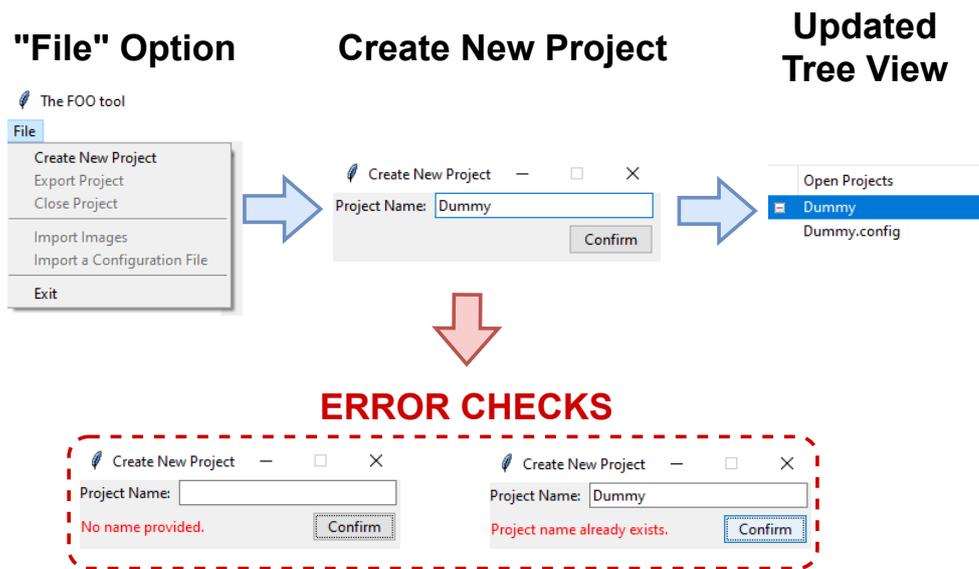


FIGURE 7.2: Creating new project.

On creating a new project, a pop-up window spawns on top of the main window, in order for the user to type the desired project name. If no project name has been provided or the inserted project name already exists, an corresponding message is printed. When the project has been successfully created, a default configuration file also imported, where the number of states per cell is set to two, the transition rule is empty and a  $29 \times 29$ , non-weighted, Moore neighborhood is displayed.

Each project can contain only one configuration file and multiple images. If there are available images in the selected project, they can be navigated via the *Image Viewer*. The selected project is highlighted with blue background and the tool automatically recognizes it. Therefore, the user doesn't have to specify which project is currently selected. There is always a selected project highlighted in the tree view, and, if the user creates or closes a project, the newly made or the first one in the *Treeview* is auto-selected respectively.

In this manner, the user can switch between open projects by merely clicking in the *Tree View*. On clicking to a different, non-selected project, the tool, firstly, auto-saves the already inserted configurations, and secondly, auto-imports the new ones. The former feature is truly important due to the structure of how projects are managed with the tool. Otherwise, if the user clicked back to an already configured project, the settings would have been vanished, or, the tool should accordingly notify the users; every time they tried to switch between projects. In the end, users' settings are invariably preserved, and they no need to fret on saving their work.

During on importing files or exporting a project, the file explorer pops-up, so as to navigate to the preferred directory. If the user aims to import a configuration file, the tool notifies prior to proceeding that the already existed one will be overwritten. In case of importing images, there is a possibility of duplicate file names. So, the user is provided with the alternatives to either overwrite or rename them. Last but not least, a zip file that contains the configuration file and the images of the selected project can be exported via the *Export a Project* option.

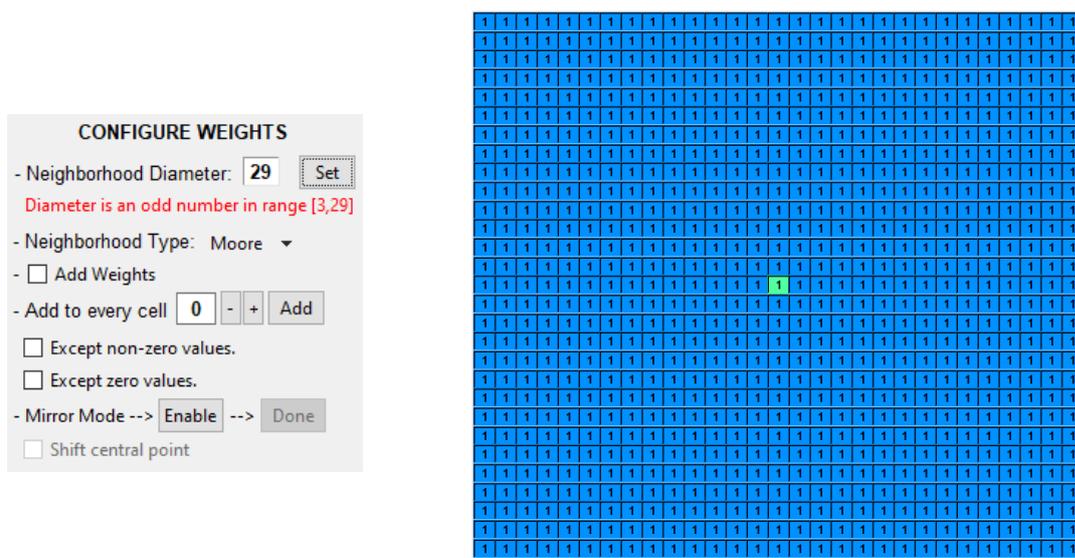
### 7.3 Configuring Weights

Given that up to 841 coefficients can be determine, the tool provides users with several mechanisms to increase convenience. Figure 7.3 displays the widgets that contribute to shape the neighborhood. The following functionalities are available:

- **Diameter Selection:** Must me an odd number, otherwise the central cell can not be determined.
- **Neighborhood Type:** A drop down menu where several patterns are available to be drawn
- **Add Weights:** Adding coefficients greater than 1.
- **Add to every cell:** The constant value within the entry.
- **Mirror Mode:** Mirrors the entries in the second quadrant of the lattice.

In order to draw patterns, which will be further discussed in this later section, all of the entries correspond to a unique,  $(x, y)$  coordinate, as if they were placed on  $XY$  axis. Given that there are  $29 \times 29$  available entries, the uppermost, left entry correspond to coordinate  $(-14, 14)$ , the lowest, right entry to the coordinate  $(14, -14)$ , and apparently, the central cell is coordinated as  $(0, 0)$ . Thus, using simple mathematical equation, numerous pattern can be drawn.

When the user sets a new diameter of the neighborhood, all entries outside the radius are disabled and set to 0. If the checkbox "Add Weights" is checked, the values of the entries are set in analogy with their distance from the central cell. The distance is simply given by the formula  $\max(x, y)$ , where  $x, y$  are the coordinates of the respective entry. Figure 7.4 depicts these two functionalities.



(a) (b)

FIGURE 7.3: Configuring weights. (a) Widgets for shaping the neighborhood, (b)  $29 \times 29$  lattice of entries.

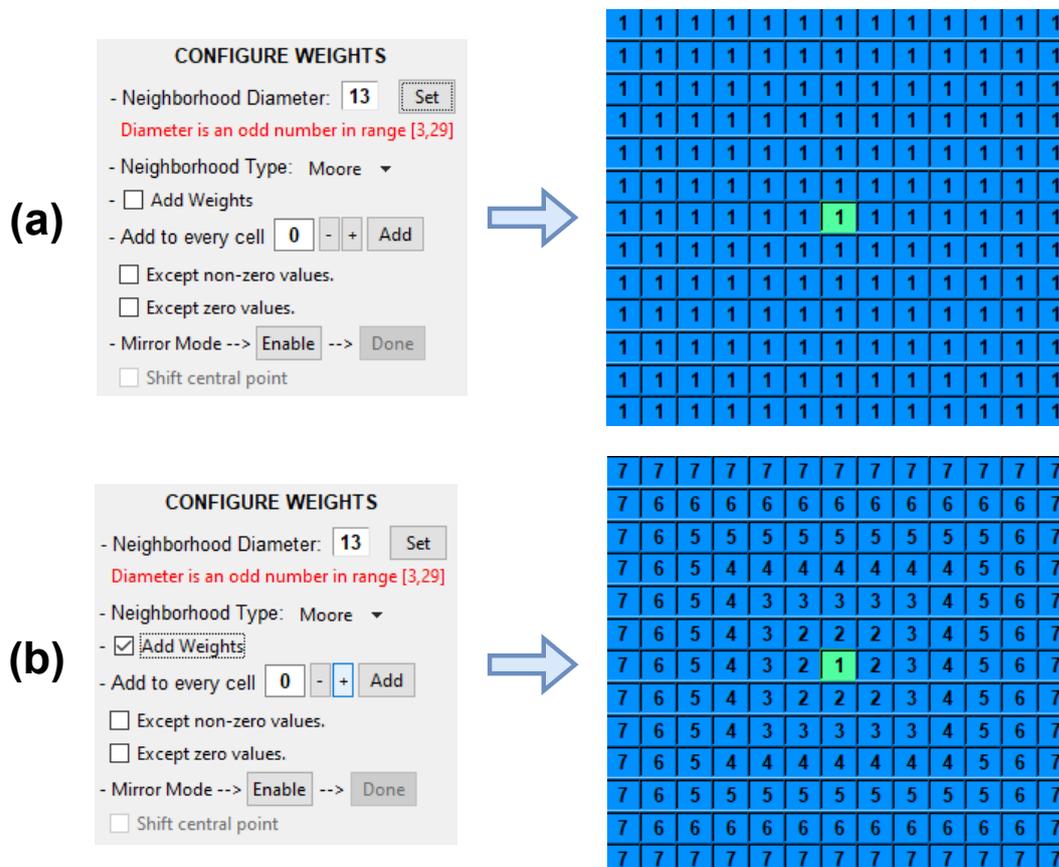


FIGURE 7.4: (a) Setting up a  $13 \times 13$  neighborhood, (b) Adding weights.

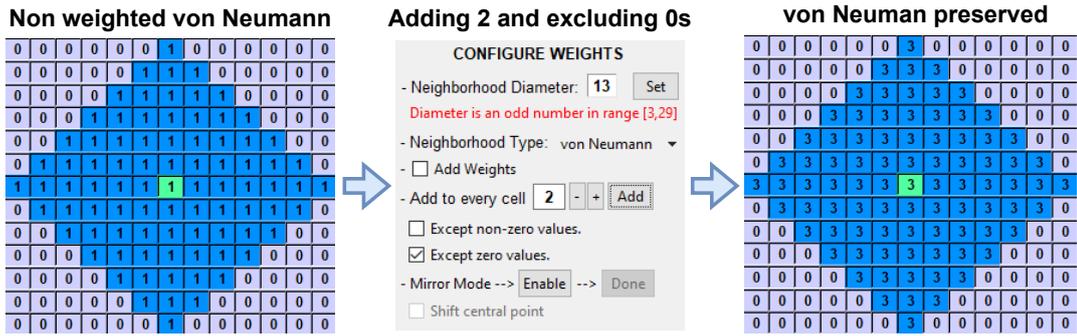


FIGURE 7.5: Adding 2 to every cell excluding zeros, in a von Neumann region.

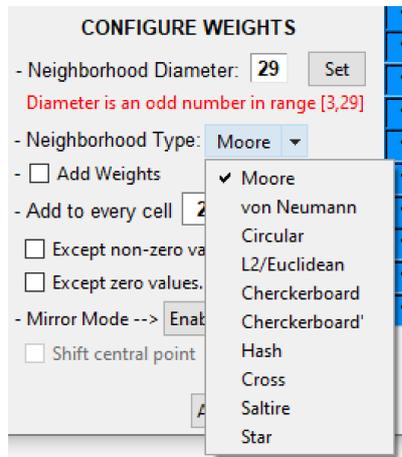


FIGURE 7.6: Drop-down menu of neighborhood types.

A constant value can be added to every entry (see the fourth dashed label in figure 7.3). The "-" and "+" button decrease or increase the value within the entry (entry with value "2") and the "Add" button; adds this value to the weights. Two exception rules may be applied, where the non-zero or zero values can be excluded from the "Add" operation. Using the latter exception rule, a neighborhood preserves its shape, for example, a von Neumann neighborhood maintains its diamond-like form, if the user wishes to increase or decrease the coefficients within the von Neumann region, as Figure 7.5 illustrates.

### 7.3.1 Neighborhood Types

The tool supports several neighborhood types, in other words, different patterns that can be drawn within the determined radius. The user can select the preferred type by clicking on a drop-down, where the supported ones are included, as Figure 7.6 showcases. Let  $(x, y)$  be the coordinates of an entry, with  $(x_0, y_0)$  representing the central point, and  $r$  denoting the radius:

$\forall x, y \in Z : -r \leq x \leq r \ \& \ -r \leq y \leq r$ , the following expressions must be true for drawing a:

- **von Neumann:**  $|x - x_0|^2 + |y - y_0|^2 \leq r$ .
- **Circular:**  $|x - x_0|^2 + |y - y_0|^2 \leq (r + 1)^2$ .
- **L2/Euclidean:**  $|x - x_0|^2 + |y - y_0|^2 \leq r^2$ .
- **Hash:**  $x = x_0 - 1 \vee x = x_0 + 1 \vee y = y_0 - 1 \vee y = y_0 + 1$ .
- **Cross:**  $x = x_0 \vee y = y_0$ .
- **Saltire:**  $y = x + 2 \times y_0 \vee y = -x$ .
- **Star:**  $y = x + 2 \times y_0 \vee y = -x \vee x = x_0 \vee y = y_0$ .

The  $x_0$  and  $y_0$  change their value, because the user may shift the central point (demonstrated in the next subsection), therefore, they are not invariably equal to  $(0,0)$  and must be encountered in the aforementioned expressions. The "Moore" type is just a square-shaped region that merely includes all the entries within the radius, so, there is no need to use a sophisticated expression. The "Checkerboard" is the complement of "Checkerboard", where a chessboard is simply drawn. If we perform a simple horizontal scan in the  $r \times r$  grid of entries, either the odd-positioned or the even-positioned ones are set to '0'. The following figures showcase all of these patterns, in case of having a  $11 \times 11$  area.

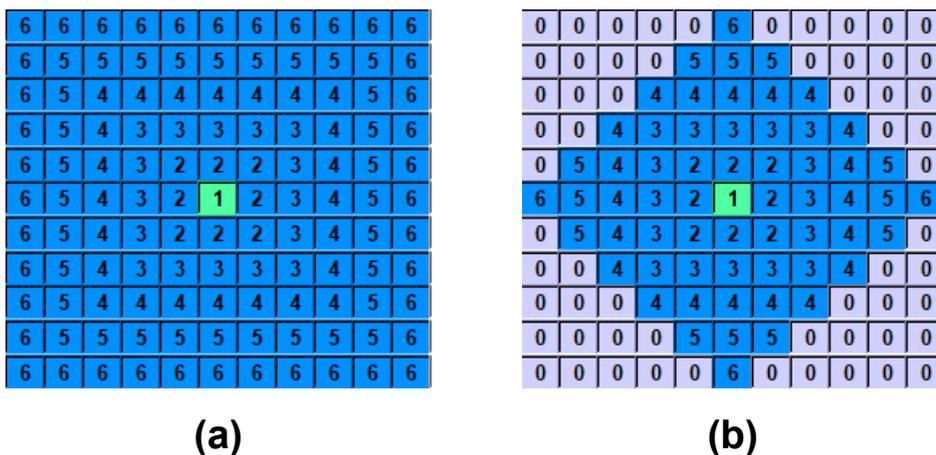


FIGURE 7.7: (a) Moore, (b) von Neumann.

0	0	0	0	6	6	6	0	0	0	0
0	0	5	5	5	5	5	5	5	0	0
0	5	4	4	4	4	4	4	4	5	0
0	5	4	3	3	3	3	3	4	5	0
6	5	4	3	2	2	2	3	4	5	6
6	5	4	3	2	1	2	3	4	5	6
6	5	4	3	2	2	2	3	4	5	6
0	5	4	3	3	3	3	3	4	5	0
0	5	4	4	4	4	4	4	4	5	0
0	0	5	5	5	5	5	5	5	0	0
0	0	0	0	6	6	6	0	0	0	0

(a)

0	0	0	0	0	6	0	0	0	0	0
0	0	5	5	5	5	5	5	5	0	0
0	5	4	4	4	4	4	4	4	5	0
0	5	4	3	3	3	3	3	4	5	0
0	5	4	3	2	2	2	3	4	5	0
6	5	4	3	2	1	2	3	4	5	6
0	5	4	3	2	2	2	3	4	5	0
0	5	4	3	3	3	3	3	4	5	0
0	5	4	4	4	4	4	4	4	5	0
0	0	5	5	5	5	5	5	5	0	0
0	0	0	0	0	6	0	0	0	0	0

(b)

FIGURE 7.8: (a) Circular, (b) L2/Euclidean.

0	6	0	6	0	6	0	6	0	6	0
6	0	5	0	5	0	5	0	5	0	6
0	5	0	4	0	4	0	4	0	5	0
6	0	4	0	3	0	3	0	4	0	6
0	5	0	3	0	2	0	3	0	5	0
6	0	4	0	2	0	2	0	4	0	6
0	5	0	3	0	2	0	3	0	5	0
6	0	4	0	3	0	3	0	4	0	6
0	5	0	4	0	4	0	4	0	5	0
6	0	5	0	5	0	5	0	5	0	6
0	6	0	6	0	6	0	6	0	6	0

(a)

6	0	6	0	6	0	6	0	6	0	6
0	5	0	5	0	5	0	5	0	5	0
6	0	4	0	4	0	4	0	4	0	6
0	5	0	3	0	3	0	3	0	5	0
6	0	4	0	2	0	2	0	4	0	6
0	5	0	3	0	1	0	3	0	5	0
6	0	4	0	2	0	2	0	4	0	6
0	5	0	3	0	3	0	3	0	5	0
6	0	4	0	4	0	4	0	4	0	6
0	5	0	5	0	5	0	5	0	5	0
6	0	6	0	6	0	6	0	6	0	6

(b)

FIGURE 7.9: (a) Checkboard, (b) Checkerboard'.

0	0	0	0	6	0	6	0	0	0	0
0	0	0	0	5	0	5	0	0	0	0
0	0	0	0	4	0	4	0	0	0	0
0	0	0	0	3	0	3	0	0	0	0
6	5	4	3	2	2	2	3	4	5	6
0	0	0	0	2	0	2	0	0	0	0
6	5	4	3	2	2	2	3	4	5	6
0	0	0	0	3	0	3	0	0	0	0
0	0	0	0	4	0	4	0	0	0	0
0	0	0	0	5	0	5	0	0	0	0
0	0	0	0	6	0	6	0	0	0	0

(a)

0	0	0	0	0	6	0	0	0	0	0
0	0	0	0	0	5	0	0	0	0	0
0	0	0	0	0	4	0	0	0	0	0
0	0	0	0	0	3	0	0	0	0	0
0	0	0	0	0	2	0	0	0	0	0
6	5	4	3	2	1	2	3	4	5	6
0	0	0	0	0	2	0	0	0	0	0
0	0	0	0	0	3	0	0	0	0	0
0	0	0	0	0	4	0	0	0	0	0
0	0	0	0	0	5	0	0	0	0	0
0	0	0	0	0	6	0	0	0	0	0

(b)

FIGURE 7.10: (a) Hash, (b) Cross.

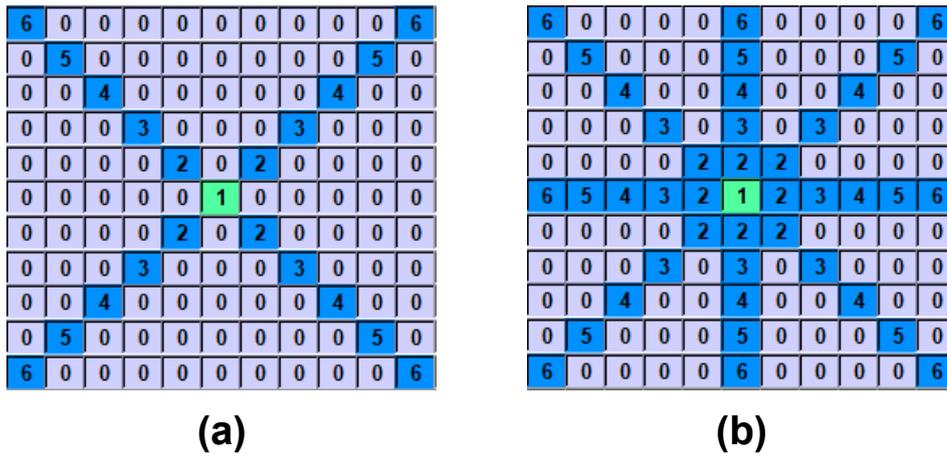


FIGURE 7.11: (a) Saltire, (b) Star.

### 7.3.2 Mirror Mode

Mirror mode is evidently the most valuable functionality for designating a neighborhood. When it is enabled, it restricts user's activity within the second quadrant. After the user has completed configuring these entries and presses the 'Done' button, they automatically get mirrored across the entire grid. Thus, in the worst case scenario of having 841 coefficients overall, only the uppermost, left,  $15 \times 15$  ( $= 225$ ) box has to be determined. Three distinct types of mirroring occur, involving the weights located in second quadrant with coordinates  $(x, y)$ . These mirroring operations include: horizontal reflection to the first quadrant over  $X$  axis ( $(-x, y)$ ), vertical reflection to the third quadrant over  $Y$  ( $(x, -y)$ ) and, central reflection to the fourth quadrant over  $(0, 0)$  point ( $(-x, -y)$ ).

Mirror mode is also compatible with the rest of the operations which have been already described. In simpler terms, the user can set weights, add a constant to every cell and configure one quarter of the selected neighborhood type. Additionally, the central point of the whole grid may be shifted to the center of the second quadrant, if and only if, an odd-dimensional box is shaped. Consequently, four sub-neighborhood can be included into the whole, if someone desires to do so. The following figures demonstrate the Mirror Mode's capabilities in a  $13 \times 13$  area, whenever or not the central point has been shifted.

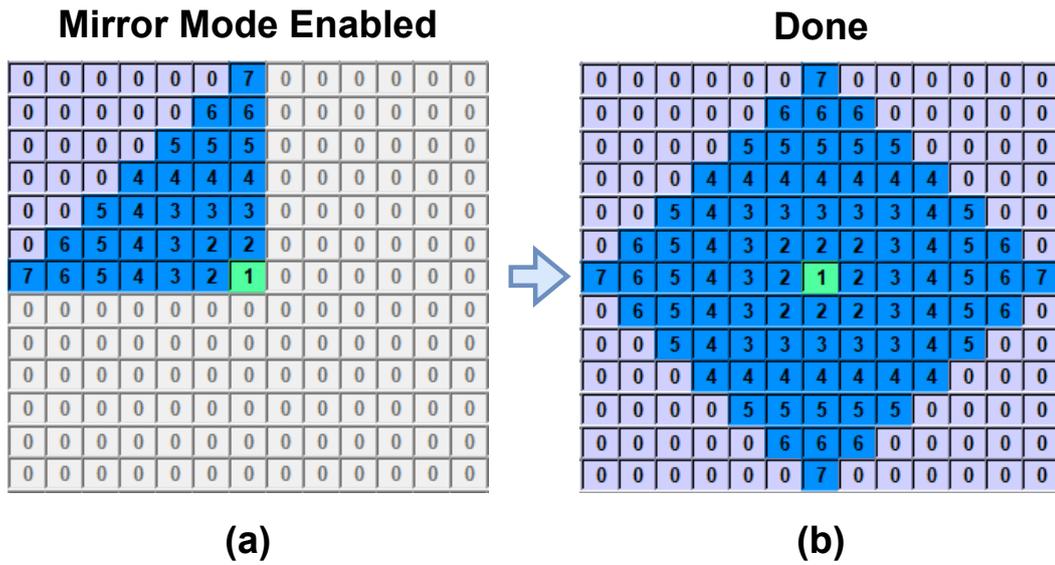


FIGURE 7.12: Mirror Mode: (a) The one fourth of a von Neumann neighborhood, (b) Properly mirrored.

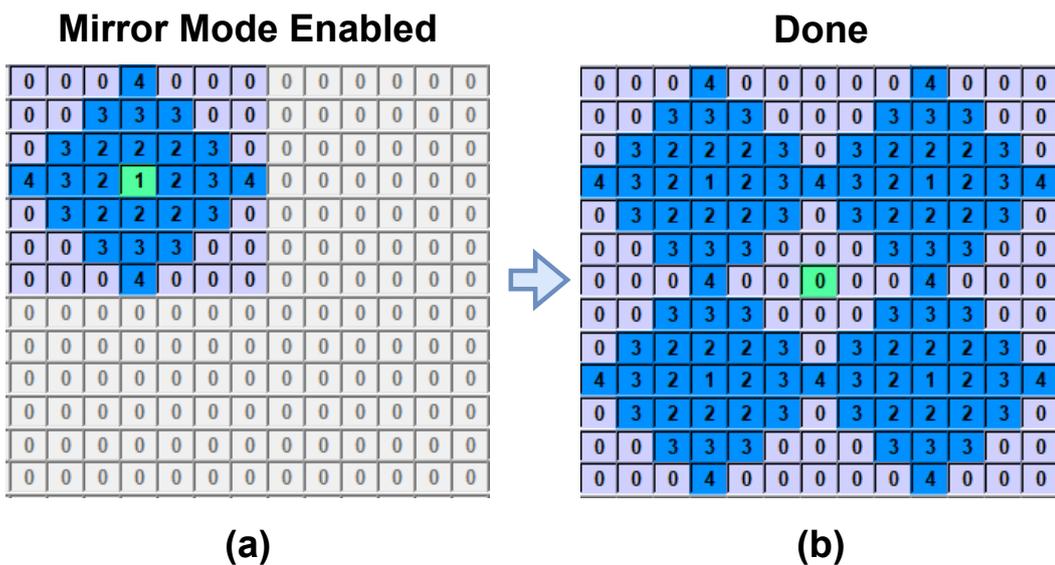


FIGURE 7.13: Mirror Mode with shifted center: (a) A small von Neumann within the second quadrant neighborhood, (b) Properly mirrored. Four von Neumann sub-neighborhoods were shaped.

## 7.4 The Remaining Configurations

The widgets represented in Figure 7.14 are placed beneath the Image Viewer. Through the *GRID PARAMETERS* section, the user can determine the number of states per cell, and the type of grid in which the CA evolves. As already said, the number of states supported are in range [2, 255] and the available

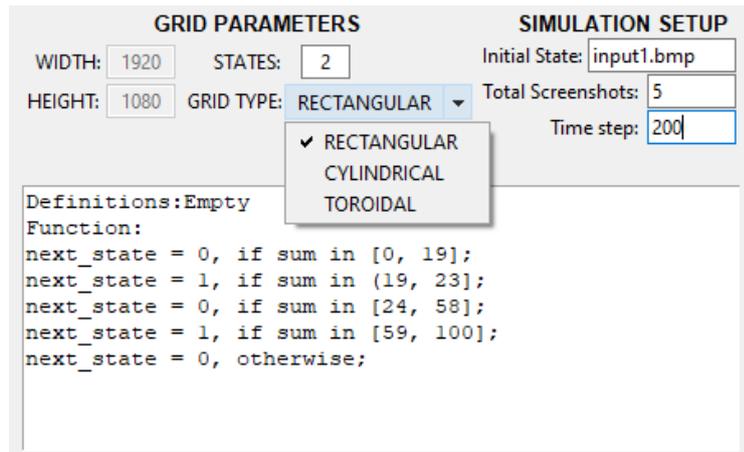


FIGURE 7.14: Widgets for configuring the grid, inserting the transition rule and setting up the simulation.

```

*** PLEASE DO NOT EDIT. This file is auto-generated by the tool. ***
States:2
Grid_Type:RECTANGULAR
# Transition Rule
Definitions:Empty
Function:
next_state = 0, if sum in [0, 19];
next_state = 1, if sum in (19, 23];
next_state = 0, if sum in [24, 58];
next_state = 1, if sum in [59, 100];
next_state = 0, otherwise;
_END_
NEIGHBORHOOD:3
0 2 0
2 1 2
0 2 0

```

FIGURE 7.15: An example of the format of a .config file.

grid types are: "Rectangular", "Cylindrical" and "Toroidal", meeting the hardware requirements. The grid size is always fix to  $1920 \times 1080$ , and the user can not modify it. It is displayed merely for clarification.

On the right hand-side, the initial state the simulation, the total amount of extractions, and a time step are specified. In the example of the figure 7.14, the simulation will run for 2000 generation in total, where 10 screenshots every 200 generation will be extracted. The lowest part is an embedded text editor, where the *Transition Rule* can be typed in, written in the CDL described in the previous chapter.

In section 7.2, we frequently referred to configuration file. In this file are

stored the number of state, the grid type, the transition rule, the neighborhood and its diameter. Figure 7.15 showcases the form of this file according to the present example (figure 7.14). Generally, this file should not be edited outside the tool, unless, the user is well-aware of how to preserve the format. It is a simple text file with only a different extension (.config), quite easy for someone to comprehend its structure.

Before running the simulation, the user should hit the "apply" button. If all of the user settings are valid and the hardware requirements are met, then, the simulation can get underway.

## Chapter 8

# System Verification, Examples of Use, Evaluation, and Results

Discovering a Cellular Automaton rule that meets a real-life application is a truly challenging task. Not only does it require a deep theoretical insight, but also many experiments have to be executed countless times with different configurations. These configurations may concern a combination of: parameterizing the transition rule, testing different neighborhood arrangements, or triggering the simulation with a variant of noisy initial states, comprised of statistical distributions. If the simulation manages to converge to a stable pattern, the goal has been reached.

As will be shown in this chapter, our framework works and it can drive the FPGA at the back end; the GUI also works and it has been used. Thus, the concept of creating a re-programmable framework has been proven, while a robust foundation has been established for future work. In this chapter, we will show how the *The Artificial Physics* and the *Game of Life* models, covering both cases of a totalistic and outer-totalistic rule are run in the system developed in the present thesis.

However, there exist limitations of our system as well, largely due to the constraint for non-conversant users to be able to use the system. The reason is that in Kyparissas' original work, the portion of the VHDL code that the user provides corresponds to a datapath which the Xilinx CAD tools create on a case-by-case basis. Hence, the limitations are only due to the FPGA resources. In our system, the user requirements (transition rules, weights, etc.) have to be mapped to an already developed architecture. The constraint to not use the Xilinx tools but use the already provided bit files restricts the range of applications vs. the original architecture. Even so, the applications which our system supports are by no means trivial.

Of the handful of models which were tested in Kyparissas' work, only the Artificial Physics and the Game of Life are fully supported in this first generation of our environment, as will be described in subsequent sections. These capabilities, however, are not at all trivial and a broad class of totalistic and outer totalistic CA can be modeled by our system.

In addition to the verification of our system, we conducted and will include in this chapter experiments of the *Hodgepodge Machine*, even though it is not supported by the CAD tool. This is a new research result, which gets into depth on some initial observations from Kyparissas' work. These observations were then called vortices. Physicists who study dynamical systems indicated that these are chimera states, which had not been observed before, as they require large CA neighborhoods. These results will be presented as well in the present chapter.

## 8.1 Artificial Physics

The *Artificial Physics* model is a totalistic CA rule, employing 2 states per cell and a weighted,  $21 \times 21$ , large neighborhood, where each cell of the grid can be either "dead" or "alive". The neighborhood of the model resembles concentric circles, while the initial state is consisted of randomly distributed alive cells in a ration of 1 to 7. The figure 8.1 depicts the neighborhood's window that was determined, while the transition rules of the model is defined as:

$$S_t(i, j) = \sum_{x=i-r}^{i+r} \sum_{y=j-r}^{j+r} w(x - i, y - j) \times c_t(x, y)$$

$$c_{t+1}(i, j) = \begin{cases} 0, & \text{if } S_t(i, j) \in [0, 19] \\ 1, & \text{if } S_t(i, j) \in (19, 23] \\ 0, & \text{if } S_t(i, j) \in (23, 58] \\ 1, & \text{if } S_t(i, j) \in (59, 100] \\ 0, & \text{otherwise} \end{cases}$$

Applying the above configuration, the simulation derives the results, depicted in the figure 8.2. As the simulation evolves, atoms are shaped in the automaton's universe. Those in close proximity bond together in a attempt to form molecules. Atoms cease to exist after a significant amount of runtime.



## 8.2 The Game Of Life

*The Game Of Life* was meticulously demonstrated in the Chapter 2. At the present section, we are going to represented screenshots extracted by the CAD tool. This models consists of an outer-totalistic rule and covers the second case of the supported models of the tool. The central state affect the condition of the transition, and as reminder, it is the case where the BRAM is divided into sub-internals. The figure 8.3 and 8.4 depicts, the derived results and the configuration withing the tool's environment respectively, while the initial state is the same to that of *Artificial Psysics'* example.

There are several well-known patterns which can be distinguished in the results, the most two popular ones are categorized as: *Still lifes* and *Oscillators*. The former refers to patterns that remain unchanged as the simulation evolving, while the latter to those that return to their initial state, after a finite number of generations. To name but a few, the *Block* and *Beehive* are involved in the first category, while the *Blinker* in the second one. For example, the *Block* resembles a 2 by 2 box of alive cells, hence each cell within the block has exactly 3 neighbors and thus its shape is preserved. On the other hand, the *Blinker* forms a straight line of three alive cells. The central cell has exactly two neighbors and remains alive throughout time, while the others "die" due to under-population. Furthermore, the two cells on either side of the central cell resurrect, given the they have three neighbors. So the line is being rotated by 90° generation by generation, composing an oscillation with period of 2.

Both of the above examples were used to verify our system in every respect: a new architecture that works (with all that it entails in terms of being "shrink-wrapped" and not re-compiled for each model, driven by the back end tool, and with rules and neighborhoods developed by the new tools). We do not report performance, as the present architecture matches that of the original, and (like the original) it is the same regardless of rule and neighborhood complexity. Hence, the system verification entailed a replication of known results (as testbenches) rather than performance evaluation.

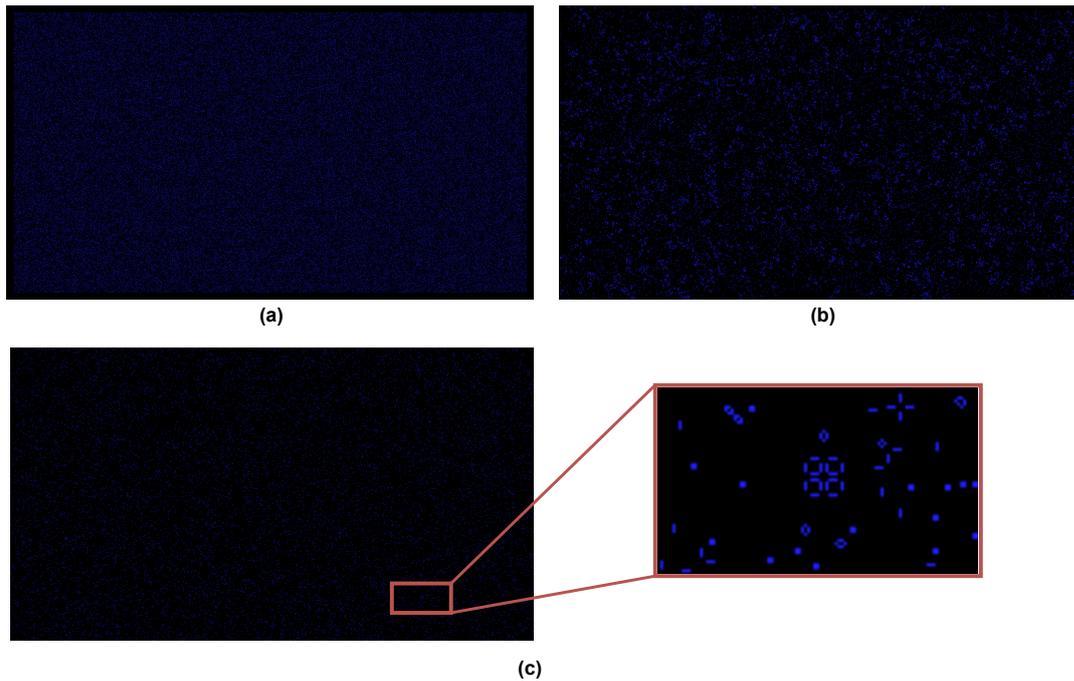


FIGURE 8.3: The evolution of *Game of Life*. (a) Initial State, (b) 500 generations, and (c), 15,000 generations along with a zoomed-in frame.

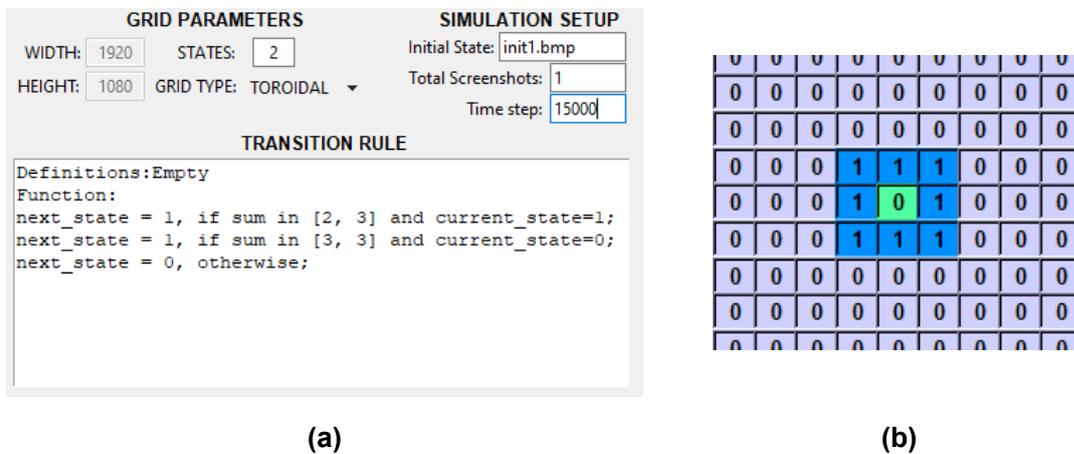


FIGURE 8.4: *Game of Life* in-GUI configuration. (a) Parameters/Transition Function (b) Neighborhood window.

## 8.3 The Hodgepodge Machine

The Hodgepodge Machine isn't yet compatible with the re-programmable framework, and by extension, it is not yet supported by the CAD tool. Even so, many experiments were conducted using the initial design, because the "vortices" which were observed in Kyparissas' work were what physicists call "chimera states" and these are observed for the first time in the specific model. In the next chapter, we suggest a potential re-programmable architecture to

support this model. In this section, the experiments and the conclusions that was performed and inferred follow.

### 8.3.1 Experiments

The Hodgepodge Machine was created by Martin Gerhardt and Heike Schuster of the University of Bielefeld in West Germany. This CA problem which models a special case of the Belousov-Zhabotinsky reaction is defined according to the following cell state transition function:

$$c_{t+1}(i,j) = \begin{cases} \frac{K_t(i,j)+I_t(i,j)}{k} & , \text{ if } c_t(i,j) = 0 \\ 0 & , \text{ if } c_t(i,j) = q \\ \frac{\text{sum of all neighbors}}{I_t(i,j)} + g & , \text{ otherwise} \end{cases}$$

Before proceeding to the presentation of our new results, we should point out the reason why the framework does not support this model at present. As can be seen in the transition function, above, the state is *not* only a function of the inner product of the window and the weights, but it requires more complex operations, plus divisions. The present "shrink wrapped" architecture (in all of its six versions for grid type and weight/state size) does not have the ability to do generalized arithmetic, whereas code written in VHDL can model such complex operations because the associated datapath is created by the Xilinx Vivado CAD tools.

Resuming with the model under study and its transition function, if the number of states are  $q + 1$ , then the values of each cell are in range  $[0, q]$ . The state given by the value 0 is called "healthy", the state  $q$  is said to be "ill" and all other values in range  $[1, q - 1]$  describe a degree of infection, the higher the value, the more infected the cell is. The variables  $K_t(i,j)$  and  $I_t(i,j)$  correspond to the number of "ill" and "infected" neighbors respectively. An infected cell should gradually approach the ill state. The constant variable  $g$  determines the growth rate of infection, while the constant  $k$  is called the "weighting" parameter for healthy cells and determines the intensity of the infection process. [40, 41]

The parameters of Hodgepodge Machine used are:  $k = 5$ ,  $g = 105$  and  $q = 255$ , using a  $1920 \times 1080$  toroidal grid. The initial state of the machine is set to noisy images that were generated with a random number generator, by using a specific seed as an indicator to the produced streams. Two random sequences were derived of total length  $1920 \times 1080$  each, where the one

corresponds to Bernoulli's random variable - employing a probability  $p$  of zeros and  $1 - p$  of ones - while the other accounts for a uniform distribution in range  $[1, 255]$ . Thus, the input image of the machine is constituted by the former sequence, in which the ones have been replaced by the respective cell values of the latter. In other words, an initial state contains  $p$  percent number of zeros and  $1 - p$  percent, uniformly, distributed values in range  $[1, 255]$ . The results of our machine - which utilizes the aforementioned configurations - are represented in the following figures at the end of this section.

Figure 8.5 shows mid-sized neighborhoods. The images (a) and (b) account for neighborhood sizes of  $15 \times 15$  and  $17 \times 17$  correspondingly. In the  $15 \times 15$  neighborhood, the Bernoulli variable is  $p = 47\%$ , while for the  $17 \times 17$  it is  $p = 50\%$ . Consequently, it is observed that equal or larger than  $17 \times 17$  neighborhood sizes are needed in order for the chimera states to occur, where the neighborhood of size  $17 \times 17$  is the starting point.

The remaining figures follow a similar pattern. Initially, they feature different sizes of large neighborhoods such as:  $21 \times 21$ ,  $25 \times 25$  and  $29 \times 29$  (Fig.8.6, Fig.8.7 and Fig.8.8 respectively). The first image (a) shows the initial state of the machine, the second one (b) shows the first appearance of an at least one chimera state, and the third (c) illustrates a converged state of the machine after 200 generations, alongside a zoomed-in-picture on a chimera state. Regarding the Bernoulli variable  $p$  from which the initial state is formed,  $p = 51\%$ ,  $p = 54\%$  and  $p = 57\%$  correspond to neighborhood sizes of  $21 \times 21$ ,  $25 \times 25$  and  $27 \times 27$ .

The Bernoulli  $p$  value not only plays a major role in terms of convergence, but also, its range of generating a valid initial state is extended as the neighborhood is increasing. This means that the larger the neighborhood size is, the wider is the range of chimera state observation as the  $p$  value grows. Table 8.1 shows the ranges of valid Bernoulli  $p$  values and the associated neighborhood sizes. These values of  $p$  refer to initial states, with which the Hodgepodge Machine CA managed properly converged *and* chimera states appeared. It should be noted that based on  $p$  the Hodgepodge machine may or may not converge, and our study shows clearly that larger CA neighborhoods have a direct impact to the increase of  $p$  values that lead to convergence and chimera states, whereas smaller neighborhoods may lead to convergence but without chimera states (e.g. up to  $15 \times 15$  neighborhoods do not lead to chimera states even after 200 iterations).

Neighborhood Size	Range of $p$	Total Length of range
$19 \times 19$	[49, 53]	5
$21 \times 21$	[49, 53]	5
$23 \times 23$	[50, 54]	5
$25 \times 25$	[52, 58]	7
$27 \times 27$	[49, 59]	11
$29 \times 29$	[51, 62]	12

TABLE 8.1: Valid values of  $p$  in relation to neighborhood size.

It should be mentioned that, the experiments were conducted by using up to twenty different initial states or seeds for each value of  $p$  in the range  $[48\%, 63\%] \in \mathbb{Z}$ , hence, the ranges in the table 8.1 might be wider as well, if the experiment is conducted with even more seeds. Nevertheless, the above statement regarding the connection between the range of  $p$  and the neighborhood size is not invalidated, because it is important to have inputs which lead to the observation of the desired phenomena after fewer iterations of the system. Furthermore, the ranges are shift towards larger values of  $p$  as the neighborhood size increases. A possible explanation to this phenomenon could be that given that the parameters of the machine remain identical for every single execution of the CA, and so does the initial state, then, the total sum in a larger area of the grid will be equal to or greater than the total sum in a smaller one. Thus, in order for the CA to maintain a similar behaviour as the neighborhood size is growing, a higher number of zeros needs to be initially placed in the grid. Last but not least, for the mean values of  $p$  in the valid ranges, the machine converges faster and more accurately, and the opposite behaviour is observed towards the ends of the window. Therefore, the ideal input appears to be at the central value of the range, as far as noisy initial states are concerned.

To conclude with this experiment, we note that the large neighborhood CA can reveal complexities which smaller neighborhoods cannot show, despite both being Turing complete. The reason is that in order for the smaller models to reach such results, the amount of compute time that would be needed is inordinate, much like a Turing machine cannot effectively (from a performance point of view) model a weather prediction model.

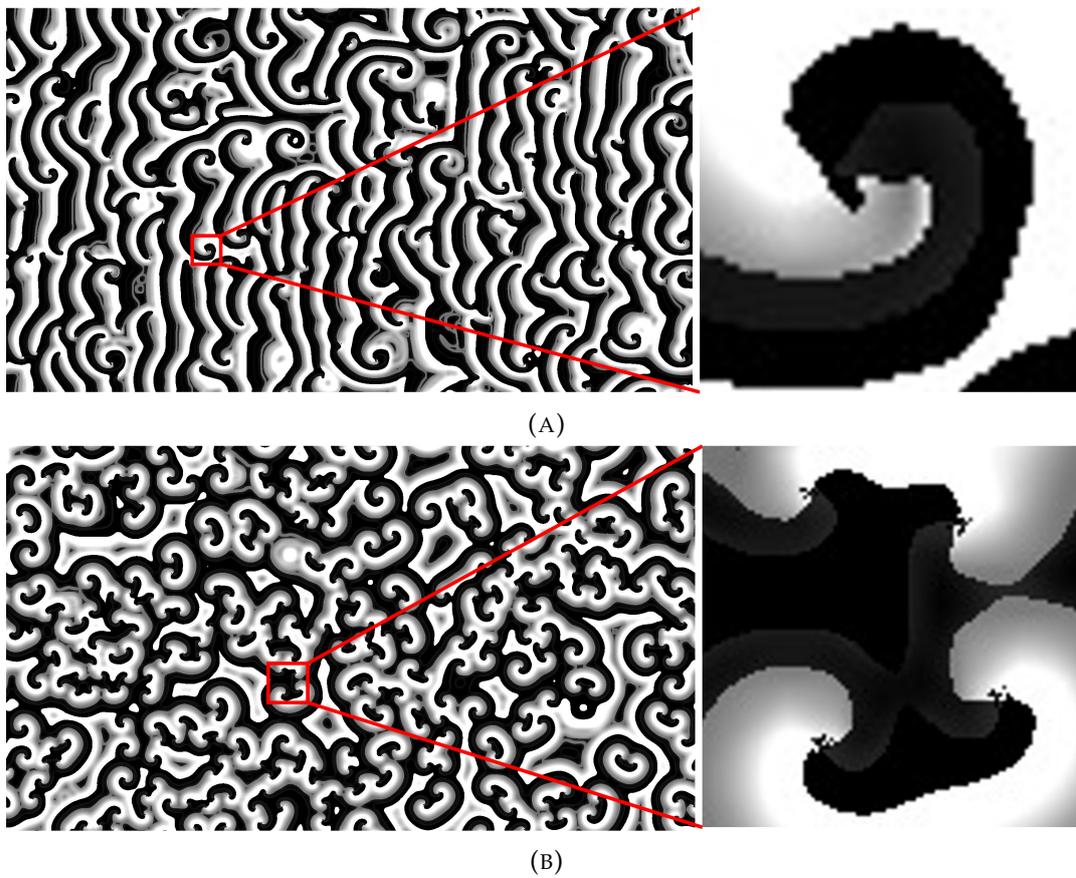


FIGURE 8.5: (a)  $15 \times 15$  and (b)  $17 \times 17$  after the system has converged.

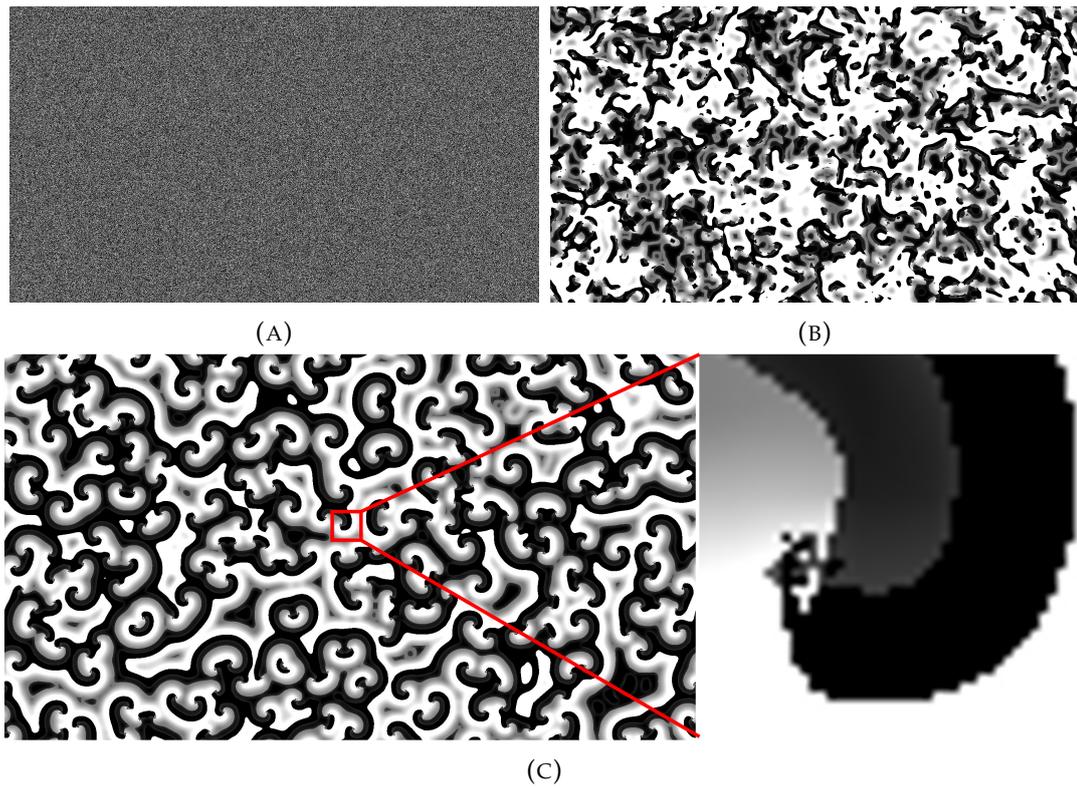


FIGURE 8.6:  $21 \times 21$  neighborhood: (a) Initial State, (b) 19 and (c) 200 generations.

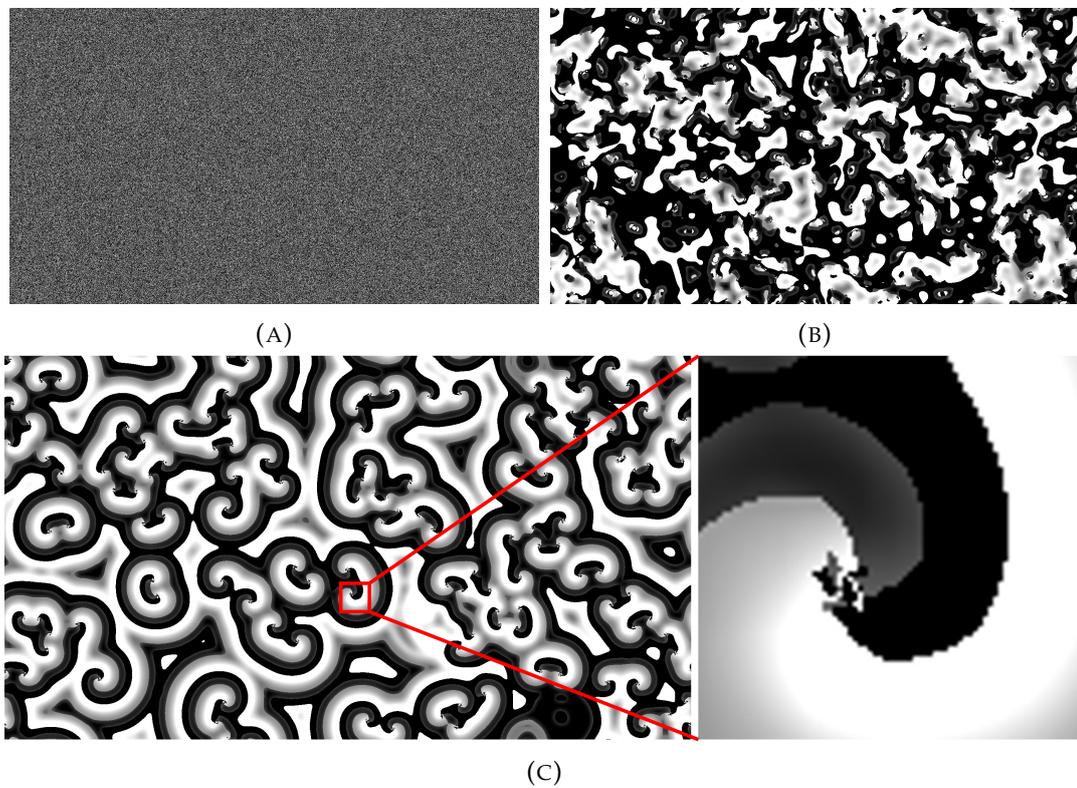


FIGURE 8.7:  $25 \times 25$  neighborhood: (a) Initial State, (b) 22 and (c) 200 generations.

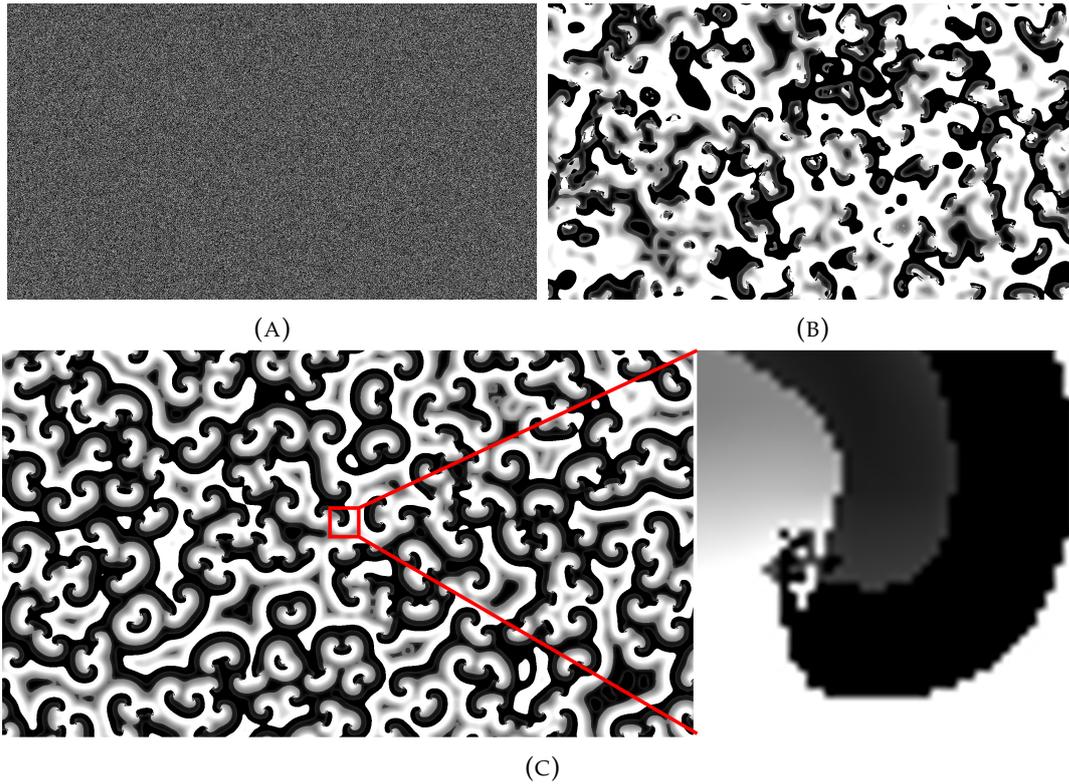


FIGURE 8.8:  $29 \times 29$  neighborhood: (a) Initial State, (b) 40 and (c) 200 generations.

## 8.4 Discussion

In this Chapter we have shown that all goals of the present thesis were met: the baseline architecture was changed, so that all of the previously supported grids and weight/state size options are supported. The new architecture is accessible through the new CAD tool and GUI, and as long as the transition rule is a map of the dot product (and the present state, if the CA is outer totalistic) to the next state subject to any arbitrary range (i.e. the next state can be, e.g. 0 for all odd values of the dot product and 1 for the even values). The new system was tested and it operates reliably (as best we can tell), and it is suitable to be ported to the Amazon AWS FPGA cloud services. Limitations of the present system stemming from complex rules requiring custom datapath are not yet supported, but will be discussed in the following chapter.



## Chapter 9

# Conclusions and Future Work

In this last chapter, we are going to summarize the current work and present potential directions for further development.

### 9.1 Conclusions

In this thesis we managed convert the initial framework of the hardware architecture into an as-was, re-programmable system. The system's initialization process is now fully automated, almost providing a plug-an-play experience to the end-user. Our machine is not anymore limited to hardware engineers exclusively, while, no valuable time is being wasted, waiting for the circuit to be generated. The GUI environment further increases convenience for configuring CA models, adjusting large neighborhood windows, organizing user's work and extracting results. The CDL, that have been developed, appears to be useful and practical for inserting transition rules, while its compiler interprets the user's input into a processable data structure. With the use of *Protocol Buffers*, the inserted parameters are serialized into a specifically formatted structure, rendering their allocation possible to the appropriate hardware components. Consequently, a basic core has been established for constructing a unified, general-purpose, Cellular Automata accelerator.

### 9.2 Future Work

At present, our CAD tool only support a limited amount of CA models. According to the most general form of the supported transition rule (see chapter 6, section 6.3), it can be said that, merely static, outer-totalistic rules can be modeled in our machine. To clarify term "*static*", the conditions and

the next state of the transition rules are decided by constant numbers, while other *Automata* reveal a more dynamic behavior. In our case, only the total sum exhibits such behavior, given that it varies for neighborhood windows.

There is a handful of applications that Kyparissas tested and verified the hardware architecture and it would be of great use to model these as well in the re-programmable framework.

### 9.2.1 Application Examples

In this particular subsection, we aim to demonstrate several well-known applications, that Kyparissas successfully simulated, and it is a dire necessity to be embedded in the re-programmable architecture. Moreover, we are going to propose one possible approach for these applications. Additionally, *the Hodgspode Machine* is also included, even if it is not represented in the upcoming examples, since it was analytically described in the previous chapter.

#### The Greenberg-Hastings

The Greenberg-Hastings model is a non-linear, dynamical system, also characterized as an excitable media [42]. Originally, it consisted of  $3 \times 3$  von Neumann neighborhood and 3 states per cell: "quiescent", "excited" and "refractory", and it was further enlarged to support more cell states and larger neighborhood windows afterwards [43, 44]. Kyparissas expanded this model to simulate it, using a  $29 \times 29$  von Neumann neighborhood and 16 states per cell. A cell can be "quiescent" (state 0), "excited" (state 1) or in a sequence of "refractory" (state 2 to 15), where the next state is determined according to the following transition rule:

$$c_{t+1}(i,j) = \begin{cases} 1, & \text{if } c_t(i,j) = 0 \text{ \& the total excited neighbors } > T, \\ & \text{where } T \text{ is a threshold value} \\ c_t(i,j) + 1, & \text{if } c_t(i,j) > 0 \\ c_t(i,j), & \text{otherwise} \end{cases}$$

The next state of a cell is decided dynamically as a function of the central one, while instead of calculating the total sum of the neighborhood, it is only required to count the occurrences of excited cells within the window. The above transition rule could be expressed in our CDL as:

Definitions :

States: 0–quiescent | 1–excited | others–refraction ;

T:4;

Function :

next\_state = 1, if central\_cell=0 and total\_excited>T;

next\_state = current\_state+1, if central\_cell >0;

next\_state = current\_state , otherwise ;

### Anisotropic Rules

In *Physics*, anisotropy is observed when a physical property of an object alters its behavior in a directionally dependent manner. A good and simple example appears to be a ruler. It feels stronger (as a structure) when a force is exerted from its edge towards the center, while significantly weaker, when the same magnitude of force is applied vertically to its surface. In a Cellular Automaton's world, this property of non-uniformity can emerge, should we define the proper transition rule.

In this example, the anisotropic rules utilizes a weighted,  $29 \times 29$ , Moore neighborhood and 256 states per cell. The central weight is equal to 15, and the weights are gradually being reduced by 1, towards the edges of the neighborhood window. The transition rule is defined as follows:

$$S_t(i,j) = \sum_{x=i-r}^{i+r} \sum_{y=j-r}^{j+r} w(x-i, y-j) \times c_t(x,y)$$

$$c_{t+1}(i,j) = \begin{cases} c_t(i,j) - 1, & \text{if } S_t(i,j) = 0 > \text{threshold} \\ c_t(i,j) + 1, & \text{if } S_t(i,j) = 0 < \text{threshold} \\ c_t(i,j), & \text{otherwise} \end{cases}$$

Using our CDL, it could be written as:

Definitions :

T:128;

Function :

next\_state = current\_state -1, if sum > T;

next\_state = current\_state+1, if sum < T;

next\_state = current\_state , otherwise ;

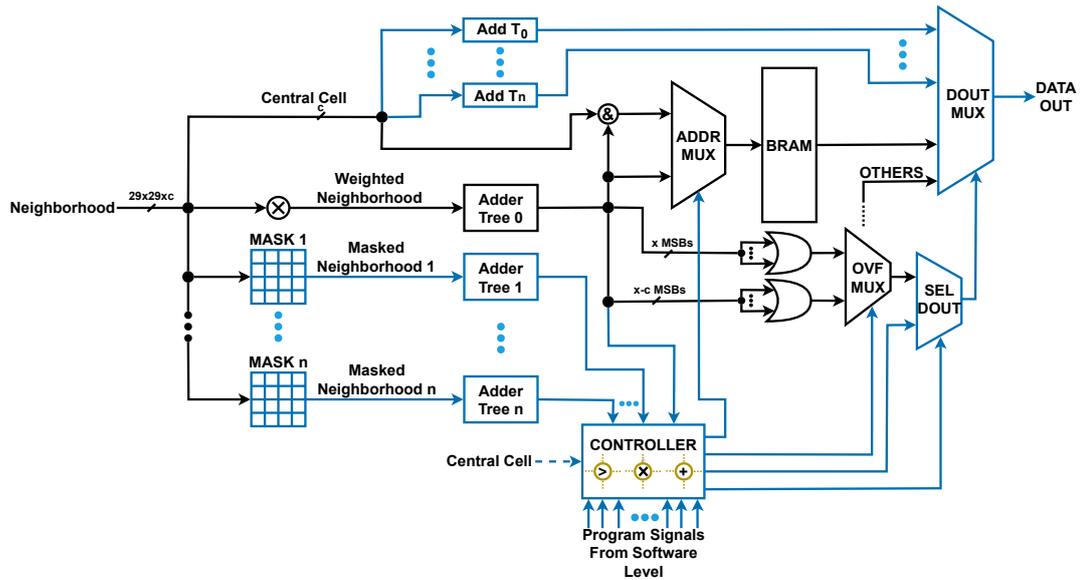


FIGURE 9.1: CA Engine's extended, re-programmable framework for supporting more complex models. A potential solution (blue color) attached to the already developed hardware (with black).

## Hardware Approach

To expand the re-programmable framework in order to support the above models, further logic and arithmetic capability needs to be built-in. Figure 9.1 showcases a possible scenario of the potential architecture, including additional components which it calls for. The more components that will be available within the circuit, the broader the range of supported models will be.

Explaining the extended, proposed, re-programmable framework from left to right, firstly, the neighborhood is being mask according to the desired cell state to be counted. Many models require to count the occurrences of a certain cell state. So, the masked neighborhood will be comprised of ones and zeros only, where the ones have replaced the desired cell state, and all other become zero. Thus, feeding the masked neighborhood into a binary adder tree component, the total number of a specific cell state is calculated. The total sum is calculated as usually, while, the more binary adder trees are placed in the design, the greater the number of cell state occurrences can be computed.

Additionally, the central cell is driven to adders, where the threshold values are applied. The increased or decreased current cell states are connected

to the data out multiplexer. In this manner, the design is capable of generating the next state as a function of the central cell. To even further enhance the capabilities of the design, multipliers and dividers can be placed alongside the adders, creating a general "OPERATOR" module, in order to apply this operations to the central cell. This demands extra wires coming from the controller, so as to select an operation properly. Although, it is not depicted in the schematic to maintain it as readable as possible.

As for the *Controller*, it accepts the results provided from the binary adder trees and an amount of program signals from software level. The *Controller* will execute a series of logical and arithmetic operations, like an *ALU*, while the program signals will implement the appropriate wiring in it. This module is identical to the condition of a mathematical branch function, it computes it and generates the select signals of the multiplexers accordingly. These series of operations may be applied to the results provided from the adder trees, to the central cell, or, in different combinations among them. Consequently, the *CA Engine* could be capable of calculating even more complex conditions of transition functions.

Regarding the multiplexing logic, the *ADDR MUX* and the *OVF MUX* would operate in a similar way as already have been developed. The addition of the *SEL DOUT* multiplexer is truly significant at this point. Let's assume that, a hypothetical rule has two cases, where, the one only depends on the total sum of the neighborhood, so the corresponding next state is stored in the *BRAM*, and, the other relies on a conditional, apart from the total sum, where the next state is decided by an operation with the central cell. If the total sum overflows, the current design will provide as output the otherwise value, which is incorrect. Instead, the *Controller* would generate the appropriate "select data out" signal. This signal will be driven to the select of *DOUT MUX* and the *OVF* signal produced would be omitted, via the the *SEL DOUT* multiplexer.

Ultimately, should this approach is eventually followed, the CAD tool has to be extended as well. The compiler of CDL will be further expanded to support even more complex transition rules as input. In addition, a rather sophisticated algorithm will be required to approximate transition functions and generate the proper values for the *BRAM*. Finally, a scrupulous premeditation is necessitated for inventing a dexterous format of program signals. Putting them all together, it can pave the way for producing implausible simulations in the universe of *Cellular Automata*.

### 9.2.2 Globalizing Accessibility and Improve Experience

The next step of the present CAD tool is to be utilized by multiple scientists all-around the globe. With today's technology, this is easily feasible by means of cloud services. More specifically, *Amazon* corporations provides accessibility on FPGA platforms remotely, addressing to both developers and clients. At this particular moment, the *Technical University of Crete* has access to *Amazon's* services, meaning that we are on the verge of fulfilling it.

The *Amazon* cloud offers access to state-of-the-art FPGAs, namely, the *Amazon EC2 F1* instance is equipped with a *Virtex UltraScale+ VU9P* FPGA board of TSMC's 16nm FinFET. In comparison to our FPGA platform, *VU9P* reaches a number of 2,586,150 system logic cells, where our board only has 101,440. This enormous difference of capacity unleashes our flexibility in order to achieve tremendous performances. All of the aforementioned applications could be indubitably fit in such scale, and furthermore, the range of supported weights, the number of cell size, the grid size and the neighborhood window could also be broadened.

By the time the simulator is online, the experience of the end-user can also be improved in several ways. Firstly, instead of exporting images one by one, the simulation could be displayed as a live stream in real time. A PCIe fabric allows the FPGA to share its memory space at up to 12 gigabytes per second bidirectionally. At a frame rate of 60FPS for a  $1920 \times 1080$  resolution, we need to transfer  $1920 \times 1080 \times 60 \times 8$  bits = 995,328 Mbits per second, where with the given bandwidth is doable.

Additionally, the *Graphics Controller* and the *UART Controller* will be removed, releasing even more resources. The *Graphics Controller* is not required any more. On the cloud environment, the values inside the memory will be received generation per generation, and the video will be produced frame by frame. Moreover, *UART* is a sluggish protocol. While the design produces 60 generations in one second, the tool demands for at least 10 seconds to capture them, owing to the slow rate of transmitting and receiving data during both initialization and extractions processes.

Furthermore, given that the live streaming function has been embedded, the *Speed Controller* could alter the speed of simulation (as it was) or pause it for manual extraction. By using, for example, the polling technique, interrupt signals of the keyboard could be captured by the tool and transmitted to FPGA. In this regard, the user could control the speed of the FPGA remotely.

Last but not least, an image generator and an embedded drawing tool are also consist of a useful addition, where statistical distributions, such as Normal, Binomial, Poisson, Gaussian, etc, could be utilized for deriving initial states.



## References

- [1] Stanislaw Ulam. "Random Processes and Transformations". In: *International Congress of Mathematicians*. Cambridge, 1950.
- [2] John von Neumann. "The General and Logical Theory of Automata". In: *Cerebral Mechanisms in Behavior: The Hixon Symposium*, John Wiley & Sons (1951).
- [3] James B. Salem and Stephen Wolfram. "Thermodynamics and Hydrodynamics with Cellular Automata". In: *Theory and Applications of Cellular Automata*, World Scientific (1986), p. 5.
- [4] Daniel H. Rothman and Stephane Zaleski. *Lattice-Gas Cellular Automata: Simple Models of Complex Hydrodynamics*. Cambridge University Press, 2004.
- [5] Paulien Hogeweg. "Cellular Automata as a Paradigm for Ecological Modeling". In: *Applied Mathematics and Computation* 27.1 (1988), pp. 81–100.
- [6] Felix A. Gers, Hugo de Garis, and Michael Korkin. "CoDi-1Bit : A Simplified Cellular Automata Based Neuron Model". In: *Lecture Notes in Computer Science* 1363 (1998), pp. 315–333.
- [7] Konrad Zuse. *Calculating Space*. MIT Technical Translation AZT-70-164-GEMIT, Massachusetts Institute of Technology (Project MAC), 1970.
- [8] John von Neumann and Arthur W. Burks. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.
- [9] Arthur W. Burks. *Essays on Cellular Automata*. University of Illinois Press, 1971.
- [10] Tommaso Toffoli. "Computation and Construction Universality of Reversible Cellular Automata". In: *Journal of Computer and System Sciences* 15.2 (1977), pp. 213–231.
- [11] Melanie Mitchell. "Computation in Cellular Automata: a Selected Review". In: *Non-Standard Computation*, Wiley-VCH Verlag in Weinheim (1998), pp. 95–140.

- [13] Nikolaos Kyparissas and Apostolos Dollas. “An FPGA-Based Architecture to Simulate Cellular Automata with Large Neighborhoods in Real Time”. In: 14.1 (2020), pp. 1–32. DOI: [10.1145/3423185](https://doi.org/10.1145/3423185).
- [14] Nikolaos Kyparissas and Apostolos Dollas. “An FPGA-Based Architecture to Simulate Cellular Automata with Large Neighborhoods in Real Time”. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 95–99. DOI: [10.1109/FPL.2019.00024](https://doi.org/10.1109/FPL.2019.00024).
- [15] Nikolaos Kyparissas and Apostolos Dollas. “Field Programmable Gate Array Technology as an Enabling Tool Towards Large-Neighborhood Cellular Automata on Cells with Many States”. In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. 2019, pp. 940–947. DOI: [10.1109/HPCS48598.2019.9188084](https://doi.org/10.1109/HPCS48598.2019.9188084).
- [18] Andrew Ilachinski. *Cellular Automata: a Discrete Universe*. World Scientific, 2001.
- [19] Martin Gardner. “Mathematical Games - The Fantastic Combinations of John Conway’s New Solitaire Game “Life””. In: *Scientific American* 223.4 (1970).
- [20] Paul W. Rendell. “A Universal Turing Machine in Conway’s Game of Life”. In: *2011 International Conference on High Performance Computing and Simulation*. Istanbul, Turkey, 2011, pp. 764–772.
- [21] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays*. 2nd ed. A K Peters Ltd., 2001.
- [22] Tommaso Toffoli. “CAM: A High-Performance Cellular-Automaton Machine”. In: *Physica D: Nonlinear Phenomena* 10.1-2 (1984).
- [23] Tommaso Toffoli and Norman H. Margolus. *Cellular Automata Machines - A New Environment for Modeling*. MIT Press, 1987.
- [24] Norman H. Margolus. “CAM-8: A Computer Architecture Based on Cellular Automata”. In: *Pattern Formation and Lattice-Gas Automata*, AMS (1993).
- [25] Norman H. Margolus. “An FPGA Architecture for DRAM-Based Systolic Computations”. In: *5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa Valley, CA, USA, 1997, pp. 2–11.
- [26] Norman H. Margolus. “An Embedded DRAM Architecture for Large-Scale Spatial-Lattice Computations”. In: *27th Annual International Symposium on Computer Architecture*. Vancouver, Canada, 2000, pp. 149–160.
- [27] Rolf Hoffmann, Klaus-Peter Völkman, and Mark Sobolewski. “The Cellular Processing Machine CEPRA-8L”. In: *Mathematical Research* 81 (1994), pp. 179–188.

- [28] Christian Hochberger et al. "The CEPRA-1X Cellular Processor". In: *Reconfigurable Architectures: High Performance by Configware*, IT Press, Bruchsal (1997).
- [29] Christian Hochberger et al. "The Cellular Processor Architecture CEPRA-1X and its Configuration by CDL". In: *IPDPS 2000. Lecture Notes in Computer Science, vol 1800*. 2000, pp. 898–905.
- [30] Paul Shaw, Paul Cockshott, and Peter Barrie. "Implementation of Lattice Gases Using FPGAs". In: *Physica D: Nonlinear Phenomena* 12.1 (1996), pp. 51–66.
- [31] Tomoyoshi Kobori, Tsutomu Maruyama, and Tsutomu Hoshino. "A Cellular Automata System with FPGA". In: *9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Rohnert Park, CA, USA, 2001, pp. 120–129.
- [32] Chase Phelps and Tanzima Islam. "Automatic Parallelization of Cellular Automata for Heterogeneous Platforms". In: *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2023, pp. 352–361. DOI: [10.1109/COMPSAC57700.2023.00055](https://doi.org/10.1109/COMPSAC57700.2023.00055).
- [33] K. Bouazza et al. "Implementing Cellular Automata on the ArMen Machine". In: *2nd International Workshop on Algorithms and Parallel VLSI Architectures*. Gers, France, 1992, pp. 317–322.
- [34] Gregorio Cappuccino and Giuseppe Cocorullo. "Custom Reconfigurable Computing Machine for High Performance Cellular Automata Processing". In: *Electronic Engineering Times (www.eetimes.com, TechOnLine Publication)* (2001).
- [35] Shakeeb Murtaza, Alfons G. Hoekstra, and Peter M. A. Sloom. "Performance Modeling of 2D Cellular Automata on FPGA". In: *2007 International Conference on Field Programmable Logic and Applications*. Amsterdam, The Netherlands, 2007, pp. 74–78.
- [36] Shakeeb Murtaza, Alfons G. Hoekstra, and Peter M. A. Sloom. "Floating Point Based Cellular Automata Simulations Using a Dual FPGA-Enabled System". In: Austin, TX, USA, 2008, pp. 1–8.
- [37] Shakeeb Murtaza, Alfons G. Hoekstra, and Peter M. A. Sloom. "Compute Bound and I/O Bound Cellular Automata Simulations on FPGA Logic". In: *ACM Transactions on Reconfigurable Technology and Systems* 1.4 (2009), p. 23.
- [38] Shakeeb Murtaza, Alfons G. Hoekstra, and Peter M. A. Sloom. "Cellular Automata Simulations on a FPGA Cluster". In: *International Journal of*

- High Performance Computing Applications*, SAGE 25.2 (2010), pp. 193–204.
- [39] André C. Lima and João Canas Ferreira. “Automatic Generation of Cellular Automata on FPGA”. In: *9th Portuguese Meeting on Reconfigurable Systems*. Coimbra, Portugal, 2013, pp. 51–58.
- [40] Alexander K. Dewdney. “Computer Recreations: The Hodgepodge Machine Makes Waves”. In: *Scientific American* 259.2 (1988).
- [41] Martin Gerhardt and Heike Schuster. “A Cellular Automaton Describing the Formation of Spatially Ordered Structures in Chemical Systems”. In: *Physica D: Nonlinear Phenomena* 36.3 (1989).
- [42] James M. Greenberg and Stuart P. Hastings. “Spatial Patterns for Discrete Models of Diffusion in Excitable Media”. In: *SIAM Journal on Applied Mathematics* 54 (1978), pp. 515–523.
- [43] Robert Fisch, Janko Gravner, and David Griffeath. “Threshold-Range Scaling of Excitable Cellular Automata”. In: *Statistics and Computing* 1 (1991), pp. 23–39.
- [44] Richard Durrett and David Griffeath. “Asymptotic Behavior of Excitable Cellular Automata”. In: *Experimental Mathematics* 2.3 (1993), pp. 183–208.

## External Links

- [16] “Moore Neighborhood”. In: (). URL: <https://mathworld.wolfram.com/MooreNeighborhood.html>.
- [17] “von Neumann Neighborhood”. In: (). URL: <https://mathworld.wolfram.com/vonNeumannNeighborhood.html>.