A Framework for the Real-Time Execution of Cellular Automata on Reconfigurable Logic

By

Nikolaos Kyparissas

Microprocessor & Hardware Laboratory
School of Electrical & Computer Engineering
Technical University of Crete

A thesis submitted to the Technical University of Crete in accordance with the requirements for the Diploma in Electrical and Computer Engineering.

February 2020

THESIS COMMITTEE:

Prof. Apostolos Dollas, Technical University of Crete, Thesis Supervisor
Prof. Dionisios Pnevmatikatos, National Technical University of Athens
Prof. Michalis Zervakis, Technical University of Crete

Abstract

Cellular automata are discrete mathematical models discovered in the 1940s by John von Neumann and Stanislaw Ulam. They constitute a general paradigm for massively parallel computation. Through time, these powerful mathematical tools have been proven useful in a variety of scientific fields.

In this thesis we propose a customizable parallel framework on reconfigurable logic which can be used to efficiently simulate weighted, large-neighborhood totalistic and outer-totalistic cellular automata in real time. Simulating cellular automata rules with large neighborhood sizes on large grids provides a new aspect of modeling physical processes with realistic features and results.

In terms of performance results, our pipelined application-specific architecture successfully surpasses the computation and memory bounds found in a general-purpose CPU and has a measured speedup of up to 51× against an *Intel Core i7-7700HQ* CPU running highly optimized software programmed in C.

ΠΕΡΙΛΗΨΗ

Τα κυψελωτά αυτόματα είναι διακριτά μαθηματικά μοντέλα που ανακαλύφθηκαν τη δεκαετία του 1940 από τον John von Neumann και τον Stanislaw Ulam. Αποτελούν ένα γενικό υπόδειγμα υπολογισμών με εκτενή παραλληλισμό. Μέχρι σήμερα τα μαθηματικά αυτά εργαλεία έχουν χρησιμεύσει σε πληθώρα επιστημονικών τομέων.

Σε αυτή τη διπλωματική εργασία παρουσιάζεται ένα παράλληλο πλαίσιο σε αναδιατασσόμενη λογική το οποίο μπορεί να χρησιμοποιηθεί για την αποδοτική εξομοίωση κυψελωτών αυτομάτων με μεγάλες γειτονιές σε πραγματικό χρόνο. Η εξομοίωση κυψελωτών αυτομάτων με μεγάλες γειτονιές σε μεγάλα πλέγματα προσδίδει νέες δυνατότητες μοντελοποίησης φυσικών διεργασιών με ρεαλιστικά αποτελέσματα.

Όσον αφορά τις επιδόσεις, η παράλληλη αρχιτεκτονική ειδικού σκοπού που σχεδιάστηκε για την παρούσα εργασία ξεπερνά σε επιδόσεις έναν επεξεργαστή γενικού σκοπού, πετυχαίνοντας έως και 51 φορές πιο γρήγορη εκτέλεση από έναν επεξεργαστή *Intel Core i7-7700HQ* ο οποίος εκτελεί βελτιστοποιημένο λογισμικό γραμμένο σε γλώσσα προγραμματισμού C.

Dedication and Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Prof. Apostolos Dollas, for his continuous support, guidance, patience and generosity. Prof. Dollas helped me not only gain a unique perspective on computer engineering, but also surpass all the problems and difficulties that arose during my studies, both in a professional and a personal level. I wouldn't be the person I am today if it weren't for this wonderful mentor and friend.

I would like to express my deepest appreciation to my committee, Prof. Dionisios Pnevmatikatos and Prof. Michalis Zervakis.

I would also like to thank the Technical University of Crete (TUC) for all the support and knowledge I received throughout all these years and especially the members of the Microprocessor & Hardware Laboratory (MHL) and Dr. Eleftheria Sergaki for their insightful remarks and their felicitous advice.

I gratefully appreciate the decisive contribution of Dr. Patrick Lysaght and Xilinx Corporation, who generously donated the Nexys 4 DDR board that was used in this work.

Many thanks to all my friends and colleagues for being there both in great and challenging times. This journey would not have been possible without you. Special thanks, alphabetically, to Maria Apostolidou for being strong enough for both of us when I couldn't be, Vasilis Boglou for all our invaluable discussions, Konstantinos Iordanou for his endless support and encouragement, Anastasis Kampylafkas for his unique insight, Sofia Maragkou for helping me dream of a better, beautiful world and Nikos Tiriakidis for bringing class and elegance in my dull routine.

I also feel much obliged to my friends, Niovi Dollas, Stathis Kolokotronis, Merel Kors, Elisavet Kostalia, Sotiris Litsios, Grigoris Ntailianis and my sister, Christina, for their valuable help and support.

Last but not least, I would like to thank my parents, John and Evi, from the bottom of my heart for all their selfless sacrifices. The opportunity to make my dreams come true is the greatest gift anyone has ever given me.

"*With four parameters I can fit an elephant,*
with five I can make him wiggle his trunk."

John von Neumann

Table of Contents

Chapter 1

Introduction

Cellular automata are discrete mathematical models discovered in the 1940s by John von Neumann and Stanislaw Ulam [1, 2]. They constitute a general paradigm for massively parallel computation and are capable of supporting very large parameter spaces for simulation and modeling. Through time, these powerful mathematical tools have been proven useful in countless ways, both as models of complexity and as models of non-linear dynamic systems in a variety of scientific fields.

1.1 Motivation

It would not be an exaggeration to say that as far as cellular automata modeling is concerned, the possibilities are endless. They are used for modeling a myriad of physical processes, from molecular dynamics [3, 4] to large ecosystems [5] and artificial brains [6]. In cosmology, digital physics suggests that a universal cellular automaton computes the evolution of the universe, which is fundamentally described by digital information [7].

Apart from modeling physical phenomena, cellular automata are also used as abstract computational systems in various applications. The *Universal Constructor*, the first cellular automaton designed by John von Neumann in the 1940s, is an abstract machine which demonstrates the logical requirements for machine self-replication [2, 8, 9]. Since then, the computational abilities of cellular automata have been meticulously studied, with universality and reversibility being essential properties of numerous cellular automata rules [10, 11].

The fascinating world of cellular automata and the potential hidden in their inherent parallelism have been a great motivation for computer scientists and engineers ever since their inception.

1.2 Thesis Contribution

In this thesis we propose a customizable parallel framework on FPGA which can be used to efficiently simulate weighted, large-neighborhood totalistic and outer-totalistic 2D cellular automata in real time. The main contribution of this work is that with the use of FPGA technology one can simulate large cellular automata rules with neighborhood sizes up to 29×29 (whereas typically these are 3×3 or 5×5) on large grids, with very realistic results on the modeling of physical processes.

In terms of results, our pipelined application-specific architecture successfully surpasses the computation and memory bounds found in a general-purpose CPU and has a measured speedup of up to 51x against an Intel Core i7-7700HQ CPU running highly optimized software programmed in C.

1.3 Thesis Outline

This thesis is structurally divided into seven chapters. Following the introductory chapter:

- Chapter 2 contains the theoretical background necessary for one to understand the basic concepts of cellular automata. It also provides a short description of FPGA technology and explains why mapping such problems and applications to FPGAs is desirable.
- Chapter 3 comprises an overview of related works and approaches that have been proposed for accelerating cellular automata simulations with the use of FPGAs.
- Chapter 4 describes in full detail the architecture designed for this thesis. It includes block diagrams of the system's modules and describes their function and interface. In addition, key decisions made during the design process are explained in this chapter.
- Chapter 5 uses examples of simulating different cellular automata rules with the use of this framework to showcase the design's features and advantages over conventional methods as well as some interesting results obtained with it.
- Chapter 6 presents the verification method and tools utilized to test the presented architecture. Furthermore, comparative performance results are presented and discussed in this chapter.
- Finally, chapter 7 provides a summary of the presented work and the conclusion drawn from the experimental results, and discusses potential future work arising from this thesis.

Chapter 2

Theoretical Background

This chapter provides the theoretical background necessary to understand the subject of this thesis. The first part of this chapter gives an overview of the basic cellular automata theory and a thorough example of how one of the best-known cellular automata works. The second part of this chapter includes a brief description of FPGA technology and explains why FPGAs can efficiently accelerate such applications and outperform multi-core CPUs. After the reader has finished reading this chapter, a basic foundation should be set in their mind about how cellular automata simulations break down and what challenges are associated with designing custom hardware for such applications.

2.1 Cellular Automata

Cellular automata are discrete mathematical models used in numerous scientific fields both as models of complexity and as models of non-linear dynamic systems. Cellular automata's main advantage is their flexible rules – just by setting a few simple states and rules, a cellular automaton can model incredibly complex systems.

In this thesis, we will be dealing with 2D cellular automata simulations. A 2D cellular automaton consists of an infinite rectangular grid of homogeneous cells, each in one of a finite number of discrete states. For each cell a set of cells called its neighborhood is defined relative to the specified cell. An initial state of the cellular automaton at time $t = 0$ is selected by assigning a state for each cell. A new generation is created according to some fixed mathematical rules described with a transition function which determines the new state of each cell in the next time interval in terms of the current state of the cell and the states of the cells in its neighborhood.

There are three basic types of neighborhoods in 2D cellular automata: von Neumann, Moore and custom. All three types can be used in an extended and weighted form (fig. 2.1). There are

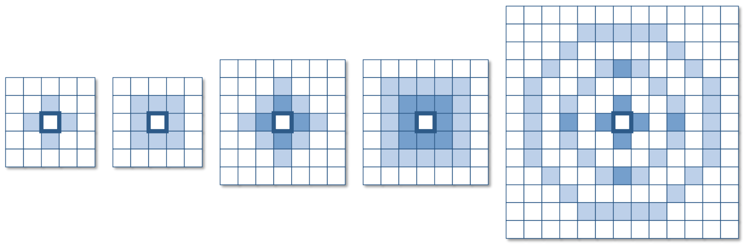


FIGURE 2.1. Basic types of neighborhoods in 2D cellular automata:
a) von Neumann, b) Moore, c) Weighted, extended von Neumann, d) Weighted, extended Moore, e) Weighted, custom neighborhood.

two ways to define a cell's neighborhood; either by its $n \times n$ dimensions in cells, or by its radius r measured as the distance from the central cell to the neighborhood's edge. Note that extending the Moore neighborhood for radius $r \geq 1$ gives a set of cells situated at Chebyshev distance r from the central cell, while extending the von Neumann neighborhood for radius $r \geq 1$ gives a set of cells situated at Manhattan distance r from the central cell.

2.1.1 Totalistic Cellular Automata

A distinct, widely-used class of cellular automata are totalistic cellular automata. The state of each cell in a totalistic cellular automaton is represented by an integer value drawn from a finite set of possible states $S = \{s_0,\ s_1,\ \ldots,\ s_n\}$, and the next state of a cell depends only on the sum of the current values of the cells in its neighborhood [12]:

$$c'(i,j) = \sum_{x=i-r}^{i+r} \sum_{y=j-r}^{j+r} w(x-i, y-j) \cdot c_t(x,y)$$

$$c_{t+1}(i,j) = \begin{cases} s_0 & \text{if } c'(i,j) \leq a \\ s_1 & \text{if } a < c'(i,j) \leq b \\ \cdots & \\ s_n & \text{otherwise} \end{cases}$$

where r is the neighborhood's radius, $w(x,y)$ the neighborhood's weights with $w(0,0) = 0$ and $a, b, \ldots \in \mathbb{Z}$.

If the next state of a cell depends on both its own current state ($w(0,0) \neq 0$) and the total sum of its neighbors, then the cellular automaton belongs to the class of outer totalistic cellular automata [12]. *Conway's Game of Life*, one of the best-known cellular automata, is an example of an outer totalistic cellular automaton with 2 states per cell and a simple Moore neighborhood.

2.1.2 A Totalistic Cellular Automaton Example: the Game of Life

The *Game of Life* is a cellular automaton designed by the British mathematician John Conway and was first published in the October 1970 issue of Scientific American [13]. It is arguably the most widely known cellular automaton and its popularity originates from the complex behavior of the different patterns that occur in it in spite of its quite simple rules.

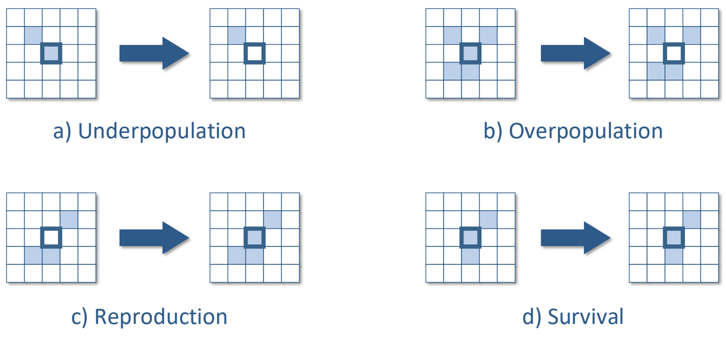


FIGURE 2.2. The *Game of Life* rules.

In the *Game of Life* a cell is either "alive" or "dead" (2 states). The state of a cell in the next time interval is determined by the current states of its neighbors found in the Moore neighborhood of radius $r = 1$ and the following rules (fig. 2.2):

1. If a cell is alive with fewer than two alive neighbors, it dies as a result of underpopulation.

2. If a cell is alive with more than three alive neighbors, it dies as a result of overpopulation.

3. If a cell is dead with exactly three alive neighbors, it is reborn.

4. If a cell is alive with two or three alive neighbors, it survives to the next generation.

These four simple rules are capable of producing complex patterns that exhibit interesting behaviors, such as *still lives*, which do not change through time, *oscillators*, which return to their initial state after a finite number of generations, *spaceships*, which move around the grid (fig. 2.3), *guns*, which are stationary and produce *spaceships* at a constant rate, *rakes*, which move around the grid and emit *spaceships*, and many more.

The fundamental features of movement and interaction that these "creatures" exhibit make it possible to build logic gates, counters and eventually finite state machines by seeding the cellular automaton with a corresponding initial configuration. As a result, *Conway's Game of Life* is capable of turing-complete computation [14, 15].

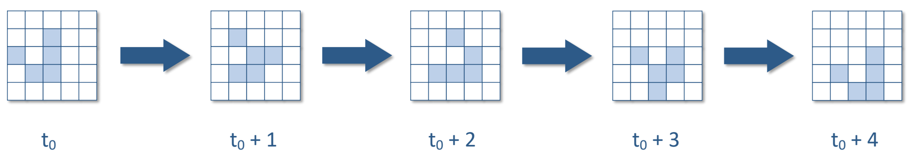


FIGURE 2.3. *Glider*, the smallest spaceship that can exist in the *Game of Life*.

2.1.3 Totalistic Cellular Automata Broadened

Even broader versions of totalistic cellular automata include transition rules with more steps involved, like the *Hodgepodge Machine* discussed later in this thesis.

Calculating cellular automata with a small set of states per cell and small neighborhoods has been thoroughly explored and can be accomplished efficiently by general-purpose CPUs thanks to algorithms like the *Hashlife* algorithm [16], a memoized algorithm which accelerates computation by several orders of magnitude. As the number of states rises, the neighborhood size grows and weights are introduced, the complexity of simulating such cellular automata rules rises dramatically and the use of efficient accelerators becomes crucial.

2.1.4 Real-time Simulation of Cellular Automata

In general, real-time simulation refers to the ability of the simulator to produce new cellular automaton generations at the rate of the medium projecting the images, e.g. a computer monitor. We arbitrarily chose 60 frames/second to match the screen update rate at Full-HD definition. This is not a real constraint though; even with the small FPGA we use, we can go up to close to 100 frames/second. However, this would mean many iterations between outputs, whereas at present a fast CPU can deliver for this type of neighborhood slightly more than 1 frame/second. Real-time observation of cellular automata simulations is not always practiced, as much of the computation is usually performed off-line.

2.2 Field-Programmable Gate Array

A Field-Programmable Gate Array (FPGA) is an integrated circuit whose internal functionality can be reconfigured and customized by the user for a specific application. It consists of a large array of Configurable Logic Blocks (CLBs) connected via programmable interconnects. It also contains memory resources known as Block RAM, which are dual-port RAM units arranged in blocks of a few kB each, and I/O Blocks for communicating with the outside world (fig. 2.4).

CLBs are the fundamental building blocks of FPGA technology. A CLB typically consists of hardware-implemented look-up tables (LUTs), memory components such as latches or flip-flops,

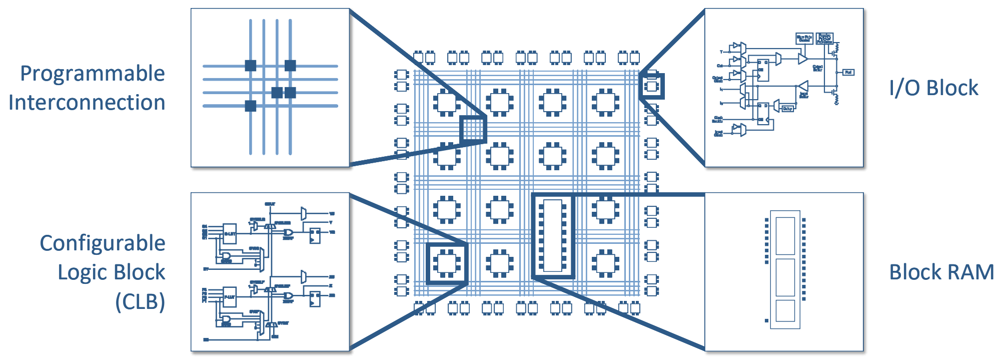


FIGURE 2.4. The basic components of an FPGA.

and multiplexers used as programmable routing controllers within the CLB. By linking several CLBs together via programmable interconnects, one can implement complex synchronous or asynchronous logic circuits.

FPGA hardware design is a multistage process. At first, the designer can specify the configuration of an FPGA by using a hardware description language (HDL), which constitutes an abstract specification of the desired circuit behavior. The code written by the designer is then turned into a logic design through the process of logic synthesis. Synthesis is followed by placement, which is the process of mapping every logic component of the synthesized design into the FPGA's resources. After the placement is completed, routing takes place. The process of routing decides the way the placed logic components will be connected to each other by configuring the FPGA's programmable interconnects.

Synthesis, placement and routing are typically conducted by automated software tools. However, their success is determined by the behavioral description of the design produced by the designer with the use of an HDL. A bad design leads to poor placement, which in turn creates longer routes. Longer routes reduce performance and might even compromise the implemented design's reliable operation. The designer, by applying digital logic techniques like logic simplification, synchronization, pipelining, etc., can ensure a successful implementation of their design.

2.2.1 A Comparison of FPGAs and CPUs

A Central Processing Unit (CPU) is a general-purpose processing device. Any algorithm written in a high-level programming language can be translated into a sequence of operations which can be executed by a CPU. These instructions are executed sequentially, with one instruction per clock cycle at best for scalar processors and a few independent instructions per clock cycle for superscalar processors.

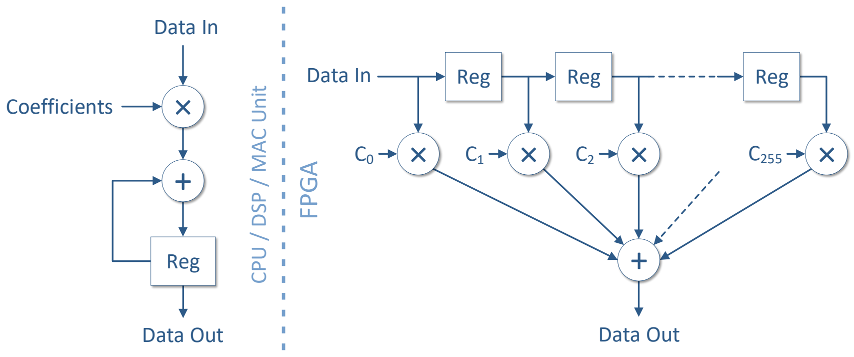


FIGURE 2.5. Multiply and Accumulate (MAC): A CPU needs at least 256 clock cycles to execute a 256-element MAC operation sequentially, whereas an FPGA needs 1 clock cycle for a new 256-element MAC result.

The CPU's clock frequency is about an order of magnitude higher than that of typical FPGA designs. However, the level of parallelism on an FPGA can be 1 to 3 orders of magnitude higher than that of a CPU, depending on the application. This can be achieved thanks to the FPGA's large internal bandwidth and the efficient exploitation of the application's spatial and loop–level parallelism.

The example that follows showcases the exploitation of the spatial parallelism found in 256-element Multiply and Accumulate (MAC) operations. For the sake of this example, latency is omitted and we assume that all data are already available in CPU's cache memory, which in reality is not always the case. For the case of the FPGA we also assume that all data are already loaded in it. As we will see later in Chapter 4, depending on the application, this is possible with the use of an efficient memory controller and a customized buffer.

As shown in fig. 2.5, a CPU core running at 3 GHz would need 256 clock cycles for a new 256-element MAC result, which translates to $256 \times 0.33\ nsec = 84.5\ nsec$. On the other hand, an FPGA design running at 200 MHz, which is a typical frequency of an FPGA design implementation, would produce a new 256-element MAC result every clock cycle (5 $nsec$).

MAC is a very common step in computing, especially in digital signal and image processing. In our application, the neighborhood of a cell is implemented as a sliding window sweeping across the cellular automaton grid. MAC operations are used to apply the neighborhood's weights and calculate all the different sums necessary, depending on the rule applied. For example, if the neighborhood's size in cells is 21×21 and the grid dimensions are 1000×1000 cells, calculating a new cellular automaton generation would require the execution of 10^6 consecutive 441-element MAC operations.

Chapter 3

Related Work

This chapter constitutes an overview of related projects and approaches that have been proposed for accelerating cellular automata simulations with the use of FPGAs. The first section presents a survey of papers and projects of significant importance which influenced the course of this project. A short summary of other noteworthy developments follows before, finally, discussing the approach followed by this thesis.

3.1 A Survey of FPGA-Based Cellular Automata Engines

The following survey presents a number of projects with significant contributions to the field. The examination of these works reveals there are two prevailing approaches to designing hardware for cellular automata simulations.

3.1.1 Toffoli and Margolus' Cellular Automata Machines (1984–2000)

Toffoli and Margolus' *Cellular Automata Machines* (*CAM*) were the first special-purpose computers designed for accelerating cellular automata simulations. Tommaso Toffoli, known for inventing the universal reversible logic gate named after him, has been working on reversible cellular automata and reversible computing since the 1970s. In 1977 he joined MIT and in 1981 he began building a programmable, high-performance cellular automata machine out of TTL electronics and memory chips. The first results were published in 1984 [17].

Toffoli explains that in a truly parallel implementation of cellular automata each cell would have to be implemented as an independent element having access to its own copy of the rule's transition function, its own state variables and its neighborhood's values. However, this approach

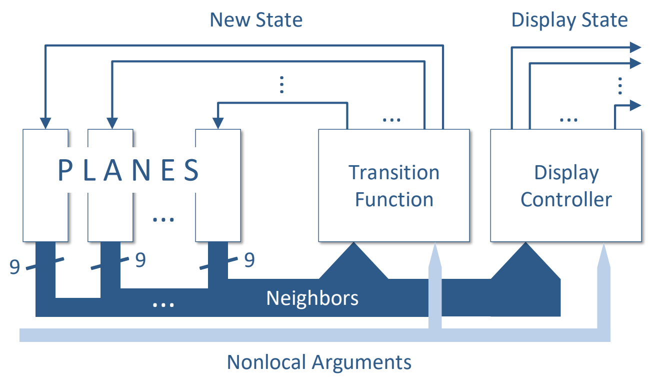


FIGURE 3.1. CAM's basic computational loop. The transition function hardware module is an SRAM LUT.

is in practice restricted by various technological constraints and is not suitable for accelerating large-sized simulations.

The alternative approach followed by Toffoli consists of only one hardware module implementing the transition function (an SRAM LUT), which is "time-shared" between cells. In other words, *CAM*'s architecture processes a stream of cells and their neighborhoods sequentially in order to, eventually, update the whole grid and produce a new frame of the simulation. The issue occurring with this approach is that, as soon as a cell is updated, the neighboring cells will immediately see the new state rather than the old one needed for computing their own new state. In order to solve this and other problems (such as grid boundary conditions), *CAM* uses memory to store both old and new cell states. This technique is commonly known as *double buffering* and we examine it in detail later on, in chapter 4.

CAM's architecture was fully pipelined and its memory consisted of 8 bit-planes, with each plane contributing a single bit to each 8-bit cell (fig. 3.1). The memory planes' size was 256×256 sites and they provided the transition function with all the 3×3 neighborhood cells simultaneously. These features made the system significantly faster than any general-purpose computer at the time, as *CAM* could display the evolution of 256×256 8-bit cells in real time.

Norman Margolus, a PhD student at the time, worked with Toffoli on further developing the prototype. During the next two years the machine's capabilities started growing, many versions followed and in 1986 *CAM-6* was completed. It spawned a book [18], which showcases the machine's numerous applications, and was produced commercially as a PC expansion board.

CAM-6's architecture was fully pipelined and its memory consisted of 4 bit-planes, with each plane contributing a single bit to each 4-bit cell. The size of each plane was 256×256 sites. A pipeline buffer (fig. 3.2) provided the transition function with all the 3×3 neighborhood cells simultaneously. Similar to its predecessor, *CAM-6* could display the evolution of 256×256 cells in real time.

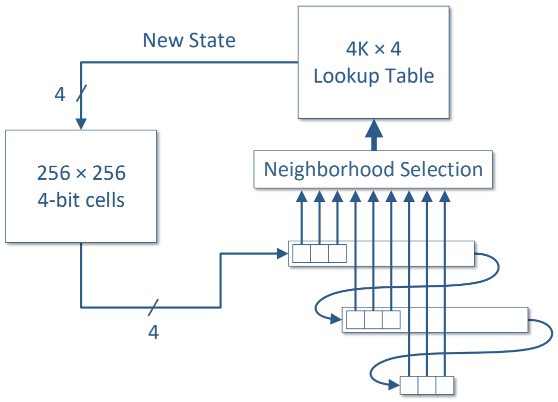


FIGURE 3.2. CAM-6's pipeline buffer provides the transition function with all the 3 × 3 neighborhood cells simultaneously.

A fascinating feature was the ability to rearrange the interconnection of the planes. The user could, at the expense of having 16 states per cell, either obtain multidimensional cellular automata simulations by "stacking" planes on top of one another, or simulate a larger grid by "gluing" planes edge-to-edge. Larger grids could also be simulated by using a technique called *scooping*: the large grid is stored in the host computer's memory and *CAM-6*'s internal 256 × 256 grid is used as a cache memory.

Margolus continued developing *CAMs* within the next few years, with *CAM-8* making its appearance in 1993 [19]. It was a multiprocessor version of its predecessors, with interconnected *CAM*-like modules processing separate sectors of the cellular automaton grid simultaneously (fig. 3.3).

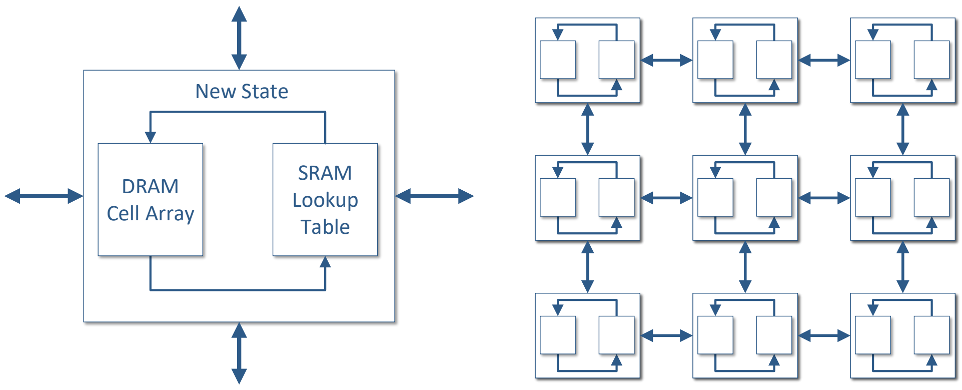


FIGURE 3.3. CAM-8 architecture. Each processing node was connected only to its nearest-neighboring module.

The sectors could be rearranged in any way forming n-dimensional spaces of any size, provided the user had enough *sector modules* in their *CAM-8*. The machine supported arbitrarily large neighborhood sizes and cell sizes in bits, as its computation was based on shifting the data of a sector accordingly before processing them.

Following the steps of its predecessors, *CAM-8*'s performance was outstanding at that time. Its shift-based data manipulation was ideal for simulating lattice gas automata and it could process a 1000×2000 FHP gas model at a rate of 190 frame updates per second. In general *CAM-8* could display the evolution of 512×512 cells in real time. However, for large neighborhoods the performance dropped significantly. For example, when simulating a cellular automaton rule with 3-bit cells and an 11×11 neighborhood size, *CAM-8* could display 10 generations per second.

Even though *CAMs* were not FPGA-based, they were the foundation for Margolus' later work on custom FPGA machines [20, 21]. Through time, his ideas of data permutation and the effective use of fast memories as custom buffers proved to be useful not only for cellular automata simulation, but also for DRAM-based systolic computation in general.

3.1.2 CEPRA: Cellular Processing Architecture (1994–2000)

The *Cellular Processing Architecture* (*CEPRA*) was an FPGA-based architecture developed during the 1990s at the Technical University of Darmstadt. It was a streaming architecture with an internal dataflow similar to that of *CAM*.

The key difference between the two systems was that *CEPRA* used pipelined arithmetic logic instead of LUTs for computing the cellular automaton's transition function. As a result, the advantage of *CEPRA* compared to *CAM* was that complex rules could be computed in one step, whereas *CAM* had to convey their computation through cascaded LUTs.

CEPRA-8L, the first member of the *CEPRA* family, was completed in 1994 [22]. It contained 8 FPGA-based cellular automata processors which could access all their 3×3 neighborhood cells simultaneously thanks to a computation window buffer. *CEPRA-8L* could display 22 generations of 512×512 8-bit cells per second.

CEPRA-1X, *CEPRA-8L*'s successor, was completed in 1997 [23]. It was an FPGA co-processor mounted on a PC expansion board and used the memory of the host computer for storing the cellular automaton grid. *CEPRA-1X* supported 2D and 3D cellular automata with neighborhoods of radius $r = 1$ and could display the evolution of 1024×1024 16-bit cells in real time.

Within the next 3 years the designers of *CEPRA-1X* also created a high-level *Cellular Description Language* (*CDL*) for translating complex cellular automata rules into Verilog HDL [24]. *CDL* programming doesn't require any special knowledge of the system's architecture and can be used as a general cellular automata description language.

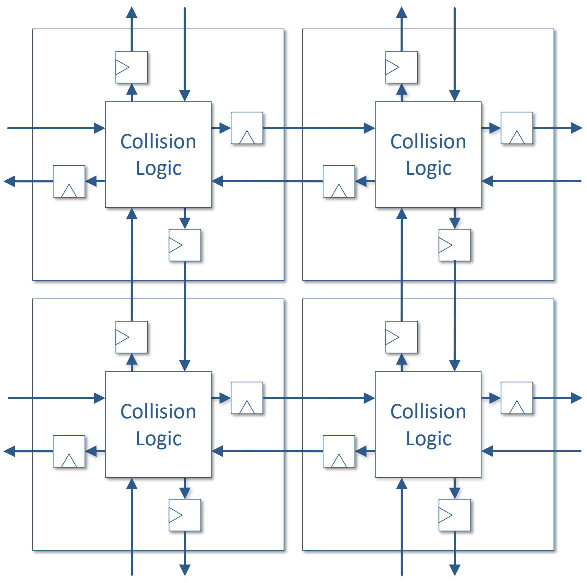


FIGURE 3.4. Top level view of the lattice gas automaton as it was implemented in SPACE.

3.1.3 SPACE: Scalable Parallel Architecture for Concurrency Experiments (1996)

In 1996, Shaw, Cockshott and Barrie from the University of Strathclyde in the UK argued that, as far as lattice gas automata are concerned, parallel machines can outperform LUT-based computers such as *CAM* and yield more useful results [25]. They introduced their *Scalable Parallel Architecture for Concurrency Experiments* (*SPACE*) and proposed a different approach to designing hardware for cellular automata simulations.

As depicted in fig. 3.4, their FPGA-based architecture consisted of an array of interconnected processing elements (PEs), each one of which represented a cell of the HPP model, a fundamental lattice gas automaton. A *SPACE* board, which contained 16 FPGA chips, could simulate a 9×30 lattice gas automaton, achieving nearly a 10x speedup over 2 *CAM-8* modules.

The architecture was scalable and larger lattice gas automata could be simulated by obtaining more *SPACE* boards. On an interesting note, as we will see later in this chapter, the size of a lattice gas automaton that can fit in only one of today's FPGA chips is an order of magnitude larger than that of two *SPACE* boards.

3.1.4 Kobori, Maruyama and Hoshino (2001)

In 2001, Kobori, Maruyama and Hoshino from the University of Tsukuba in Japan presented their own FPGA-based cellular automata system at the 9th IEEE International Symposium on

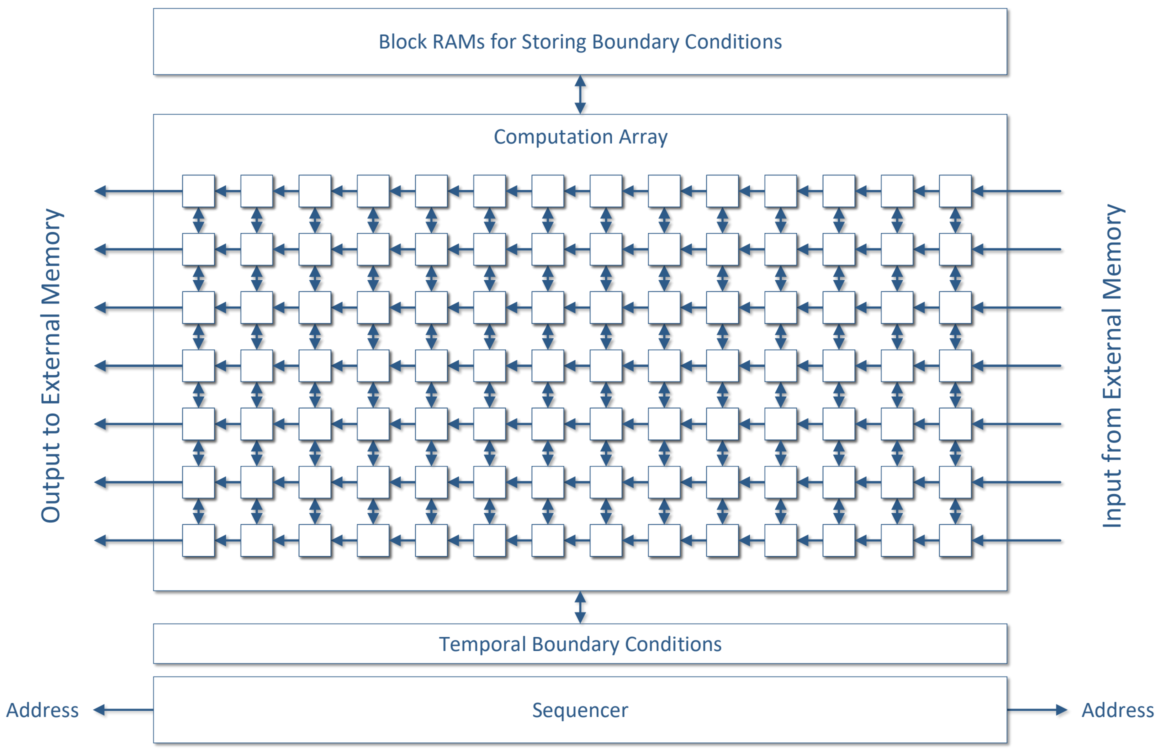

Block RAMs for Storing Boundary Conditions
Computation Array
Output to External Memory
Input from External Memory
Temporal Boundary Conditions
Address
Sequencer
Address

FIGURE 3.5. Overview of Kobori, Maruyama and Hoshino's system architecture.

Field-Programmable Custom Computing Machines [26]. Their design was the result of combining the two approaches discussed in the previous sections of this chapter.

Their streaming architecture consisted of an array of PEs sweeping across the cellular automaton grid (fig. 3.5). In this computation method, if the depth of the PE array is n, each cell of the grid is processed n consecutive times within the FPGA. As a result, if the input cells belong to generation g, the output cells will belong to generation $g+n$.

The problem arising from this method is that when the width of the cellular automaton grid is larger than that of the FPGA's I/O, as the computation moves on within the FPGA, the cells located at the edge of the PE array cannot access the new state of the cells located in their neighborhood, therefore, their new values are invalid. As a result, after each sweep of the grid, the cells that the system has read outnumber the ones that have been produced in its output. The issue is tackled by overlapping consecutive scans of the grid, as shown in fig. 3.6.

Kobori, Maruyama and Hoshino's FPGA-based cellular automata system consisted of an off-the-shelf PCI board with one FPGA and used the host computer to display the results. It could simulate a 2048×1024 FHP lattice gas automaton and calculate 400 generations per second, achieving nearly a 155x speedup over a high-end CPU at the time. However, the cellular automaton's visualization was in pseudo-real time, as most calculated generations never reached

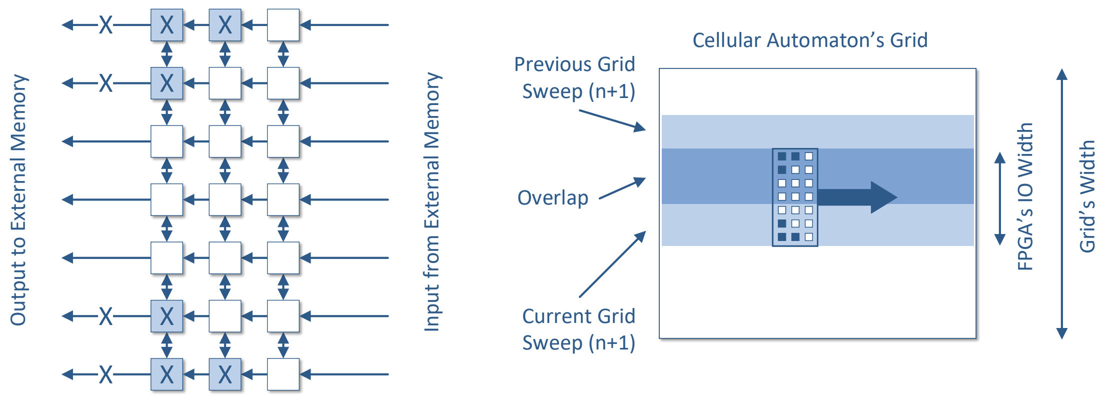

Output to External Memory
Input from External Memory
X
Cellular Automaton's Grid
Previous Grid Sweep (n+1)
Overlap
Current Grid Sweep (n+1)
FPGA's IO Width
Grid's Width

FIGURE 3.6. When the width of the cellular automaton grid is larger than that of the FPGA's I/O, data integrity is preserved by overlapping consecutive scans of the grid.

the PE array's output.

Their system was accompanied by a custom high-level language which could be translated into Verilog HDL. By using that language, the user could specify the size of the PE array and the cellular automaton rule.

3.1.5 Other Significant Work

The contribution of the aforementioned projects to the field of custom cellular automata computers is substantial. However, they comprise only a fraction of the landscape. During the last 3 decades, many other significant projects and developments have contributed their valuable share in the exploration of the field.

In 1991, Bouazza et al. used the *ArMen Machine* to implement cellular automata in a way similar to that of *CEPRA-8L* [27]. The *ArMen Machine* consisted of interconnected FPGAs arranged in a ring and was controlled by a host computer interface board. While the system's performance results were comparable to those of *CAM*, *ArMen*'s routing resources were not sufficient for simulating cellular automata rules with large neighborhoods.

Ten years later, Cappuccino and Cocorullo introduced *CAREM*, a configurable cellular automata co-processor [28]. The processor's architecture consisted of a variable number of PEs which depended on each particular cellular automaton that the processor would be executing. For example, simulating a cellular automaton with only 2 states per cell (1-bit cell) resulted in generating 32 PEs within *CAREM*, since 32 is the maximum number of 1-bit cells that can fit in the system's 32-bit memory word. Although it might seem restricting, this method actually gave the designers plenty of freedom by keeping the memory management unit simple.

From 2007 to 2010, Murtaza, Hoekstra and Sloot from the University of Amsterdam performed a series of studies on the performance modeling of FPGA-based cellular automata systems [29, 30, 31, 32]. With architectures similar to those of Kobori, Maruyama and Hoshino, they experimented with different topologies, sizes and types of PEs, depending on whether a particular cellular automaton simulation is compute-bound or memory-bound. Their experiments concluded with the floating point execution of lattice Boltzmann fluids on FPGA clusters.

In 2013, Lima and Ferreira from the University of Porto presented their own reconfigurable cellular automata architecture [33]. The approach followed was similar to that of *SPACE*. The processing unit consisted of a PE array which implemented the whole cellular automaton within the FPGA. The system could be configured with the use of a GUI software running at the host computer and it could simulate any cellular automata with neighborhoods of radius $r = 1$. The size of the automaton's grid varied and could reach up to 72×72 cells depending on the rule's complexity.

Since then, FPGAs have been widely used to simulate cellular automata, however, most implementations have been custom to a specific cellular automaton rule without the use of large neighborhoods [34].

3.2 Thesis Approach

As we have seen in this chapter, there are two prevailing approaches for designing custom hardware accelerators for cellular automata simulations. The majority of relevant works so far have concentrated on accelerating cellular automata rules with a few states per cell or small neighbordhood sizes. A commonplace approach is to exploit a cellular automaton's spatial parallelism by implementing it as an array of PEs. Each PE represents a cellular automaton cell and is interconnected to its adjacent PEs which are, in turn, the neighboring cells. This method results in outstanding performance when simple cellular automata rules are concerned. However, when it comes to complex rules with many states per cell and large neighbordhood sizes, a PE's demand in logic and routing resources increases and performance drops.

The other approach, which is also followed by this thesis, is to design a streaming architecture which processes the cellular automaton as a stream of cells. This approach is effectively the more sustainable when it comes to rules with large neighborhoods on large grids.

In contrast to prior works, this thesis proposes a system which can execute real-time simulations of any complex cellular automaton with a neighborhood size of up to 29×29 cells on large grids.

Chapter 4

Design Features and Implementation

In this chapter, the architecture designed for this thesis is presented in full detail. Before discussing the system's internal function in section 4.2, section 4.1 provides a description of the system's features in order for the reader to comprehend the decisions made during the design process. Section 4.3 describes the design's versatile customization through an automated process which transforms our design into a universal, reusable environment that can be used to simulate countless cellular automata. Finally, section 4.4 assembles the pieces into a complete and coherent system whose components synchronize and perform like clockwork.

4.1 Design Features

The system designed for this thesis offers a variety of features listed below. As we will see, thanks to these capabilities, our system can be characterized as a versatile general-purpose cellular automata machine.

4.1.1 Number of States and Neighborhood Size

A cellular automaton is ultimately described by the number of states per cell, the size and shape of the cell's neighborhood and the transition rule. The number of cell states and the cell's neighborhood are the two key parameters which determine the rule's capability of accurately modeling various phenomena.

Some models require rules with only a few cell states which represent distinctive states of a phenomenon; for example, in the *Game of Life* a cell is either "alive" or "dead" (2 states). Other models require a higher number of states in order to describe either the evolution of gradual phenomena, such as the temperature change of a material's surface, the density of a substance within a mixture, etc., or dynamic properties, such as spatial orientation or heading direction.

The size of a cell's neighborhood defines the level of detail a model can reproduce both macroscopically and microscopically. Adjusting the neighborhood's shape and weights can result in complex behavior, such as movement, interlacing and anisotropy.

Our system supports either 4-bit cells (up to 16 states per cell) or 8-bit cells (up to 256 states per cell) and neighborhood sizes of up to 29×29 cells. As shown later in section 4.3, these two parameters can be set by the user at will before synthesizing our design and they determine the size of the system's buffers, internal ports and buses.

4.1.2 Types of Grids and Grid Size

In theory, cellular automata evolve on an infinite grid. In practice, computers have finite memory and can only store and simulate cellular automata on a finite grid. A type of finite grid is the rectangular grid, which is often used because of its simplicity to implement and visualize. However, its use requires the need of explicitly defined boundary conditions, as the cells located at the edge of the grid have no access to a complete neighborhood.

One solution is to create a zone of fixed-state cells around the rectangular grid (for example, zero padding) and special rules for those cells in order to enclose the cellular automaton's evolution within the said grid. This technique is useful when it comes to spatially bounded phenomena, such as a lattice gas automaton propagating within a container. However, it is quite restrictive as far as spatially large phenomena are concerned, as it would be better to completely avoid the need for "special cases".

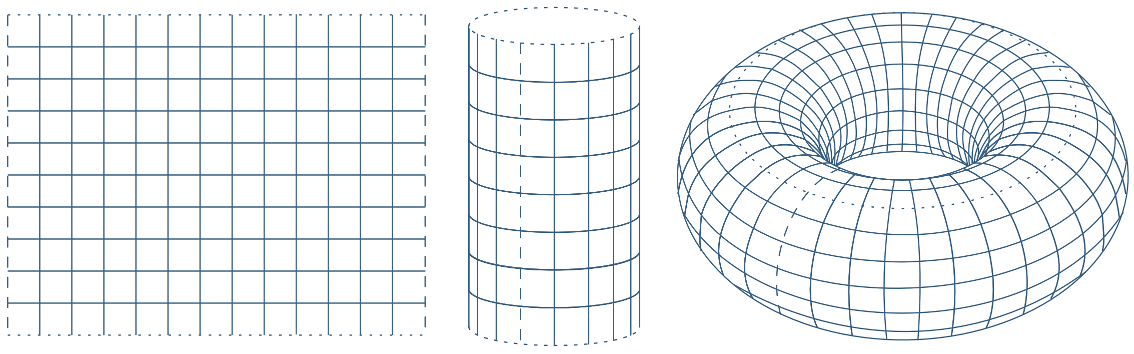


FIGURE 4.1. Our design supports three different types of grids. The "periodically infinite" toroidal grid eliminates the need for special boundary conditions.

Another solution arising to tackle this problem is to handle the cellular automaton's space as a finite grid without edges, in the shape of a torus. The toroidal grid, which is often described as "periodically infinite", is formed by wrapping-around the edges of the rectangular grid as shown in fig. 4.1. As we saw earlier in chapter 3, all previous cellular automata hardware simulators supported toroidal grids. The types of grids supported by our system are the rectangular grid, the toroidal grid and an intermediate type of grid which is the cylindrical grid (fig. 4.1).

Large cellular automaton neighborhoods have no use if the automaton is executed in small grids, as they produce large patterns which are visible only within a large grid. The grid size supported by our system is 1920×1080 cells, large enough for large patterns to form, propagate and interact with each other across the cellular automaton's universe. The grid size chosen is also the resolution of modern Full-HD screens, with each pixel of the screen representing a cellular automaton cell of the grid.

4.2 System Architecture and Implementation

At the beginning of this section we will provide a brief description of the system as a whole, before analyzing its internal structure and operation. A simplified schematic of the design is shown in fig. 4.2. Our system was designed with VHDL and consists of four basic subsystems:

1. *Memory Initialization* and *Frame Extraction* running at 100 MHz.
2. *Memory Controller* generated by *Xilinx's Memory Interface Generator* running at 325 MHz, providing a user interface clock at 81.25 MHz (4:1).
3. *Cellular Automaton Engine* datapath and buffers running at 200 MHz.
4. Full-HD Graphics running at 148.5 MHz.

At first the system's memory needs to be loaded with the initial cellular automaton's state, which is defined as the initial configuration for each cell of the grid at time $t = 0$, via UART from a computer. After the memory initialization process is complete, the system starts displaying the stored cellular automaton grid on screen via VGA at 1080p. Every line that is loaded into the *Graphics Controller*'s buffer following the controller's request is also loaded into the *Cellular Automaton Engine*'s buffer.

The *Cellular Automaton Engine*'s buffer holds all the grid lines needed to provide the engine with the neighborhood of each cell of the line being processed. This results in a sliding window in the size of the cell's neighborhood moving across the grid, processing 1 cell per clock cycle.

As we mentioned earlier, this thesis project constitutes a framework on which the user can build their own cellular automaton hardware simulations. This translates into a design which automatically performs the process of resource dimensioning, allocation, interconnection and synchronization, and generates a ready-to-go system which can support any cellular automaton simulation, provided that the automaton's characteristics lie within the system's specifications.

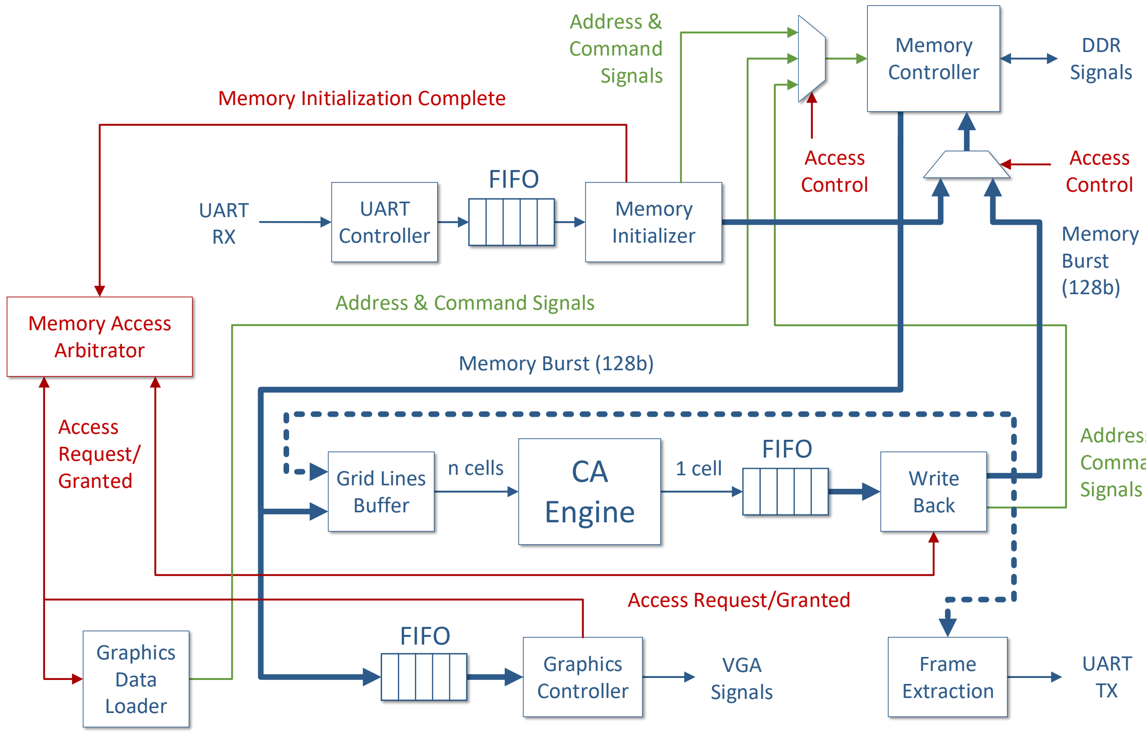


FIGURE 4.2. A simplified schematic of the system architecture.

In order for the reader to fully comprehend the automatic generation process, which will be described sufficiently in section 4.3, the system's inner workings and their correlation with cellular automata must be presented first. Thus, for now, we will simply introduce a few variables which will help us describe the design's inner structure dimensions and resources without troubling ourselves with how these will be actualized. We introduce the following variables:

n,	$n \times n$ neighborhood size, $n \in [3, 29]$. For example: $n = 29$ for a 29×29 neighborhood.
c,	cell's size in bits, $c \in \{4, 8\}$. For example: $c = 4$ for a 4-bit cell.
b,	memory burst's size in bits, $b \in \mathbb{N}$. For example: $b = 128$ for a 128-bit burst.
x, y,	the grid's dimensions in cells, $x, y \in \mathbb{N}$. For example: $x = 1920$ and $y = 1080$ for a 1920×1080 grid.

4.2.1 Double Buffering

A cellular automaton is a discrete world with discrete time, which means that time in cellular automata consists of distinctive time steps. A new cellular automaton generation is produced in the next time interval once the transition function determines the new state of each cell of the grid in terms of the current state of the cell and the states of the cells in its neighborhood. Thus,

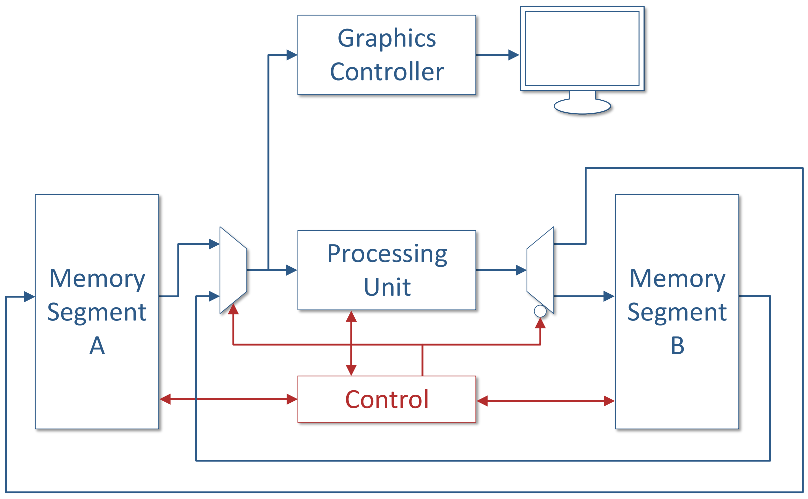


FIGURE 4.3. Double buffering.

in order to calculate a new cellular automaton state we need to have a complete timestamp of the previous state of the automaton.

A technique called double buffering constitutes the best choice for us. As shown in fig. 4.3, with the use of double buffering, a completed frame of our cellular automaton's state is presented to the user while the same frame's data are processed by our system's cellular automaton engine to produce the next timestamp. In order to simulate a cellular automaton in real time, the aforementioned process must be completed 60 times per second.

4.2.2 Memory Controller and Grid Representation in Memory

The two memory segments shown in fig. 4.3 are two distinctive parts of the external memory. Those two segments are large enough to hold a frame of $x \times y \times c$ bits each. As it becomes obvious, for the same grid size, the segments occupy a variable amount of memory space which depends on c, the cell's size in bits.

The external memory, which in our testing setup was a 128 MB DDR2 memory, receives and transmits data in the form of word bursts. Our system uses *Xilinx's Memory Interface Solution* as its *Memory Controller*. This customizable controller supports DDR, DDR2 and DDR3 memory interfaces. In our case, the controller runs at 325 MHz and provides a user interface clock at 81.25 MHz (4:1). It sends and receives 128-bit bursts ($b = 128$) which contain eight 16-bit words each.

In our application, the memory's word size is irrelevant. The data bursts contain cellular automaton cells and, to our system, these are the equivalent of words. However, the cell's size is variable and, as a result, a new significant variable is introduced:

c_b, number of cells per memory burst, $c_b = \frac{b}{c}$, $c_b \in \mathbb{N}$.

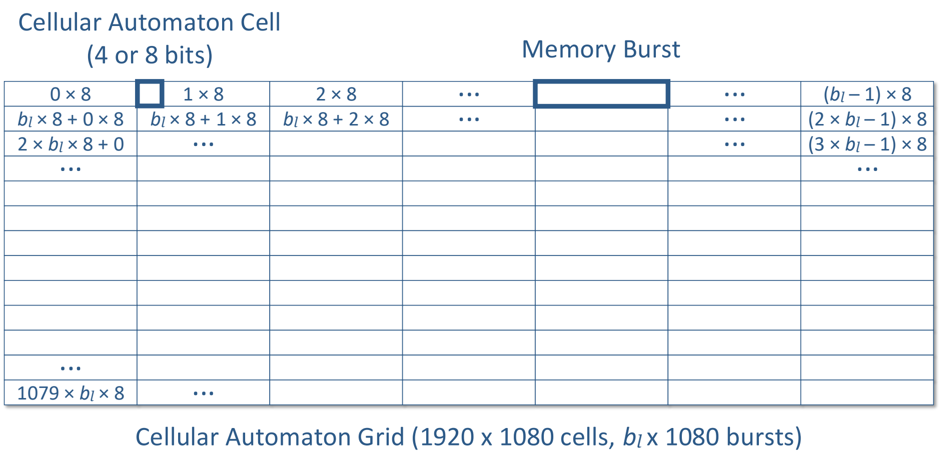

Cellular Automaton Cell
(4 or 8 bits)
Memory Burst
0 × 8
1 × 8
2 × 8
⋯
⋯
$(b_l - 1) \times 8$
$b_l \times 8 + 0 \times 8$
$b_l \times 8 + 1 \times 8$
$b_l \times 8 + 2 \times 8$
⋯
⋯
$(2 \times b_l - 1) \times 8$
$2 \times b_l \times 8 + 0$
⋯
⋯
$(3 \times b_l - 1) \times 8$
⋯
⋯
⋯
$1079 \times b_l \times 8$
⋯
Cellular Automaton Grid (1920 x 1080 cells, b_l x 1080 bursts)

FIGURE 4.4. Grid representation in memory and burst addressing.

The 1920 × 1080 grid is represented in memory as shown in fig. 4.4. Each line consists of 1920 cells parcelled up in bursts, with each burst containing c_b cells. The number of cells per burst is a variable number and, consequently, so is the number of memory bursts per grid line:

b_l, number of memory bursts per grid line, $b_l = \frac{x \times c}{b} = \frac{x}{c_b}$, $b_l \in \mathbb{N}$.

The memory holds two consecutive frames of the grid for the purpose of double buffering. Both memory segments can be accessed with the same addressing pattern as shown in fig. 4.4, and the only thing distinguishing them is the address's most significant bit. This addressing pattern is known as *simple horizontal scan*. However, it must be clarified that this method has nothing to do with the way the system sweeps the cellular automaton grid in order to process it. The horizontal scan concerns only the way the system reads data from the external memory and loads it into its buffers.

4.2.3 Initial Configuration and System Initialization

An initial configuration of the automaton's grid, also commonly known as the cellular automaton's initial state, is required in order for a cellular automaton to begin its operation. This initial state is set by assigning a state for each cell of the grid.

As far as our design is concerned, the preparation of the grid's initial configuration must be prepared in the host computer before our system begins its operation. Due to the fact that this is a one-person project focusing on hardware design, the process designed for this task consists of a few basic scripts and programs instead of developing a complete software suite with a graphical user interface. This process, which is shown in fig. 4.5 and described below in full detail, comprises a simple yet efficient and ubiquitous way for preparing an initial configuration for our system.

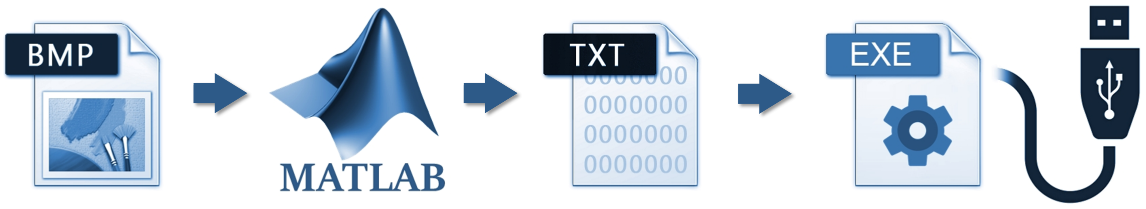


FIGURE 4.5. The process that must be followed in order to prepare and send an initial cellular automaton configuration to our system.

The first step of the process is to prepare a bitmap image of the initial state of the automaton's grid. A bitmap image is a suitable way of representing the grid for two reasons. First, it constitutes a visual way of representing the grid's data which helps the user have a complete view of the grid after every modification, no matter how major or minor that is. Second, bitmap image pixels can be represented by a variable number of bits per pixel. This feature is convenient for us, as there can be a direct match between pixels of 4 bits (16 colors) and 4-bit grid cells, or between pixels of 8 bits (256 colors) and 8-bit grid cells.

Any image editing software which can handle bitmap files will do for the task. All images prepared for the application examples of this thesis (chapter 5) were designed with *Microsoft Paint*, whose 16-color palette is one of those also supported by our system's graphics subsystem.

Once the bitmap image of the grid is complete, a Matlab script transforms the image into a delimited text file which contains space-separated pixel values. Every text line represents a different row of the image.

An executable running in *Microsoft Windows* handles the UART connection between the host computer and the FPGA board and transfers the text file to our system via USB at a rate of 2 MBd. The file transfer time depends on c, the cell's size in bits, since we can pack either one or two cells per byte being transmitted. A grid which consists of 8-bit cells takes up to 10 seconds to be transmitted, while transmitting a 4-bit cells grid takes up half the time.

During the initialization process, our system stores every byte that receives in a FIFO buffer which packs the bytes up into bursts. Our design's initialization subsystem is responsible for storing the bursts in our system's external memory in order to reserve the two segments of the memory needed for double buffering. It implements a Finite-State Machine (FSM) whose operation is shown in fig. 4.6. While some steps could be parallelized, that would be a waste of resources since UART is much slower than our design's logic. Once the initialization process is complete, it asserts a signal notifying the rest of the system that it can now begin processing the automaton's grid.

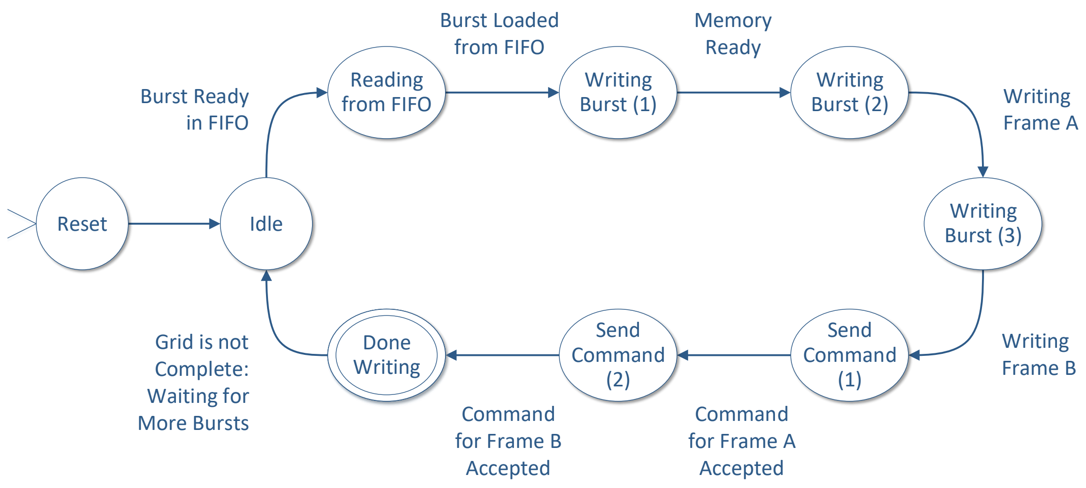


FIGURE 4.6. The FSM describing our system's memory initialization process.

4.2.4 Graphics and System's Data Loader

After initialization is complete, the system starts displaying the contents of the memory on screen alternating between the two memory segments for the purpose of double buffering. Our design executes the simulation of cellular automata in real time, which means that it produces and displays 60 cellular automaton generations per second.

The graphics subsystem is the part of the system which is responsible for displaying the simulation's frames on a screen. It consists of three modules:

1. The *Graphics Data Loader* which loads from memory the data to be displayed.
2. The *Full-HD controller*, the graphics controller which generates the video synchronization pulses.
3. The *Color Palette* which associates each pixel data with a color.

The graphics controller is the one defining the system's timing and synchronization requirements as a whole, as we have to make sure that the processing engine will have generated a new frame before the graphics subsystem requests for it.

4.2.4.1 Graphics Data Loader

The graphics subsystem must have immediate access to the external memory whenever it needs to load part of the frame. Since memory access is shared between the graphics loading part of the frame and the processing engine storing part of the new frame, the memory access time had to be kept as short as possible for the graphics. For that purpose, the graphics subsystem loads one frame line at a time into its buffer in order to render it on the screen. Horizontal scanning, which we described earlier in section 4.2.2, accelerates the load operation, achieving nearly 1 burst received per clock cycle.

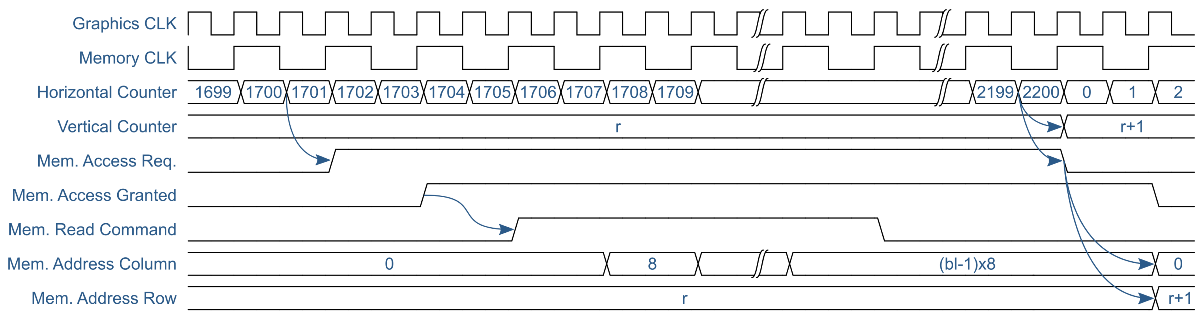


FIGURE 4.7. The timing diagram of a frame line being loaded into the system's buffers. The system loads line r while the graphics subsystem is almost done rendering line $r-1$ on the screen.

Loading a new frame line into the graphics' buffer is carried out by the *Graphics Data Loader* once the graphics controller has completed drawing 75% of the line currently being displayed on screen (fig. 4.7). The loader is in direct communication with the graphics controller and acts as a mediator between the graphics controller and the memory controller. Every time the loader completes loading a frame, it alternates reading between the two memory segments as dictated by double buffering.

The *Graphics Data Loader* is also the whole system's data loader, as it is the only module requesting data to be loaded from memory. The *Cellular Automaton Engine* accesses the same data without requesting any additional loads from memory.

4.2.4.2 Full-HD Controller

The *Full-HD Controller* is the part of the graphics subsystem which generates the video synchronization pulses for the monitor in order for it to enter the 1920 × 1080 @ 60 Hz resolution state (full high definition). It also provides horizontal and vertical counters for the pixel currently displayed. While our system can theoretically be adjusted for any grid size, in our case x and y are defined by the graphics controller's resolution and our choice to associate each cellular automaton cell with 1 pixel on screen.

4.2.4.3 Color Palette

In order to display an image on the screen, the graphics subsystem needs to associate each synchronization pulse with an RGB value. It reads c-bit (4-bit or 8-bit) data serially from a memory burst stored in its buffer, transforms it into a 12-bit color output signal according to the color palette chosen by the user, and transmits it alongside the corresponding pulse signals produced by the Full-HD synchronization controller.

The color palettes supported by our system are shown in fig. 4.8. The first two consist of 16 colors and refer to 4-bit cells. The first one is the standard 16-color *Microsoft Windows* palette, suitable for displaying cellular automata with a few, distinctive states which must be easily distinguishable by the viewer. The second 16-color palette is a black and white gradient palette, suitable for displaying cellular automata which model gradual phenomena. The third palette is also a gradient one consisting of 256 colors and it refers to cellular automata with 8-bit cells.

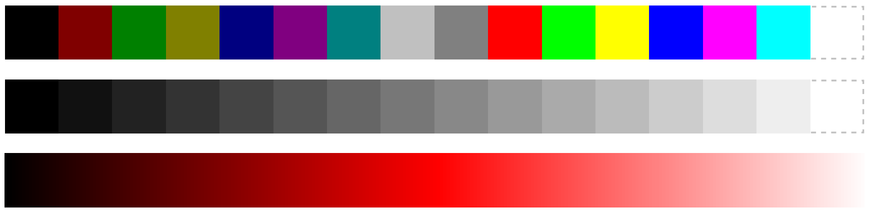


FIGURE 4.8. The color palettes supported by our system.

4.2.5 Cellular Automaton Engine

The *Cellular Automaton Engine* is responsible for calculating the new values of the cellular automaton grid's cells. It operates at 200 MHz and it produces a new cell value per clock cycle. Processing a cellular automaton cell consists of three basic steps:

1. Loading the neighborhood's cells.
2. Multiplying the cells with the neighborhood weights and then computing their sum ("Multiply and Accumulate").
3. Calculating the cell's new state according to the sum's value (Transition Function).

All three steps have to be optimized in order for the engine to be able to process 1 cell per clock cycle. In order to optimize the neighborhood loading process, our *Cellular Automaton Engine* features a pipelined neighborhood window implemented as an array of shift registers. The pipelined window, which can be seen in fig. 4.10, accepts a neighborhood column consisting of $n \times c$ bits as its input and shifts the whole window at every clock cycle. This results in a sliding window in the size of the cell's neighborhood moving across the grid as shown in fig. 4.9. This way, and loading a cell's neighborhood data into the engine is reduced from a O(n^2) problem into a O(1) task in terms of time, i.e., the engine has the complete neighborhood of the cell being processed available at clock rate.

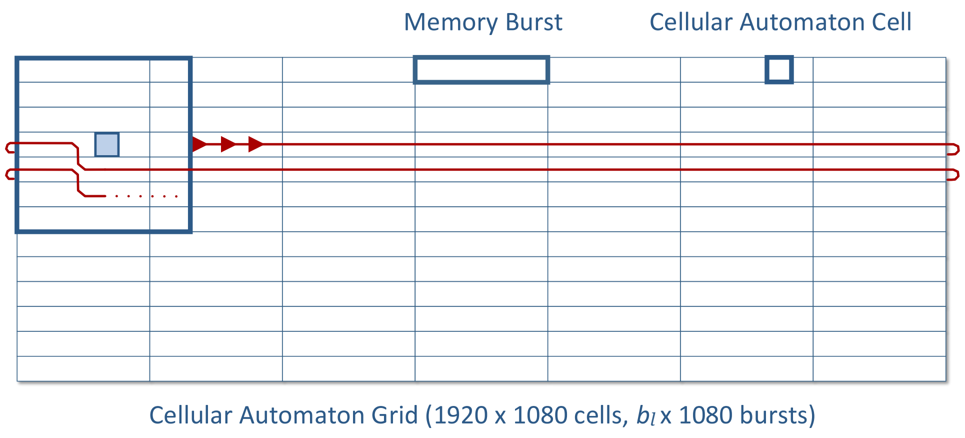

Memory Burst
Cellular Automaton Cell
Cellular Automaton Grid (1920 x 1080 cells, b_l x 1080 bursts)

FIGURE 4.9. The pipelined neighborhood sliding window sweeping across the grid.

After loading the neighborhood, all the cell values located in the cell's neighborhood have to be multiplied and accumulated (MAC). Each of the $n \times n$ cells is multiplied with its weight value, before entering an adder tree which calculates the sum of all the weighted cells. The adder tree performs the addition of $n \times n$ elements, each of which is $c \times w$ bits wide, where w is the size of each neighborhood weight in bits. With the neighborhood reaching sizes up to 29×29 cells, the amount of the required arithmetic logic resources limits the width of the weights to 4 bits. However, if numerous weighted cell values can be calculated only with the use of shift operations, one can use larger weights and multipliers for the rest of the neighborhood.

The arithmetic logic of the *Cellular Automaton Engine* is completely pipelined, providing the *Transition Function* with a new result at every clock cycle. The engine's *Transition Function* is a simple look-up table and does not require more than 1 clock cycle to produce the new value of the cell being processed. As a result, the *Cellular Automaton Engine* as a whole can produce a new cell per clock cycle, provided it's supplied with a neighborhood column of cell at clock rate.

Another important feature, which needs to be mentioned for future reference, is that there is no input signal enabling the *Cellular Automaton Engine*'s data input. The *Cellular Automaton Engine* reads a neighborhood column at clock rate, without taking into account what lies on the bus. Instead, it also accepts a 1-bit *valid* signal which characterizes the central cell of the column currently being read. If the central cell of the column is *valid*, then the cell's new value will be written back to the external memory, otherwise it will be neglected by the system's *Write-Back* module.

The *Cellular Automaton Engine*'s high performance is apparent when the window slides

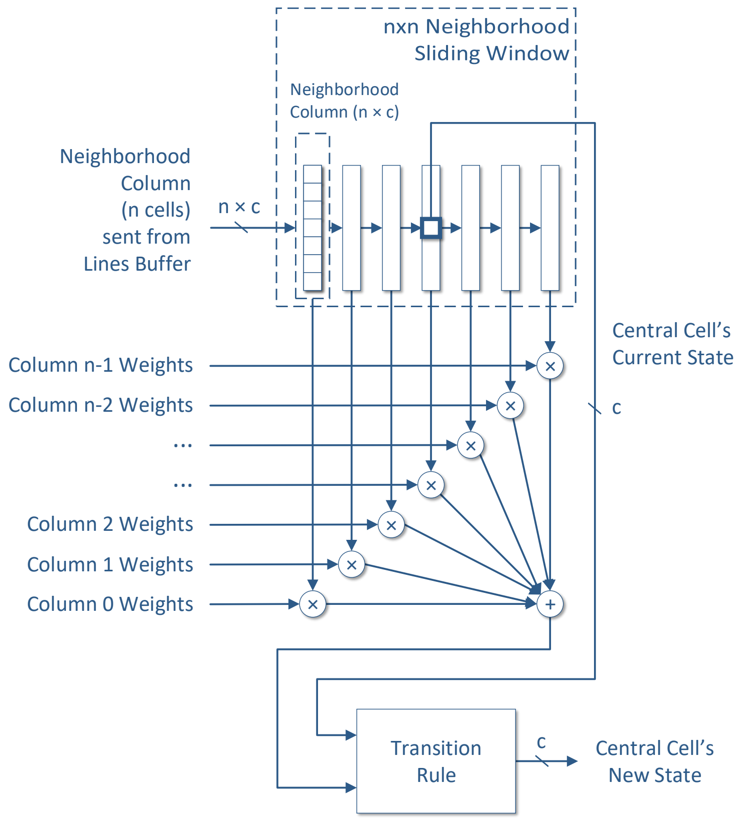

nxn Neighborhood Sliding Window
Neighborhood Column (n × c)
Neighborhood Column (n cells) sent from Lines Buffer
n × c
Column n-1 Weights
Column n-2 Weights
...
...
Column 2 Weights
Column 1 Weights
Column 0 Weights
Central Cell's Current State
c
Transition Rule
c
Central Cell's New State

FIGURE 4.10. The Cellular Automaton Engine.

horizontally across the grid. However, the window can not slide vertically. The system has plenty of time between frame lines to unload the last cells of the line currently being processed and load the neighborhood window with the first valid cells of the next line.

While in general our design is automatically generated based on the user's preferences, the *Cellular Automaton Engine*'s function depends specifically on the cellular automaton's rule. For that reason, the engine must be redesigned every time according to the application, with the aforementioned general guidelines applying in all cases as far as its internal structure is concerned. Therefore, the engine's latency varies and depends directly on the neighborhood's size n and the rule's complexity.

4.2.6 Grid Lines Buffer: Rectangular and Cylindrical Grid

As we mentioned earlier, the *Cellular Automaton Engine* produces a new cell per clock cycle, provided it's supplied with a neighborhood column of cells at clock rate. This is possible thanks to the *Grid Lines Buffer*.

The *Grid Lines Buffer* (fig. 4.11) consists of n BRAM modules which store the grid lines needed to supply the *Cellular Automaton Engine* with the $n \times n$ neighborhood of the all the cells located in a grid line, plus one BRAM module used as a write buffer. As a result, the amount of BRAM resources required equals to $(n+1) \times x \times c$ bits $= (n+1) \times b_l \times b$ bits.

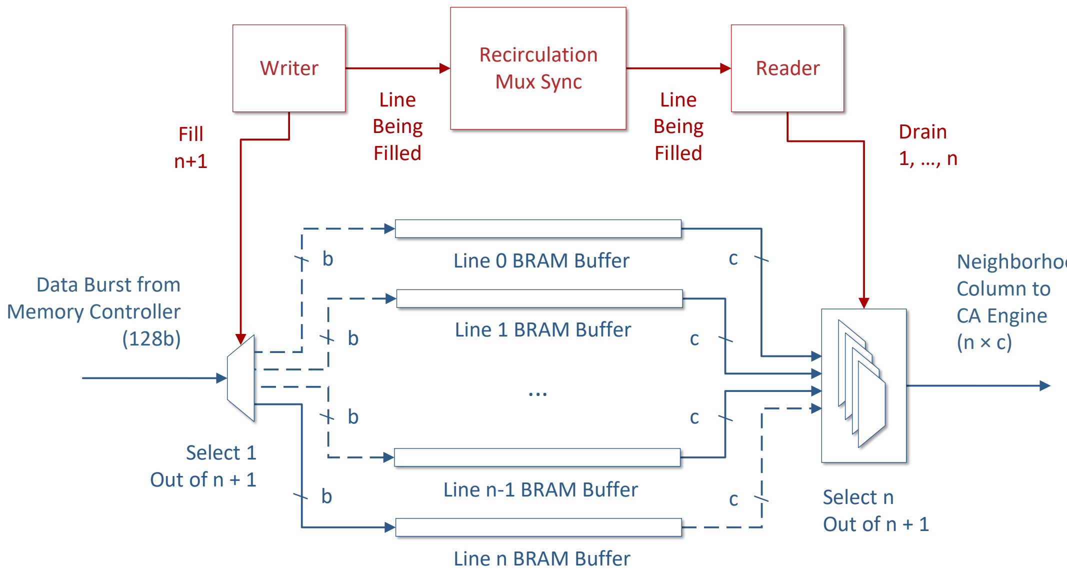


FIGURE 4.11. The Grid Line Buffer's internal structure.

The *Grid Lines Buffer*'s control is implemented as two communicating FSMs; a reader and a writer. The writer operates at 81.25 MHz, which is the frequency at which the memory bursts arrive, and writes the buffer every time the system receives a grid line requested by the *Graphics Feeder*. In the meantime, the reader drains n BRAM modules at 200 MHz in order to feed the *Cellular Automaton Engine*.

As soon as a complete grid line has been drained/filled, the buffer's control logic will "shift down" the buffer window. The line that was loaded last is ready to be drained along with the $n-1$ most recently loaded lines, and the earliest loaded line is ready to take upon the role of the write buffer.

The reader and the writer communicate with each other with the help of a recirculation multiplexer synchronizer, so that we make sure that the system won't start filling a line that hasn't yet been drained. This event will never take place unless the real-time characteristics of the system have somehow been compromised.

The *Grid Lines Buffer* module does not require any intervention by the user. The number and size of the BRAM modules required and the depth of the pipeline are generated automatically based on n, the size of the cellular automaton's neighborhood, and c, the cell's size in bits. The pipeline's depth is variable because, as the neighborhood size increases, so does the number of BRAM lines and we need to provide the routing process with some slack. Hence, the latency of the buffer is also variable and equal to n cycles.

4.2.6.1 Rectangular and Cylindrical Grid

In section 4.2.5, when we described the the *Cellular Automaton Engine*'s operation, we mentioned the *valid* signal and the fact that the *Cellular Automaton Engine* always reads a neighborhood column at clock rate without having an input signal enabling its data input.

This feature allows us to pre-load the neighborhood of the *Cellular Automaton Engine* before providing it with valid cells as shown in fig. 4.12. As a result, we can define the neighborhood of the cells located at the edge of the rectangular grid, which would normally have incomplete neighborhoods. By pre-loading the *Cellular Automaton Engine*'s pipelined neighborhood window with zero values, we create a zero-padded rectangular grid for our cellular automaton simulation.

The cylindrical type of grid is the result of "wrapping-around" the vertical edges of the rectangular grid. In that way, a cell located at the left edge of the grid does not have an incomplete neighborhood, since its neighborhood contains cells located in the right edge of the grid. This is feasible by pre-loading the *Cellular Automaton Engine*'s neighborhood window with the $(n-1)/2$ last cells of the buffer's lines before sending the valid cells to be processed. For the atmost right cells of the grid the buffer sends the $(n-1)/2$ first cells of the buffer's lines even after it has finished sending valid cells.

The user can choose between using a rectangular, a cylidrical and a toroidal grid without intervening with the design. The toroidal grid's implementation is described later in section 4.2.6, as it is necessary for the reader to have a complete view of the system first.

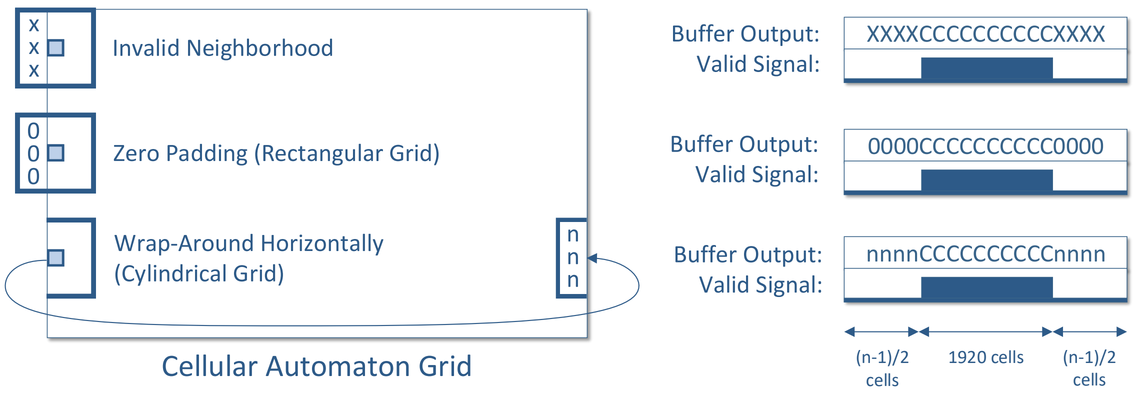


FIGURE 4.12. Pre-loading the Cellular Automaton Engine's neighborhood window.

4.2.7 Write-Back

Every time the *Cellular Automaton Engine* processes a grid line, it produces $x = 1920$ valid cells at the rate of 1 cell per clock cycle. The engine's output is connected to a FIFO buffer which parcells up the cells in bursts. The FIFO's data output is connected directly to the *Memory Controller*'s data bus.

Write-Back (fig. 4.13), which operates at the *Memory Controller*'s interface clock rate, acts as the mediator between the FIFO buffer and the *Memory Controller*, making sure that a burst has been successfully written to the memory before requesting for the next burst from the FIFO buffer. It handles the *Memory Controller*'s and the FIFO buffer's control signals, including the memory's address bus.

The *Write-Back* module keeps track of how many bursts and lines it has written to memory. Once it has written $x \times b_l$ bursts, it alternates writing between the two memory segments as dictated by double buffering.

As soon as a burst is available within the FIFO buffer, *Write-Back* attempts to write it in memory, provided it has access to do so, which means the *Graphics Data Feeder* does not read anything from the memory at the time. The *Write-Back* module is fully pipelined and can write 1 burst per clock cycle as long as both the FIFO buffer and the *Memory Controller* are available.

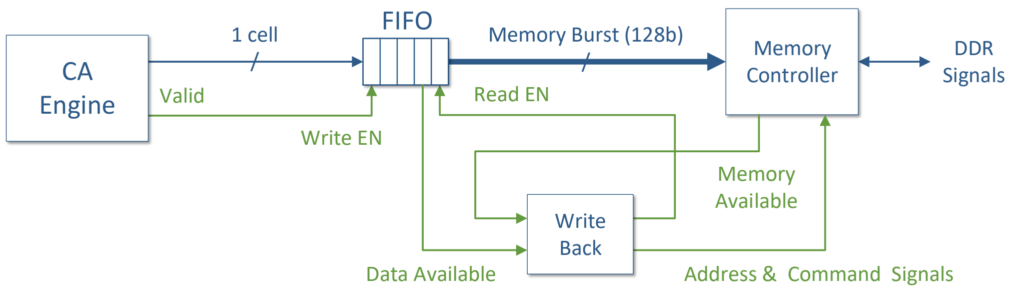


FIGURE 4.13. Write-Back.

4.2.8 Memory Access Arbitrator

We've mentioned numerous times so far that memory access is shared between the *Graphics Data Loader* and the *Cellular Automaton Engine's Write-Back*. The *Memory Access Arbitrator* is responsible for settling which subsystem has access to the memory bus at any given clock cycle.

The *Graphics Data Loader* has a priority over *Write-Back* in order to not disrupt the graphics' 60 frames per second refresh rate. The loader occupies the memory bus only for 25% of the time needed to render a frame on the screen. The remaining 75% of the time is enough for the

Cellular Automaton Engine and *Write-Back* to process and write an entire new cellular automaton generation in memory.

4.2.9 Grid Lines Buffer: Toroidal Grid

Now that the reader has a complete view of the processing stages and the system's structure so far, we will discuss the implementation of the toroidal grid. The toroidal grid is a grid in the shape of a torus, which means that it has no edges and, when seen as a rectangular frame, the neighborhood can wrap-around both horizontally and vertically.

The first step in implementing a toroidal grid is to wrap-around the vertical edges of the grid to form a cylinder, like we did for the cylindrical grid. When it comes to a torus, this is known as a *toroidal wrap-around*. Folding the horizontal edges of the cylinder in order to create a torus is called a *poloidal wrap-around*.

When it comes to the toroidal (horizontal) movement of the neighborhood's sliding window across the grid, the *Grid Lines Buffer* operates exactly as it did for the cylindrical grid, which means pre-loading the *Cellular Automaton Engine* with the last cells of the grid lines before beginning with the processing and so on. However, when it comes to the poloidal (vertical) movement of the neighborhood around the grid, extra BRAM line buffers and logic are required since the wrap-around involves grid lines that the system has accessed before and won't be loaded again by the *Graphics Data Loader*.

Consider the neighborhood of a cell located in the first line of the grid. All the neighborhood cells located above the said cell are found in the last $(n-1)/2$ lines of the frame. Yet the last lines of the frame will not be loaded by the *Graphics Data Loader* until the rendering process reaches the end of the frame. In order to tackle this problem, a solution is to store these lines in the *Grid Lines Buffer* when *Write-Back* writes them in memory while processing the previous frame.

In a similar way, let us consider the neighborhood of a cell located in the last line of the grid. All the neighborhood cells located under the said cell are found in the first $(n-1)/2$ lines of the frame. However, those lines have been long loaded by the *Graphics Data Loader* at the start of the frame rendering process and no longer exist within our system's buffers. In order to overcome this problem, the toroidal *Grid Lines Buffer* stores those lines in separate BRAM line buffers when the *Graphics Data Loader* loads them at the start of the frame rendering process and keeps them until needed at the end of the frame.

The amount of BRAM resources required to implement the toroidal *Grid Lines Buffer* equals to $[(n+1)+(n-1)/2+(n-1)/2] \times x \times c$ bits $= 2 \times n \times x \times c$ bits $= 2 \times n \times b_l \times b$ bits. The extra BRAM line buffers needed for the toroidal buffers are called *Poloidal Line Buffers*, as they involve the poloidal movement of the neighborhood across the grid. Figure 4.14 shows the grid lines affected, the new structure of the *Grid Lines Buffer* and attempts to explain the new data flow. The rest of the system remains as is, except for *Write-Back* which now notifies the toroidal *Grid Lines Buffer* of the line being written in memory at any given time, in case the buffer needs it for its *Poloidal Line Buffers*.

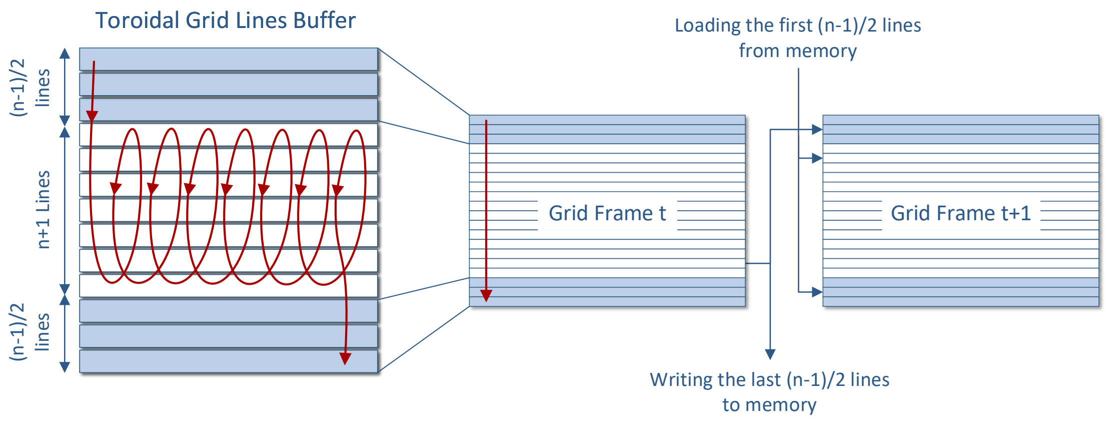

Toroidal Grid Lines Buffer
(n-1)/2 lines
n+1 Lines
(n-1)/2 lines
Loading the first (n-1)/2 lines from memory
Grid Frame t
Grid Frame t+1
Writing the last (n-1)/2 lines to memory

FIGURE 4.14. Toroidal Grid Lines Buffer.

4.2.10 User Input: Simulation Speed and Image Extraction

The simulation of cellular automata is executed in real time, which translates into 60 frames per second. Real-time simulations are useful in accelerating the evolution of cellular automaton patterns which require hundreds of generations in order to form and converge into a stable equilibrium. However, once the simulation has reached that point, slowing it down can help us notice useful details. Our system provides the user with the option to adjust the simulation speed at any time during the process by choosing between two speed settings: 1 or 60 automaton generations per second.

The user can also extract a bitmap image of the cellular automaton simulation's current state. The image extraction process consists of the same steps as the *System Initialization* process described in section 4.2.3, but in reverse order.

4.3 Automatic Code Adjustment

In this chapter we introduced certain variables which were used to describe the system's inner structure dimensions and resources. These variables are not abstract theoretical entities but are actually used to determine the properties of the design each time it is to be synthesized.

The aforementioned variables are manifested in VHDL in the form of *generics*. *Generics* allow the designer to parametrize the entity's structure and behavior. In our case, the variables we mentioned so far are found in the design's top-level VHDL file and can be set by the user at will, depending on the cellular automaton simulation's characteristics. The top-level VHDL file's *generics*:

```
ENTITY TOP_LEVEL IS
      GENERIC (
            GRID_X : INTEGER := 1920; -- NUMBER OF CELLS IN A LINE
            GRID_Y : INTEGER := 1080; -- NUMBER OF LINES
            CELL_SIZE : INTEGER := 8; -- 4 OR 8
            -- CELL SIZE IN BITS
            -- CELL_SIZE = 4 => 2^4 = 16 STATES
            -- CELL_SIZE = 8 => 2^8 = 256 STATES
            NEIGHBORHOOD_SIZE : INTEGER := 29;
            -- NEIGHBORHOOD SIZE MUST BE AN ODD NUMBER >= 3
            GRID_TYPE : STRING  := "TOROIDAL";
            -- VALID VALUES: "RECTANGULAR", "CYLINDRICAL" AND "TOROIDAL"
            BURST_SIZE : INTEGER := 128; -- NUMBER OF BITS
            NUMBER_OF_BURSTS_PER_LINE : INTEGER := GRID_X/(BURST_SIZE/CELL_SIZE);
            -- CELL_SIZE = 4 => NUMBER_OF_BURSTS_PER_LINE = GRID_X * CELL_SIZE / BURST_SIZE = 60
            -- CELL_SIZE = 8 => NUMBER_OF_BURSTS_PER_LINE = GRID_X * CELL_SIZE / BURST_SIZE = 120
            PALETTE : STRING := "WINDOWS";
            -- VALID VALUES: "WINDOWS" AND "GRADIENT"
            -- APPLICABLE ONLY TO 4-BIT CELL RULES, 8-BIT RULES HAVE A FIXED BLACK-RED-WHITE
            -- GRADIENT PALETTE
            SPEED : INTEGER := 60;
            -- SPEED: EVERY N FRAMES => NEW GENERATION
            -- FOR EXAMPLE, OUR GRAPHICS HERE RUN AT 60 FPS.
            -- IF SPEED = 120 THEN WE HAVE A NEW FRAME @ 60/120 = 0.5 HZ
            MEMORY_ADDR_WIDTH : INTEGER := 27
            -- NUMBER OF ADDRESS BITS, DEPENDS ON THE MEMORY CONTROLLER
      );
      PORT (
            ...
```

The user only needs to set the values of the *generics* found in the top-level file. The subsystems' VHDL files inherit their generic values from the top-level file without any intervention by the user and adjust their internal structure and behavior accordingly. One of the most striking examples of this process is the dimensioning and allocation of the *Grid Lines Buffer*'s BRAM resources, which depend on n, the automaton's neighborhood size and c, the cell's size in bits:

```
GENERATE_LINE_BUFFERS: FOR I IN 0 TO NEIGHBORHOOD_SIZE GENERATE
-- 0 to NEIGHBORHOOD_SIZE-1 LINES + ONE FOR BUFFERING
      LINE_BUFFER_4:    IF (CELL_SIZE = 4) GENERATE
            LINE_BUFFER: LINE_BUFFER_4b
            PORT MAP(
                  clka => UI_CLK,
                  wea => WRITE_ENABLE(I),
                  addra => WRITE_ADDRESS(5 DOWNTO 0),
                  dina => DATA_IN,
                  clkb => CLK_200,
                  addrb => READ_ADDRESS,
                  doutb => LINE_DATA(0, I)
            );
      END GENERATE LINE_BUFFER_4;

      LINE_BUFFER_8:    IF (CELL_SIZE = 8) GENERATE
            LINE_BUFFER: LINE_BUFFER_8b
            PORT MAP(
                  clka => UI_CLK,
                  wea => WRITE_ENABLE(I),
                  addra => WRITE_ADDRESS,
                  dina => DATA_IN,
                  clkb => CLK_200,
                  addrb => READ_ADDRESS,
                  doutb => LINE_DATA(0, I)
            );
      END GENERATE LINE_BUFFER_8;
END GENERATE GENERATE_LINE_BUFFERS;
```

Memory addressing is also affected by *Generics*. Both *Write-Back* and *Graphics Data Loader* set and reset their address counters based on the values of y, the number of grid lines and b_l, the number of memory bursts per grid line.

Automatic code adjustment is a fundamental feature which allows our design to be called a framework. Once the user has prepared their version of the *Cellular Automaton Engine*, our design generates a system which can execute any cellular automaton simulation in real time without any additional intervention by the user, provided that the automaton's characteristics lie within the system's specifications.

4.4 Putting It All Together: Real-Time Execution

The ability of our system to simulate cellular automata in real time stems from the fact that, thanks to the *Grid Lines Buffer* and *Cellular Automaton Engine*'s sliding neighborhood window, loading a cell's neighborhood data into the processing unit is reduced from a O(n^2) problem into a O(1) task in terms of time (but with O(n^2) resources - i.e. level of parallelism). In addition, by virtue of the *Grid Lines Buffer*, each cell needs to enter the FPGA only once, which means that for each external memory access to read an input datum we have O(n^2) operations which are performed internally to the FPGA at a multi-TB internal bandwidth. Combined with the fact that the *Memory Controller* maximizes aggregate memory bandwidth, this results in a system capable of executing real-time cellular automaton simulations with each graphics frame displaying a new automaton generation.

As shown in fig. 4.15, the system loads, processes and stores a grid line during the time it takes the *Graphics Controller* to render a line on the screen. The communication required between the system's modules is minimal, with each part of the architecture having its own independent control unit based on b_l, the number of bursts per line and the number of lines that have been processed so far. This results in a versatile system architecture which the user can customize in an automated fashion without risking jeopardizing the system's synchronization integrity, in spite of the clock domain crossings taking place.

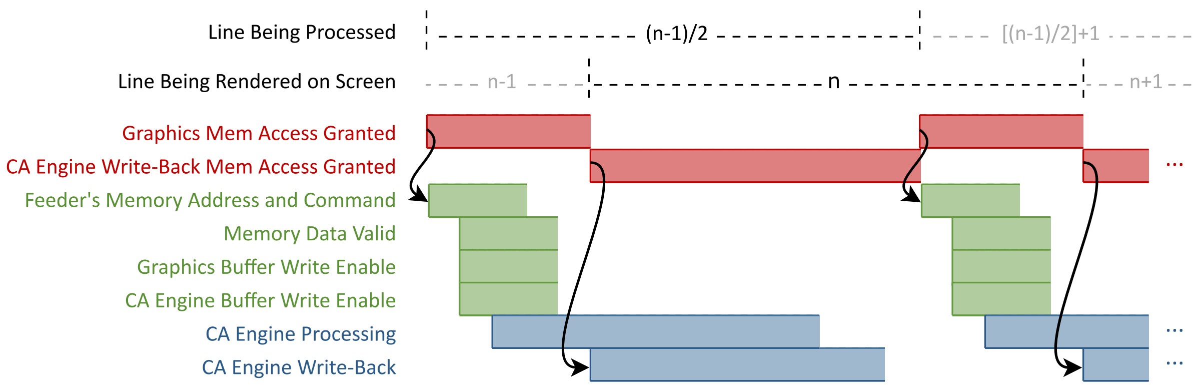

Line Being Processed
(n-1)/2
[(n-1)/2]+1
Line Being Rendered on Screen
n-1
n
n+1
Graphics Mem Access Granted
CA Engine Write-Back Mem Access Granted
Feeder's Memory Address and Command
Memory Data Valid
Graphics Buffer Write Enable
CA Engine Buffer Write Enable
CA Engine Processing
CA Engine Write-Back
...

FIGURE 4.15. The timing diagram of the system loading, processing and writing 1 line of the cellular automaton's grid.

Chapter 5

Applications and Usage Examples

As Toffoli stated in 1984, applying cellular automata modeling to real world problems is a challenging task, as one would have to experiment with various cellular automata configurations, as well as develop a solid theoretical background [17].

Large-neighborhood cellular automata have not been thoroughly explored, although their advanced modeling capabilities over automata with simple $r = 1$ neighborhoods have been proven in various fields like physics and chemistry as we will describe below.

This chapter will demonstrate our design's capabilities and reusability by simulating four different large-neighborhood cellular automata. The FPGA platform used to run the simulations was *Digilent's Nexys 4 DDR* board featuring *Xilinx's Artix 7* FPGA and a 128 MB DDR2 memory module. The *Artix 7* part mounted on board is the medium-sized *XC7A100T-1CSG324C* which contains 15850 logic slices (4 x 6-input LUTs and 8 flip flops each), 4860 Kbits of BRAM and 240 DSP slices. The VHDL design was synthesized and implemented using *Xilinx's Vivado 2018.1 Design Suite*.

The FPGA board was connected via USB to a computer running *Microsoft Windows 10* and via a VGA cable to a Full-HD computer monitor. All the cellular automata images used in this chapter have been extracted from simulations executed by the system described in this thesis.

5.1 Artificial Physics

The first cellular automaton rule simulated by our design is known as *Artificial Physics* and it is an outer totalistic cellular automaton with 2 cell states and a weighted, large neighborhood. The rule's name originates from its fascinating behavior (fig. 5.1). As the simulation moves on, "atoms" appear in the automaton's universe. As time goes by, these "atoms" attract each other and bind together forming "molecules".

Each cell of the grid can be either "alive" or "dead". The cellular automaton's initial state of the grid must have 1/7 of its cells alive and randomly distributed across the grid in order for atoms to appear, otherwise the sea of alive cells disappears within a few generations. A bitmap image of the initial state with these properties is easy to create with the use of tools like Matlab by filling a 1920×1080 array randomly with ones and zeroes.

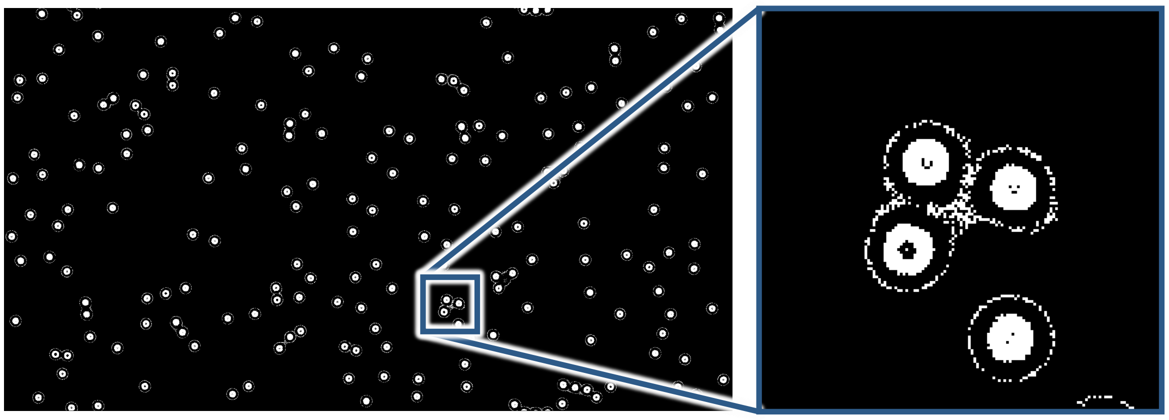


FIGURE 5.1. *Artificial Physics*, an outer totalistic cellular automaton with a large neighborhood.

The rule requires a quite large 21×21 neighborhood with binary weights as shown in fig. 5.2. The cell's state transition function is defined as:

$$c'(i,j) = \sum_{x=i-r}^{i+r} \sum_{y=j-r}^{j+r} w(x-i, y-j) \cdot c_t(x,y)$$

$$c_{t+1}(i,j) = \begin{cases} 0 & \text{if } \ c'(i,j) \leq 19 \\ 1 & \text{if } \ 20 < c'(i,j) \leq 23 \\ 0 & \text{if } \ 24 < c'(i,j) \leq 58 \\ 1 & \text{if } \ 59 < c'(i,j) \leq 100 \\ 0 & \text{otherwise} \end{cases}$$

As we mentioned in chapter 4, the user chooses the neighborhood size and the automaton's cell size in bits, and the system is automatically generated based on these parameters. The only part of the design the user needs to edit by hand is the *Cellular Automaton Engine* whose operation is determined by the rule's transition function. The system's parameters used for this simulation were $n = 21$ and $c = 4$ bits, and the grid was toroidal.

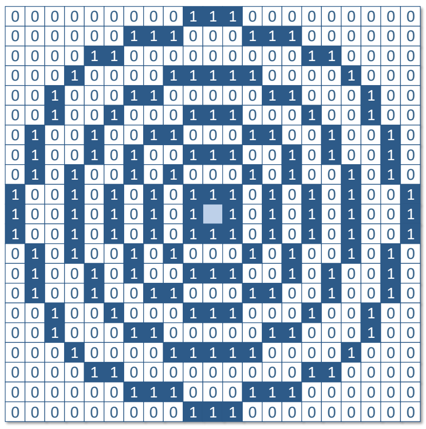


FIGURE 5.2. The neighborhood of *Artificial Physics*.

5.2 The Greenberg-Hastings Model

The second cellular automaton simulation example is an excitable media model called the *Greenberg-Hastings* model. Excitable media are non-linear dynamical systems which are able to support the propagation of excitation waves, which usually appear periodically after a certain period of time called refractory time. Forest fires and chemical reaction-diffusion systems are two well known examples of excitable media.

Excitable media are often described by cellular automata as a simpler replacement for complex differential equations. The *Greenberg-Hastings* model is one of the simplest cellular automata to model excitable media [35]. It originally had a von Neumann neighborhood with radius $r = 1$ and 3 cell states: "quiescent", "excited" and "refractory", but it was later expanded to support more states and larger neighborhoods [36, 37].

Excitable cellular automata usually generate patterns like spirals or horns and converge into an excitation equilibrium. However, this is not guaranteed when starting from a randomly-filled grid, and the initial grid condition for such rules is often explored empirically. In the original form of the *Greenberg-Hastings* model, there are numerous initial conditions which lead to periodic behavior [38]. Such initial conditions are proven to exist in the extended version of the rule as well, where the width of the waves has also been proven to be proportional to the neighborhood's size [36].

For this simulation example we used a toroidal grid, a 29×29 von Neumann neighborhood and 16 cell states, thus $n = 29$ and $c = 4$. A cell can be "quiescent" (state 0), "excited" (state 1) or in a sequence of "refraction" (states 2 to 15). The cell's state transition function is defined as:

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

As shown in fig. 5.3, our simulation of the *Greenberg-Hastings* cellular automaton has reached an excitation equilibrium with numerous oscillating horns producing waves. By using a circular neighborhood instead of a von Neumann neighborhood, the patterns formed are heavily affected by the change and the waves and vortices become curved, which is an interesting qualitative result in its own right.

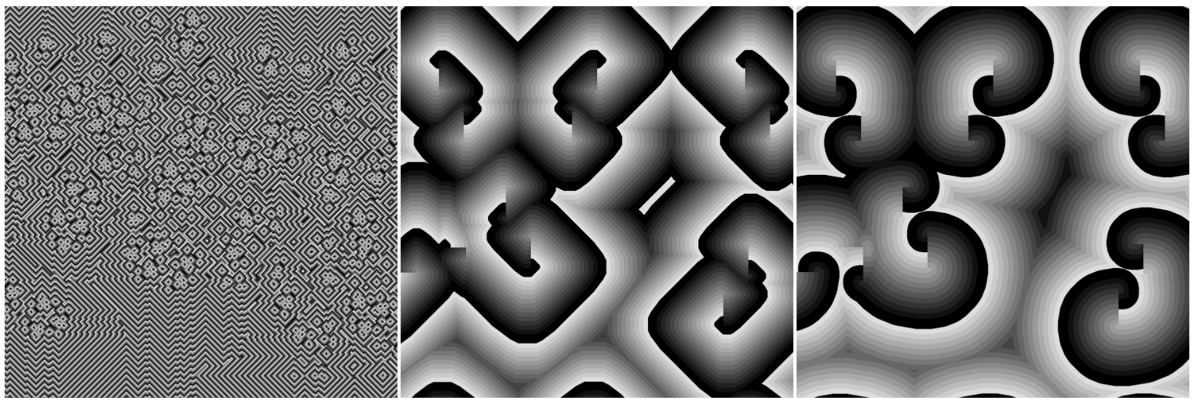


FIGURE 5.3. The *Greenberg-Hastings* model after 500 iterations with a 3×3 von Neumann, a 29 × 29 von Neumann and a 29 × 29 circular neighborhood respectively. The three images above are of the same resolution, with each pixel representing a cell of the automaton's grid.

5.3 The Hodgepodge Machine

The third example of our system's applications is the simulation of the *Hodgepodge Machine*. The *Hodgepodge Machine* models excitable media in a more complex fashion than that of the *Greenberg-Hastings* cellular automaton and is often used to model the Belousov-Zhabotinsky chemical reaction [39, 40]. Originally the *Hodgepodge Machine* had a 3 × 3 Moore neighborhood, but large-neighborhood versions of the rule have been explored to some extent during the last decade. The excitation equilibrium, also known in cellular automata as auto-organization, is also found empirically like in the *Greenberg-Hastings* rule and depends on the rule's parameters as well as the initial state of the grid.

For this example we used a 29×29 Moore neighborhood and 256 cell states, thus $n = 29$ and $c = 8$. A cell can be "healthy" (state 0), "ill" (state 255) or in a sequence of "infection" (states 2 to 254). The cell's state transition function is defined as:

$$c_{t+1}(i,j) = \begin{cases} \text{number of infected and ill cells } /k & \text{if } c_t(i,j) = 0 \\ 0 & \text{if } c_t(i,j) = 255 \\ (\text{sum of cells / sum of infected cells}) + g & \text{otherwise} \end{cases}$$

All the cells mentioned in the transition function concern the cells located in $c_t(i,j)$'s neighborhood, and k and g are the two parameters of the rule that determine when and how fast the "infection" will spread.

As shown in fig. 5.4 our simulation of the *Hodgepodge Machine* has reached an excitation equilibrium with numerous horns and spirals producing waves. Note that the waves seem to consist of only four kinds of cells forming four wide cell bands. However, that is not the case. The noticeable gradient "glow" appearing between those four bands of cells is not a result of image processing, but is actually created by the cellular automaton itself and its many different cell states. We also notice that the vortices produced co-exist with small, stable, vortex-like patterns located in the center of the larger vortices. This phenomenon was not present in any of our earlier experiments with smaller neighborhood sizes reaching up to 19×19 cells. The automaton's parameters used for this simulation were $k = 5$ and $g = 105$. The grid was toroidal and initially filled with $n \times n$ sized tiles of random numbers.

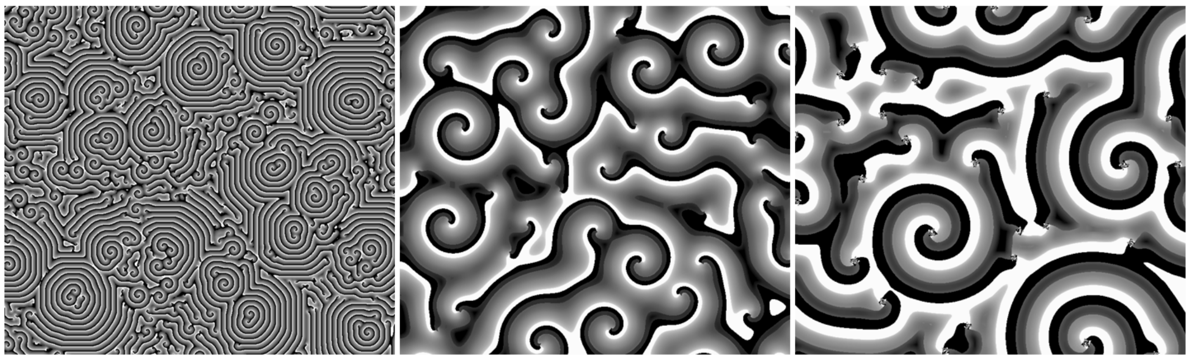
5.4 Anisotropic Rules

The fourth and final application example is an anisotropic cellular automaton with a large, 29×29 neighborhood. The anisotropy of cellular automata lies either in the anisotropy of the grid, the anisotropy of the neighborhood or both. In this example, we experimented with the neighborhood's anisotropy. Section 7.2.3 describes how to implement the anisotropy of the grid as well, either by packing relevant information within the data of each grid cell, or by using counters to calculate the coordinates of each cell within the frame.

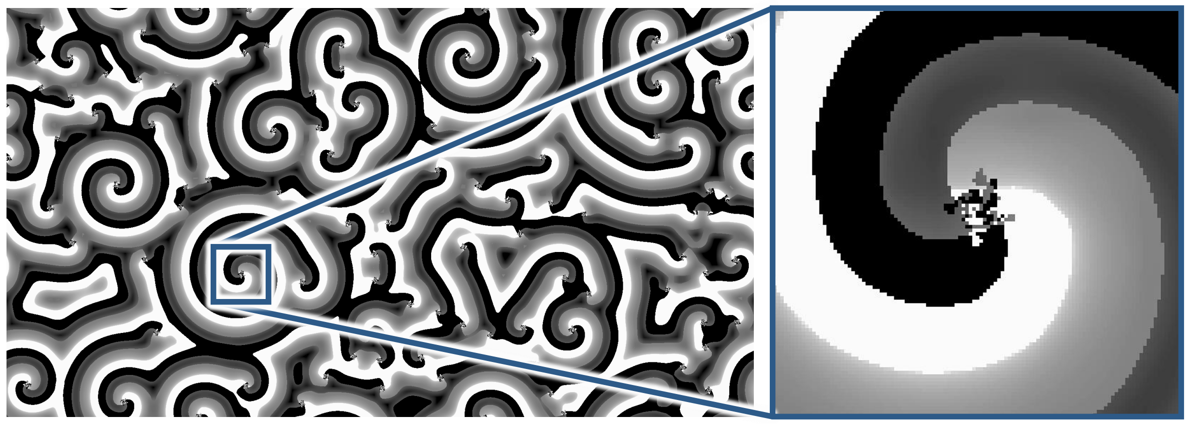
In this example, our anisotropic neighborhood contains the largest weights at its far right (eastern) edge, with the weight value gradually being reduced to 1 towards the far left (western) edge of the neighborhood. The cell's state transition function is defined as:

$$\text{weighted sum} = \sum_{x=i-r}^{i+r} \sum_{y=j-r}^{j+r} w(x-i, y-j) \cdot c_t(x,y)$$

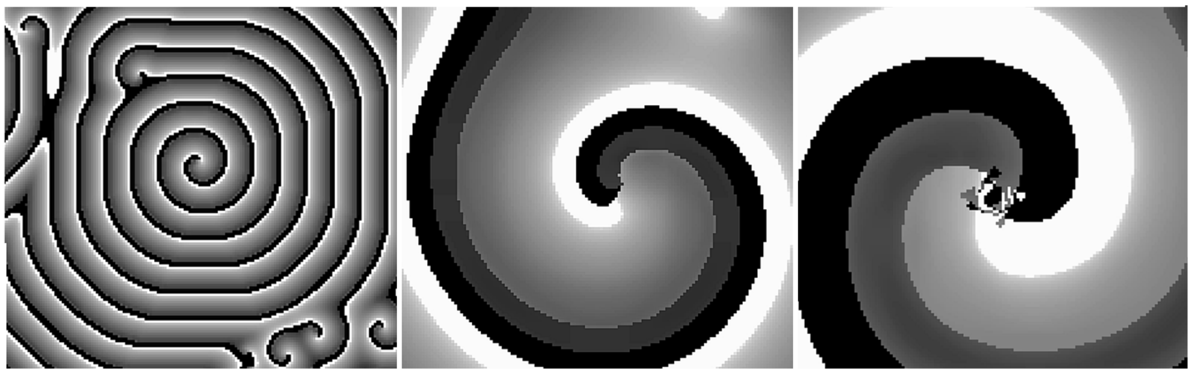
$$c_{t+1}(i,j) = \begin{cases} c_t(i,j) - 1 & \text{if weighted sum} > \text{threshold} \\ c_t(i,j) + 1 & \text{if weighted sum} < \text{threshold} \\ c_t(i,j) & \text{otherwise} \end{cases}$$



(a)



(b)



(c)

FIGURE 5.4. The *Hodgepodge Machine* with a 29 × 29 Moore neighborhood produces interesting, unprecedented results which are not produced with smaller neighborhoods. (a) The *Hodgepodge Machine* after 500 iterations with a 3×3, a 19×19 and a 29×29 Moore neighborhood respectively. The three images above are of the same resolution, with each pixel representing a cell of the automaton's grid. (b) The use of 29×29 neighborhoods results in large vortices with stable core patterns. (c) The stable vortex cores are not formed with smaller neighborhoods, such as the 3×3 or the 19×19 Moore neighborhood.

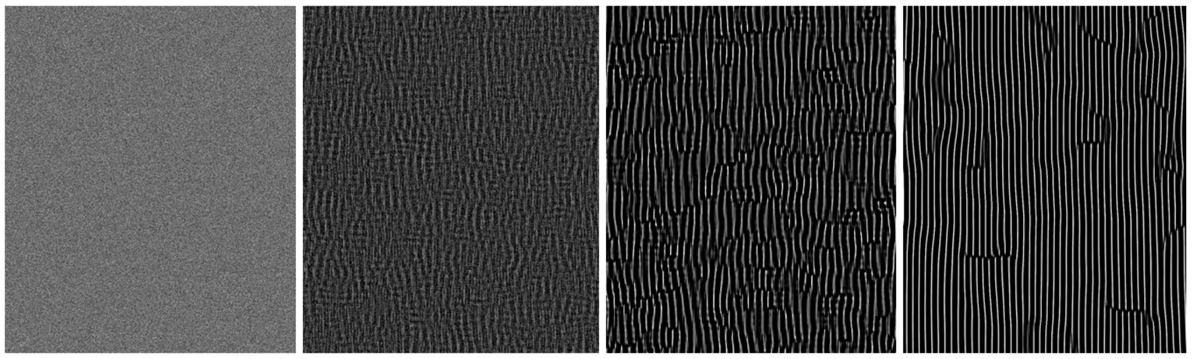


FIGURE 5.5. The self-organization properties of a simple anisotropic cellular automaton with a large 29 × 29 Moore neighborhood after 1, 120, 500 and 10000 generations.

This quite simple anisotropic rule exhibits interesting self-organization properties as shown in fig. 5.5. Starting from a randomly filled grid, the automaton cells form long, thin stripes after a few thousand generations. In the course of time, ripples appear and tend to propagate to the left by virtue of the anisotropic neighborhood's weights. The long, rope-like structures become more coherent once the rules allow for a ripple to disappear. As a result, the two edges of the rope merge into one, creating a ring around the toroidal grid.

For this anisotropic cellular automaton example we used a 29 × 29 Moore neighborhood and 256 cell states, thus $n = 29$ and $c = 8$. The grid was toroidal and randomly filled using a uniform distribution $U\{0,255\}$.

5.5 Weighted Neighborhoods

The rules simulated above utilize quite large neighborhoods, but they do not have any computationally demanding weights to be calculated. Large-neighborhood cellular automata usually use binary weights that enable or disable a cell, like in *Artificial Physics*, or normalized weights that raise the significance of the cells located closer to the center of the neighborhood.

In order to test and ensure that our design can support weighted neighborhoods, we applied random 4-bit weights to the *Hodgepodge Machine*'s 29 × 29 neighborhood, which translates into 841 multipliers operating in parallel. Our medium-sized FPGA could fit all of them without jeopardizing the system's real time characteristics, which means that routing was successful and all the timing constraints were met. However, if a larger FPGA is used or if numerous weighted cell values can be calculated only with the use of shift operations, one can use larger weights and multipliers.

The results of synthesis and implementation of the aforementioned simulations, as well as those of the weighted-neighborhood version of the *Hodgepodge Machine*, will be presented and discussed in chapter 6 that follows.

Chapter 6

Design Verification and Performance Results

This chapter is composed of two parts. In the first part, we will present the method followed to verify the correct operation of our work. Finally, in the second part we will discuss our design's performance results, both in terms of computation speed and resource allocation.

6.1 Design Verification

In order to verify the correct operation of our implemented design, we need to ensure the satisfaction of its real-time constraints as well as the validity of the produced results. Two different tools were used during the verification process:

1. The post-place and route simulation.
2. *Xilinx's Integrated Logic Analyzer* monitoring each actual run on *Digilent's Nexys 4 DDR* FPGA system.

Along with the process of synthesis, placement and routing, *Vivado 2018.1* creates a simulation model of the design for each one of those three stages. The post-place and route simulation model is the most accurate model of the implemented design. It contains the true timing delay information of the implemented design and can be used to verify its timing as well as its functionality.

The *Integrated Logic Analyzer* is an IP core provided with *Vivado 2018.1* by *Xilinx*. It can be adapted to the implemented design and be used to monitor its internal signals while using the actual FPGA device.

Both tools were used to verify the validity of the system's arithmetic results in accordance with the results produced by the software implementation. The *Integrated Logic Analyzer* was also used to ensure that the real-time constraints of the implemented design were met in all cases. In fig. 6.1 we can see a line of the *Hodgepodge Machine*'s grid being processed in time. The *Integrated Logic Analyzer*'s waveform and the system's timing diagram described in chapter 4 are very much alike.

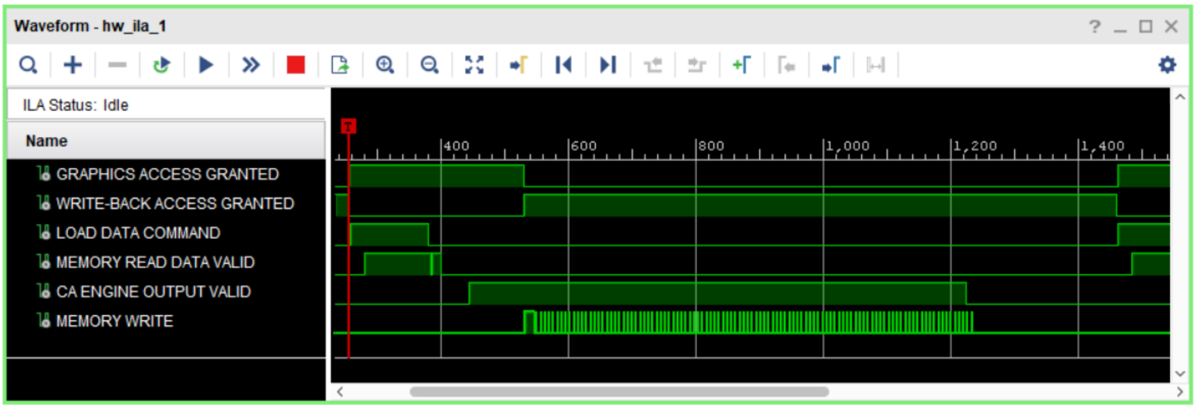


FIGURE 6.1. Timing verified with the use of Xilinx's Integrated Logic Analyzer.

6.2 Performance Results

In terms of results, our system calculates 60 cellular automaton generations per second regardless of the rule being simulated. The grid size is 1920 × 1080 cells which gives us 1920 × 1080 × 60 = 124416000 cells per second. As shown below, our system successfully surpasses the computation and memory bounds found in a general purpose CPU and has a measured speedup of up to 51× against an *Intel Core i7-7700HQ* CPU (1 core) running highly optimized (-O3) software programmed in C. The table shows the result average from multiple runs for each cellular automaton simulation.

Cellular Automaton	**i7-7700HQ, 1000 generations**	**Our Design, 1000 generations**	**Our Design's Speedup**
Artificial Physics, n = 21	538.77 sec	16.67 sec	32.32×
Greenberg-Hastings, n = 29	469.58 sec	16.67 sec	28.16×
Hodgepodge Machine, n = 29	851.29 sec	16.67 sec	51.06×

TABLE 6.1. Comparative results: execution time and speedup.

The speedup offered by contemporary GPUs is comparable to our design's speedup for similar cellular automata rules [41, 42], however, a GPU consumes at least 10 times more power than an FPGA. At the system level, a contemporary GPU requires 170 W, whereas our *Digilent's Nexys 4 DDR* board is powered by USB, i.e. maximum power supply of 7.5 W, so, for similar speeds the power consumption is better than one-tenth and hence the total energy for the computation scales accordingly. Table 6.2, below, shows the current work, recently published work on GPUs, as well as the older, classic FPGA-based *CAM* architectures.

Architecture	**Neighborhood Size**	**Performance**
Margolus, 1993-2001, CAMs	experimented with up to 11×11	10 gen./sec for a 512×512 grid with 3-bit cells
Gibson et al., 2015, Workstation with Nvidia GTX 560 Ti	experimented with up to 11×11	≈ 65× over serial for Game of Life on a 2048×2048 grid
Millan et al., 2017, Nvidia TitanX GPU	experimented with up to 11×11	21.1× over serial for Game of Life on a 4096×4096 grid
Current work, 2019, Artix-7 FPGA	experimented with up to 29×29	51× over serial for the Hodgepodge Machine on a 1920×1080 grid

TABLE 6.2. Large-neighborhood cellular automata implementations on hardware.

Some indicative results of the resource utilization after the implementation of the aforementioned cellular automata can be seen in fig. 6.2. The amount of FPGA resources utilized for each rule differs and depends mainly on the cellular automaton's neighborhood and cell size. As the size of the neighborhood grows, the number of the BRAM modules rises as well. As the cell's size grows, so does the size of each BRAM module. Another important cellular automaton feature that also determines the amount of the required resources is the type of the automaton's grid. The toroidal grid requires more BRAM resources in order to implement the poloidal wrap-around.

A large part of the LUT and FF resources required is determined by the rule's complexity itself and has nothing to do with the framework's design, as we can see for example in the case of the *Hodgepodge Machine*. The two designs of the *Hodgepodge Machine* are identical, with the only difference being the addition of the neighborhood weight multipliers. In addition to the resource utilization differences depicted in fig. 6.2, in fig. 6.3 you can see the difference between the placement of the two implementations.

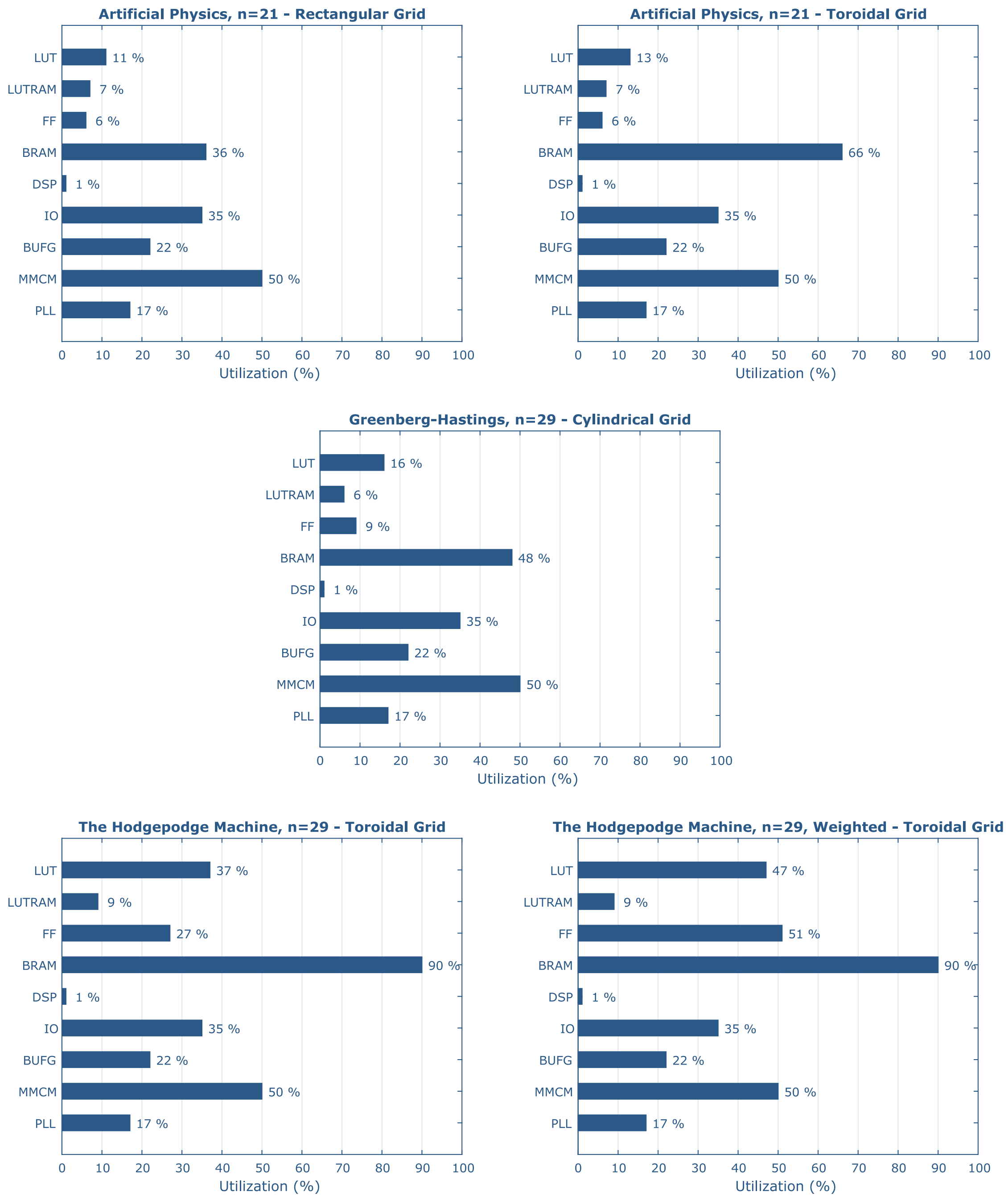


FIGURE 6.2. Bar charts depicting the FPGA resource utilization for different cellular automata implementations.

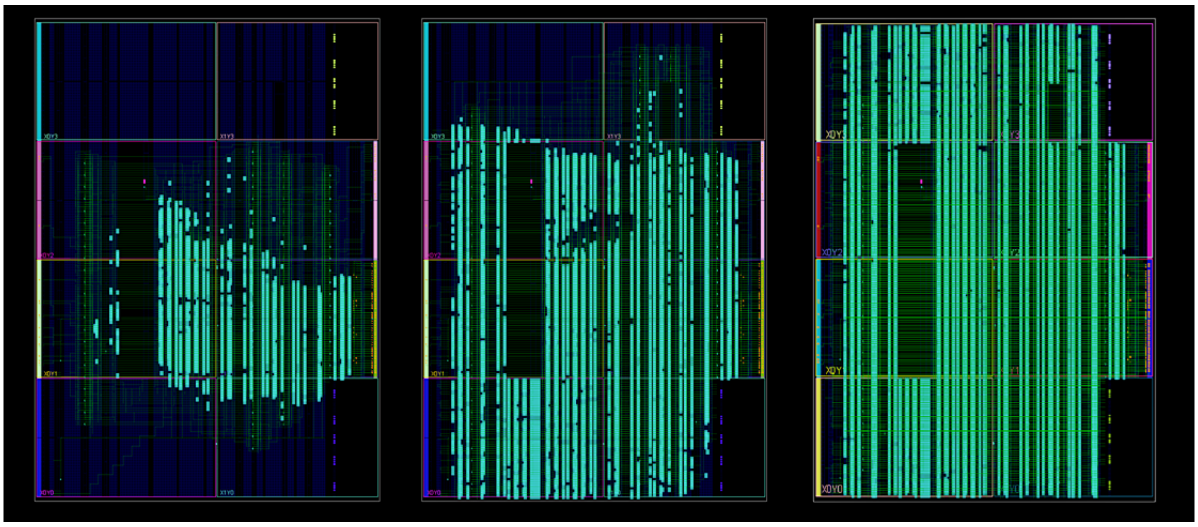


FIGURE 6.3. The implementations of Artificial Physics ($n = 21$), the Hodgepodge Machine ($n = 29$), and the weighted Hodgepodge Machine ($n = 29$) using *Xilinx's Artix 7*, a medium-sized FPGA chip.

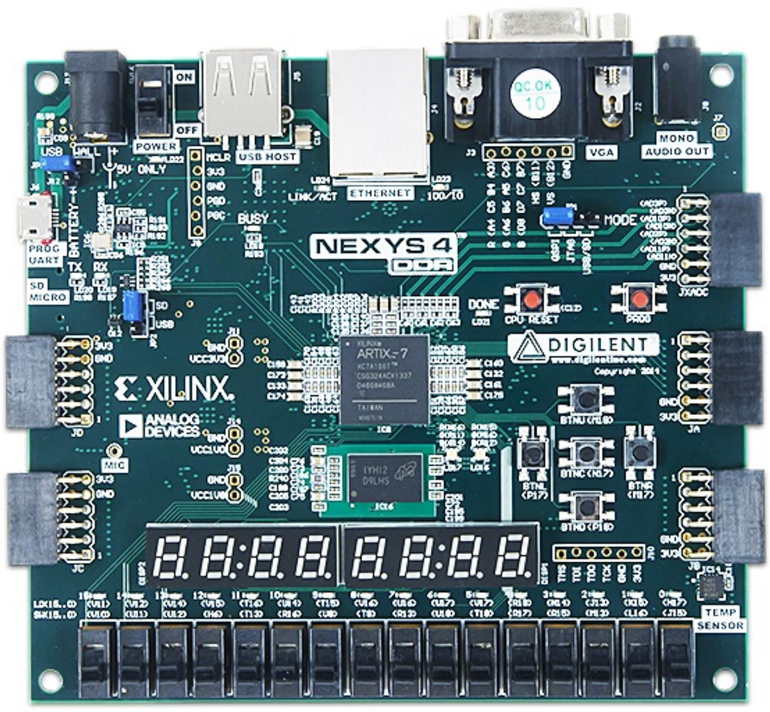


FIGURE 6.4. *Digilent's Nexys 4 DDR* board, the FPGA platform used to run the cellular automata simulations.

Chapter 7

Conclusion and Future Work

The last chapter of this thesis summarizes the accomplishments of this work and discusses future work directions and ideas arising from it.

7.1 Conclusion

In this thesis project we developed a parallel framework on reconfigurable logic which can be used to efficiently simulate large-neighborhood cellular automata in real time. Simulating cellular automata rules with large neighborhood sizes on large grids provides a new aspect of modeling physical processes with realistic features and results. In addition, it helps the user save precious time by virtue of its ability to automatically perform the process of resource dimensioning, allocation, interconnection and synchronization. The proposed design yields significant speedup compared to a high-end general-purpose CPU and its potential leaves plenty of room for further exploration of the maximum range of its capabilities.

7.2 Future Work

In this section we will present an outlook for future work, which contains potential improvements of the design as well as its prospective capabilities as far as the simulation of cellular automata is concerned. The ideas described have also reference to reconfigurable logic in general and its capacity to take cellular automata simulations a step further.

7.2.1 A More Compact Toroidal Grid Lines Buffer

The toroidal *Grid Lines Buffer* was an unplanned addition to this thesis project. It was implemented as shown in chapter 4 with the purpose of taking a step further into creating a more versatile general-purpose cellular automata machine. By adding the *Poloidal Line Buffers* in the already existing *Grid Lines Buffer* and applying a few synchronization tweaks, we managed to preserve the real-time characteristics of the system and have a toroidal grid option for any cellular automaton simulation. However, the BRAM resources required can be further reduced.

By using $(n-1)/2$ regular *Grid Line Buffers* as *Poloidal Line Buffers* at the start of every frame, we can reduce the amount o BRAM resources required down to $[(n+1)+(n-1)/2] \times x \times c$ bits $= [(n+1)+(n-1)/2] \times b_l \times b$ bits. This modification would require a more complex control system, since the size of the cell's neighborhood n and the frame's y dimension determine which *Grid Line Buffers* will be taking upon the role of the *Poloidal Line Buffers* during any given simulation.

7.2.2 Programmable System

While in general our design is automatically generated based on the user's preferences, the *Cellular Automaton Engine*'s function depends specifically on the cellular automaton's rule and must be redesigned every time according to the application.

This process requires that the user is familiar with a hardware description language (HDL) like VHDL. However, it is not that difficult to turn our framework into a complete general-purpose cellular automata design with the use of a high-level language which can generate an HDL description of the *Cellular Automaton Engine*. The structure and function of most cellular automata rules can be described with a high-level language and be translated into an HDL as we saw earlier in chapter 3.

In case we want to turn our framework into an as-is programmable system, we can implement a system capable of operating at its maximum capacity and then let the user program it and choose between the features they need for their application. For example, the system will always simulate the largest neighborhood possible based on the FPGA resources available, but smaller neighborhoods can be simulated at will by setting the value of multiple weights to zero. All the programmable registers and multiplexers of the *Cellular Automaton Engine* can be programmed by the user during the initialization process. Turning the design into an as-is programmable system means that the time-consuming processes of synthesis and implementation have to be executed only once per FPGA chip.

7.2.3 Anisotropy and Multiple-Neighborhood Automata

In multiple-neighborhood cellular automata each cell of the grid uses an arbitrary number of different neighborhoods, one at each time interval between two cellular automaton generations.

Multiple-neighborhood cellular automata produce interesting patterns of immense complexity thanks to their alternating neighborhood mechanism. They could implemented in hardware as shown in fig. 7.1, with a ring counter being used to select one neighborhood at a time. By using a linear-feedback shift register (LFSR) to choose between similar neighborhoods, one could simulate noise or isotropy in cellular automata.

Multiple neighborhoods can also be applied to simulate anisotropy, which implies different properties in different parts of the automaton's grid. In this case part of the cell's data could hold information relevant to each specific cell, for example it can refer to the cell's coordinates on the grid which in turn can be used to select the corresponding neighborhood configuration.

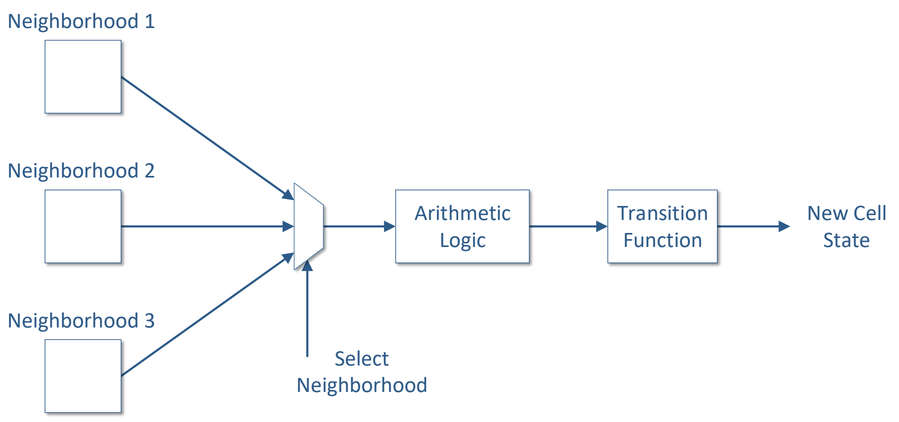


FIGURE 7.1. A way of implementing multiple-neighborhood cellular automata and anisotropy in hardware.

7.2.4 High-Order Cellular Automata

The idea that a cell's data could hold more information than just its state can help us simulate other classes of cellular automata as well. A distinctive example would be high-order cellular automata, a class of automata invented by Edward Fredkin while investigating the reversibility of cellular automata [43]. In high-order cellular automata a cell's state depends not only on its neighborhood, but also on its state at previous generations. In hardware, a cell's data can hold a history of the cell's states along with its current state. In this way the *Cellular Automaton Engine* actually loads and processes a complete past timeline of previous generations without any significant changes to the original hardware prototype design.

7.2.5 Other Types of Grids and Multidimensional Cellular Automata

During this thesis we have been referring to the rectangular grid as a grid of cells which has the shape of a rectangle as a whole. However, in theory we do not care about the shape and boundaries of the theoretically infinite grid, unless we want to implement it with finite resources such as a computer's available memory. In cellular automata literature a rectangular grid is the grid consisting of rectangular cells, like the one implemented in this thesis project.

While the rectangular grid is the most common one, it is not the only type of grids found in cellular automata. The triangular grid and the hexagonal grid are two types of grids with many applications in cellular automata. However, the rectangular grid remains the optimal way of representing a cellular automaton's grid in hardware, mostly because of its efficient representation in memory. Thus, the most efficient way to represent a triangular or a hexagonal grid in hardware would be to map it on a rectangular grid, which is a trivial procedure described in fig. 7.2.

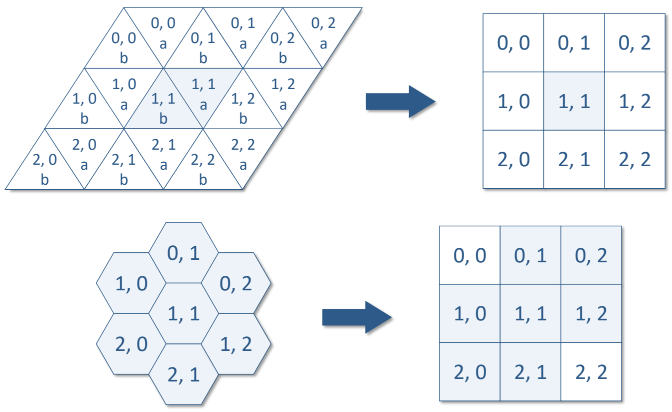


FIGURE 7.2. Mapping a triangular and a hexagonal grid on a rectangular grid.

The shape of the grid's cell is not the only property defining the grid. The other significant property which defines a cellular automaton's space is the number of its dimensions. This thesis project dealt with the simulation of 2D cellular automata, however, our design's performance provides an indication for its potential performance while simulating 3D cellular automata. Our system has been succesfully tested on simulating neighborhood sizes of up to 29×29 cells in real time on a medium-sized FPGA, which means it can process 841 cells per clock cycle. It also loads, processes and writes back $1920 \times 1080 = 2073600$ cells per frame. The aforementioned values can also fit a 3D cellular automaton with a $9 \times 9 \times 9$ neighborhood evolving on a 3D grid $100 \times 100 \times 207$ cells large.

7.2.6 Floating-Point Arithmetic

The general outline of our framework can be applied to many other, more complicated cellular automata rules like continuous cellular automata. This particular class of automata consists an effort to generalize and extend cellular automata so that a cell's state is not discrete but continuous [44].

Continuous cellular automata have applications of significant importance in the field of artificial life. Generalizing *Conway's Game of Life* to a continuous domain has produced fascinating rules like *Smooth Life* and, most recently, *Lenia* which generate highly complex patterns that resemble biological organisms [45, 46]. This special class of automata has also been successfully used to accurately model physical phenomena like lattice gases, thermodynamics and chemical oscillators. In our opinion large-neighborhood continuous cellular automata can fill the gap between stencil computing and computational fluid dynamics.

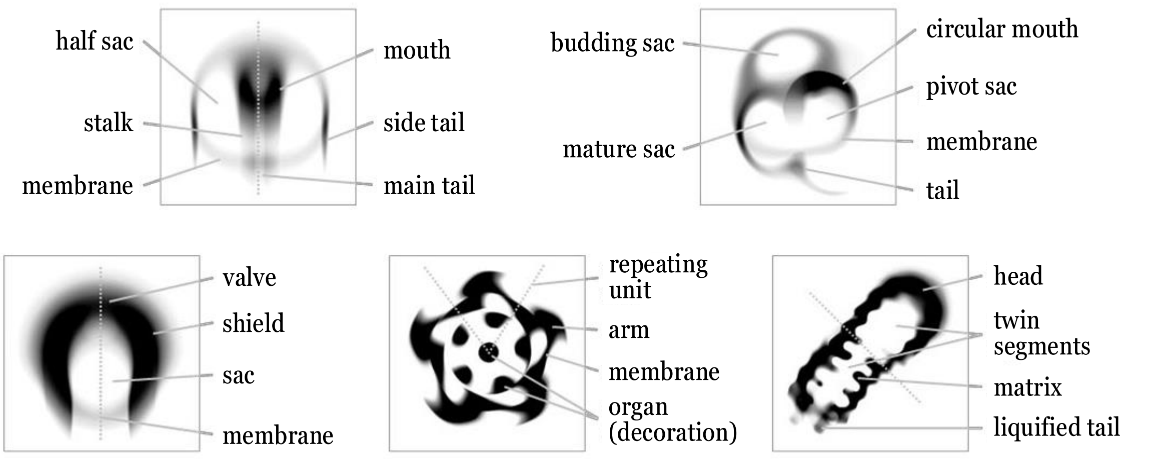


FIGURE 7.3. Complex patterns appearing in Lenia, a continuous cellular automaton.

In order for hardware to be able to simulate continuous cellular automata, the support of floating point arithmetic is required. As one can see in fig. 6.1, which depicts the timing of our design processing a grid line of an 8-bit version of the *Hodgepodge Machine*, the system has plenty of spare time to load and process even more data than it already does. Even though *Write-back* is fully pipelined, the incoming data rate is so slow that most of the time it writes a memory burst every 5 to 6 clock cycles. The aforementioned performance is determined by c, the cell's size in bits, which in turn affects c_b, the number of cells per memory burst, and as a result our system might be already capable of supporting 16-bit cells. In addition, the framework's relatively low resource utilization leaves plenty of room for the implementation of floating-point arithmetic logic within the *Cellular Automaton Engine*.

These observations lead to the conclusion that our cellular automata framework can potentially support the real-time simulation of continuous cellular automata using half-precision floating point arithmetic without any additional intervention. However, in order to raise the size of the cell to 32 or 64 bits, one would have to adjust our framework accordingly and make it run at higher frequencies so that it successfully loads and processes that amount of data in time. Since the maximum frequencies that an FPGA design can reach today are much higher than our implementation's 200 MHz, we are convinced that our framework's general outline can be successfully applied to implement continuous cellular automata in hardware using floating point arithmetic.

APPENDIX A

PUBLICATIONS STEMMING FROM THIS THESIS

The following conference publications originated from this thesis:

[1] N. KYPARISSAS and A. DOLLAS, *An FPGA–based Architecture to Simulate Cellular Automata with Large Neighborhoods in Real Time*, in: *International Conference on Field Programmable Logic and Applications (FPL '19)*, Barcelona, Spain, September 2019.

[2] N. KYPARISSAS and A. DOLLAS, *Field Programmable Gate Array Technology as an Enabling Tool Towards Large–Neighborhood Cellular Automata on Cells with Many States*, in: *International Conference on High Performance Computing and Simulation (HPCS '19)*, Dublin, Ireland, July 2019.

BIBLIOGRAPHY

[1] S. ULAM, *Random Processes and Transformations*, in: *International Congress of Mathematicians*, Cambridge, 1950.

[2] J. von NEUMANN, *The General and Logical Theory of Automata*, Cerebral Mechanisms in Behavior: The Hixon Symposium, John Wiley & Sons (1951).

[3] J. B. SALEM and S. WOLFRAM, *Thermodynamics and Hydrodynamics with Cellular Automata*, Theory and Applications of Cellular Automata, World Scientific (1986).

[4] D. H. ROTHMAN and S. ZALESKI, *Lattice-Gas Cellular Automata: Simple Models of Complex Hydrodynamics*, Cambridge University Press, 2004.

[5] P. HOGEWEG, *Cellular Automata as a Paradigm for Ecological Modeling*, Applied Mathematics and Computation 27.1 (1988).

[6] F. GERS, H. de GARIS, and M. KORKIN, *CoDi-1Bit : A Simplified Cellular Automata Based Neuron Model*, Lecture Notes in Computer Science 1363 (1998).

[7] K. ZUSE, *Calculating Space*, MIT Technical Translation AZT-70-164-GEMIT, Massachusetts Institute of Technology (Project MAC), 1970.

[8] J. von NEUMANN and A. W. BURKS, *Theory of Self-Reproducing Automata*, University of Illinois Press, 1966.

[9] A. W. BURKS, *Essays on Cellular Automata*, University of Illinois Press, 1971.

[10] T. TOFFOLI, *Computation and Construction Universality of Reversible Cellular Automata*, Journal of Computer and System Sciences 15.2 (1977).

[11] M. MITCHELL, *Computation in Cellular Automata: a Selected Review*, Non-Standard Computation, Wiley-VCH Verlag in Weinheim (1998).

[12] A. ILACHINSKI, *Cellular Automata: a Discrete Universe*, World Scientific, 2001.

[13] M. GARDNER, *Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game "Life"*, Scientific American 223.4 (1970).

[14] E. R. BERLEKAMP, J. H. CONWAY, and R. K. GUY, *Winning Ways for Your Mathematical Plays*, A K Peters Ltd., 2001.

[15] P. W. RENDELL, *A Universal Turing Machine in Conway's Game of Life*, in: *International Conference on High Performance Computing and Simulation*, 2011.

[16] W. GOSPER, *Exploiting Regularities in Large Cellular Spaces*, Physica D: Nonlinear Phenomena 10.1-2 (1984).

[17] T. TOFFOLI, *CAM: A High-Performance Cellular-Automaton Machine*, Physica D: Nonlinear Phenomena 10.1-2 (1984).

[18] T. TOFFOLI and N. MARGOLUS, *Cellular Automata Machines - A New Environment for Modeling*, MIT Press, 1987.

[19] N. MARGOLUS, *CAM-8: A Computer Architecture Based on Cellular Automata*, Pattern Formation and Lattice-Gas Automata, AMS (1993).

[20] N. MARGOLUS, *An FPGA Architecture for DRAM-Based Systolic Computations*, in: *5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97)*, 1997.

[21] N. MARGOLUS, *An Embedded DRAM Architecture for Large-Scale Spatial-Lattice Computations*, in: *27th Annual International Symposium on Computer Architecture (ISCA '00)*, 2000.

[22] R. HOFFMANN, K. VOLKMANN, and M. SOBOLEWSKI, *The Cellular Processing Machine CEPRA-8L*, Mathematical Research 81 (1994).

[23] C. HOCHBERGER et al., *The CEPRA-1X Cellular Processor*, Reconfigurable Architectures: High Performance by Configware, IT Press, Bruchsal (1997).

[24] C. HOCHBERGER et al., *The Cellular Processor Architecture CEPRA-1X and its Configuration by CDL*, in: *IPDPS 2000 Workshops, LNCS 1800*, 2000.

[25] P. SHAW, P. COCKSHOTT, and P. BARRIE, *Implementation of Lattice Gases Using FPGAs*, Physica D: Nonlinear Phenomena 12.1 (1996).

[26] T. KOBORI, T. MARUYAMA, and T. HOSHINO, *A Cellular Automata System with FPGA*, in: *9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, 2001.

[27] K. BOUAZZA et al., *Implementing Cellular Automata on the ArMen Machine*, in: *2nd International Workshop on Algorithms and Parallel VLSI Architectures*, 1992.

[28] G. CAPPUCCINO and G. COCORULLO, *Custom Reconfigurable Computing Machine for High Performance Cellular Automata Processing*, Electronic Engineering Times (www.eetimes.com, TechOnLine Publication) (2001).

[29] S. MURTAZA, A. G. HOEKSTRA, and P. M. A. SLOOT, *Performance Modeling of 2D Cellular Automata on FPGA*, in: *International Conference on Field Programmable Logic and Applications (FPL '07)*, 2007.

[30] S. MURTAZA, A. G. HOEKSTRA, and P. M. A. SLOOT, *Floating Point Based Cellular Automata Simulations Using a Dual FPGA-Enabled System*, in: 2008.

[31] S. MURTAZA, A. G. HOEKSTRA, and P. M. A. SLOOT, *Compute Bound and I/O Bound Cellular Automata Simulations on FPGA Logic*, ACM Transactions on Reconfigurable Technology and Systems 1.4 (2009).

[32] S. MURTAZA, A. G. HOEKSTRA, and P. M. A. SLOOT, *Cellular Automata Simulations on a FPGA Cluster*, International Journal of High Performance Computing Applications, SAGE 25.2 (2010).

[33] A. C. LIMA and J. C. FERREIRA, *Automatic Generation of Cellular Automata on FPGA*, in: *9th Portuguese Meeting on Reconfigurable Systems (REC '13)*, 2013.

[34] G. Ch. SIRAKOULIS, *Cellular Automata Hardware Implementation*, in: *Cellular Automata: A Volume in the Encyclopedia of Complexity and Systems Science, Second Edition*, ed. by A. ADAMATZKY, Springer US, New York, NY, 2018.

[35] J. M. GREENBERG and S. P. HASTINGS, *Spatial Patterns for Discrete Models of Diffusion in Excitable Media*, SIAM Journal on Applied Mathematics 54 (1978).

[36] R. FISCH, J. GRAVNER, and D. GRIFFEATH, *Threshold-Range Scaling of Excitable Cellular Automata*, Statistics and Computing 1 (1991).

[37] R. DURRETT and D. GRIFFEATH, *Asymptotic Behavior of Excitable Cellular Automata*, Experimental Mathematics 2.3 (1993).

[38] J. M. GREENBERG, C. GREENE, and S. HASTINGS, *A Combinatorial Problem Arising in the Study of Reaction-Diffusion Equations*, SIAM J. Matrix Analysis Applications 1 (1980).

[39] A. K. DEWDNEY, *Computer Recreations: The Hodgepodge Machine Makes Waves*, Scientific American 259.2 (1988).

[40] M. GERHARDT and H. SCHUSTER, *A Cellular Automaton Describing the Formation of Spatially Ordered Structures in Chemical Systems*, Physica D: Nonlinear Phenomena 36.3 (1989).

[41] M. J. GIBSON, E. C. KEEDWELL, and D. A. SAVIĆ, *An Investigation of the Efficient Implementation of Cellular Automata on Multi-Core CPU and GPU Hardware*, Journal of Parallel and Distributed Computing 77 (2015).

[42] E. N. MILLÁN et al., *Performance Analysis and Comparison of Cellular Automata GPU Implementations*, Cluster Computing 20.3 (2017).

[43] G. VICHNIAC, *Simulating Physics with Cellular Automata*, Physica D: Nonlinear Phenomena 10 (1984).

[44] E. F. CODD, *Cellular Automata*, Academic Press, NYC, 1968.

[45] S. RAFLER, *Generalization of Conway's Game of Life to a Continuous Domain - SmoothLife*, arXiv:1111.1567 (2011).

[46] B. W. C. CHAN, *Lenia - Biology of Artificial Life*, arXiv:1812.05433 (2018).