TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Remote Execution of an FPGA-based Cellular Automata Accelerator on the Amazon F1 Cloud

---

Author:
AIKATERINI
TSIMPIRDONI

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Michalis ZERVAKIS
Prof. Sotirios IOANNIDIS

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

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

September 24, 2024

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Remote Execution of an FPGA-based Cellular Automata Accelerator on the Amazon F1 Cloud

by AIKATERINI TSIMPIRDONI

Cellular automata, introduced by John von Neumann and Stanislaw Ulam in the 1940s, are discrete mathematical models used to simulate complex systems through simple rules. They are widely applied in various scientific fields to study dynamic systems. In this thesis, a reprogrammable FPGA-based framework was developed to efficiently simulate cellular automata (CA) models on the AWS F1 platform. The design builds upon the architecture initially developed by Nikolaos Kyparissas and later extended by Emmanouil Milonakis, with additional improvements introduced in this work. The AWS FPGA Developer AMI was employed, offering a pre-configured environment with tools like Xilinx Vivado, accessed remotely via NICE DCV, eliminating the need for a physical FPGA board.The framework was created by generating an Amazon FPGA Image (AFI) bitstream, compatible with the AWS F1 instance. After implementing logic changes in the VHDL code, the bitstream was synthesized and deployed. Optimizations included upgrading the memory system from a 128-bit DDR2 to a 512-bit DDR4 configuration, enhancing data handling and increasing burst size. Data transfer between the host and FPGA was managed via PCIe using Direct Memory Access (DMA) by configuring the PCIe to AXI bridge for efficient communication.The FPGA executed the CA model, achieving a 21.2x performance improvement over traditional software methods, particularly when processing a 1920x1080 CA grid.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Remote Execution of an FPGA-based Cellular Automata Accelerator on the Amazon F1 Cloud

by AIKATERINI TSIMPIRDONI

Τα κυψελωτά αυτόματα, που αρχικά σχεδιάστηκαν από τον Τζον Βον Νόιμαν και τον Στανισλάβ Ούλαμ τη δεκαετία του 1940, είναι διακριτά μαθηματικά μοντέλα για την προσομοίωση σύνθετων συστημάτων μέσω απλών κανόνων.Αυτή η έννοια λειτουργεί ως ένα θεμελιώδες εργαλείο για την κατανόηση διαφόρων δυναμικών συστημάτων σε διάφορους επιστημονικούς και μαθηματικούς τομείς.Στην παρούσα εργασία, αναπτύχθηκε μια επαναπρογραμματιζόμενη αρχιτεκτονική βασισμένη σε FPGA για την αποδοτική προσομοίωση μοντέλων κυτταρικών αυτόματων (CA) στην πλατφόρμα AWS F1. Ο σχεδιασμός βασίζεται στην αρχιτεκτονική που αναπτύχθηκε αρχικά από τον Νικόλαο Κυπαρισσά και επεκτάθηκε αργότερα από τον Εμμανουήλ Μυλωνάκη, με επιπλέον βελτιώσεις. Χρησιμοποιήθηκε το AWS FPGA Developer AMI, το οποίο προσφέρει ένα προ-ρυθμισμένο περιβάλλον με εργαλεία όπως το Xilinx Vivado, με απομακρυσμένη πρόσβαση μέσω NICE DCV, εξαλείφοντας την ανάγκη για φυσική πλακέτα FPGA. Η αρχιτεκτονική δημιουργήθηκε μέσω της παραγωγής ενός bitstream Amazon FPGA Image (AFI), συμβατού με την AWS F1 instance. Μετά την εφαρμογή αλλαγών στη λογική του κώδικα VHDL, το bitstream αναπτύχθηκε. Οι βελτιστοποιήσεις που έγιναν περιλάμβαναν την αναβάθμιση του συστήματος μνήμης από 128-bit DDR2 σε 512-bit DDR4, βελτιώνοντας τη διαχείριση δεδομένων και αυξάνοντας το μέγεθος των μεταφερόμενων δεδομένων.Η μεταφορά δεδομένων μεταξύ του host και του FPGA πραγματοποιήθηκε μέσω PCIe χρησιμοποιώντας Direct Memory Access (DMA). Το FPGA εκτέλεσε το μοντέλο CA, επιτυγχάνοντας βελτίωση απόδοσης κατά 21,2 φορές σε σχέση με τις παραδοσιακές μεθόδους λογισμικού, ειδικά κατά την επεξεργασία πλέγματος CA ανάλυσης 1920x1080.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Apostolos Dollas, for his invaluable guidance and his support throughout this journey. His insights and encouragement have been a great source of strength for me, both academically and mentally.

I am also grateful to my committee members, Professor Sotirios Ioannidis and Professor Michael Zervakis.

Additionally, I must extend my thanks to my friends and family, whose enduring support and encouragement have sustained me throughout this process. Their belief in my abilities has been a constant motivation and has made this challenging journey more enjoyable.

Each one of you has contributed significantly to my growth and this achievement, and I am eternally thankful for your generosity and support.

Contents

List of Figures

List of Tables

List of Abbreviations

AFI	**A**mazon **F**PGA **I**mage
ALU	**A**rithmetic **L**ogic **U**nit
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuits
AWS	**A**mazon **W**eb **S**ervices
BRAM	**B**lock **R**andom **A**ccess **M**emory
CA	**C**ellular **A**utomaton-on/-a
CAD	**C**omputer **A**ided **D**esign
CPLD	**C**omplex **PLD**
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GUI	**G**raphical **U**ser **I**nterface
HIP	**H**ybrid **I**ntegration **P**rotocol
IP	**I**ntegrated **P**rotocol
LSB(s)	**L**east **S**ignificant **B**it(s)
MPI	**M**essage **P**assing **I**nterface
MSB(s)	**M**ost **S**ignificant **B**it(s)
PLD	**P**rogrammable **L**ogic **D**evice
RAM	**R**andom **A**ccess **M**emory
SPLD	**S**imple **PLD**
TCL	**T**ool **C**ommand **L**anguage
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter
VHDL	**VHSIC H**ardware **D**escription **L**anguage
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
XDMA	**Xilinx Direct Memory Access**

Heartily Dedicated to my family and friends…

Chapter 1

Introduction

A cellular automaton is a discrete mathematical models for simulating complex systems whose result depends on the simplistic rules. Cellular automata were invented in the 1940s by John von Neumann and Stanislaw Ulam. The original purpose of cellular automata was the simulation of biological self-replication mechanisms. This is a grid of cells that evolve at discrete time intervals. The status of each and every cell in this grid is determined by a predetermined rule, which depends on the cell's current status and that of each of its neighboring cells. It has spawned a universal platform in which most dynamic systems in science and mathematics can be studied and analyzed.

1.1 Motivation

In the rapidly advancing field of technological applications, the demand for efficient computational methods is more needed than ever. Traditional computational models often grapple with complexities and time constraints inherent in research, highlighting a crucial need for innovative solutions.This thesis explores the potential of FPGA technology and cellular automata in addressing these challenges. FPGA's flexibility and efficiency in handling parallel computations make it an ideal candidate for modeling complex system parameters. At the same time, cellular automata offer a unique approach to simulating complex behaviors through simple, rule-based interactions.By integrating FPGA-based cellular automata accelerators with the concept of remote execution, this research proposes a novel framework that promises to enhance accessibility, scalability, and collaboration in mathematical research and applications.My motivation for this research comes from a deep interest in my interest in the fields of mathematics and computer architecture. This

dual fascination has driven me to explore the intersection of these two domains, particularly focusing on how advancements in computer architecture can be applied to address complex challenges in mathematical modeling.

1.2 Thesis Contributions

In this thesis, we have made significant scientific contributions by developing an FPGA-based framework for cellular automata, optimized for enhanced performance and implemented on the Amazon F1 Cloud.The distinct contributions of this research are outlined below:

- **Design and Implementation of FPGA-Accelerated Cellular Automata Framework:** A scalable architecture was developed to execute cellular automata (CA) algorithms on FPGA hardware. By utilizing the high-performance computational capabilities of the Xilinx UltraScale+ FPGAs on AWS EC2 F1 instances, this framework achieves notable computational efficiency, addressing the complexity and scalability challenges inherent in large cellular automata grids (up to 21x21 neighborhood configurations). This innovation enhances execution speeds up to 21x faster than traditional CPU-based systems.
- **Optimization for High-Performance Remote Execution:** By optimizing the CA framework for the AWS F1 environment, the system benefits from AWS's cloud infrastructure, enabling the deployment of large-scale simulations remotely. The use of Amazon FPGA Image (AFI) technology facilitates dynamic reconfiguration without the need for physical hardware, making it accessible to a broader range of users.

1.3 Thesis Outline

- **Chapter 1 - Introduction:** Introduces the topic of FPGA-based cellular automata, outlining the thesis structure and the research motivations.
- **Chapter 2 - Theoretical Background:** Discusses various types of cellular automata, their mathematical models, and applications.
- **Chapter 3 - Related Work:** Reviews existing implementations and hardware architectures for cellular automata, focusing on FPGA technologies.

- **Chapter 4 - Hardware Architecture:** Details the specific FPGA architecture utilized in this research, including its configuration and technological aspects.
- **Chapter 5 - AWS Configuration for FPGA Implementation:** Explains the setup and configuration of the FPGA system on the Amazon F1 Cloud, including necessary AWS services.
- **Chapter 6 - Design of the Re-programmable Framework in AWS:** Details the design and development of a re-programmable FPGA framework on AWS, discussing the connectivity and the configurations.
- **Chapter 7 - System Verification, Examples of Use, and Results:** Covers the verification processes for the FPGA system, provides examples of its applications in cellular automata simulations, and presents the performance results.
- **Chapter 8 - Conclusions and Future Work:** Summarizes the findings, discusses the implications and contributions of the research, and outlines potential future research directions and applications.

Chapter 2

Theoretical Background

This section lays the underlying theory in order to understand the topics discussed in this thesis, including all relevant definitions and calculations, which are crucial for understanding the focus of both the hardware design and the CAD tool that has been developed.

2.1 Cellular Automata Types Of Grid

Cellular automata are mathematical models designed to simulate complex systems through simple rules, often on a grid of cells. They are categorized by the number of dimensions, such as 1D (one-dimensional), 2D (two-dimensional), and even 3D (three-dimensional). While one-dimensional and three-dimensional cellular automata exist, my thesis will exclusively concentrate on the two-dimensional type. A 2D -cellular automaton consist of a grid of cells NXN, each in a specific discrete state at any given time. This grid is updated repeatedly in discrete steps, with each cell's state changing based on a fixed mathematical rule called transition function that depend on the states of its neighboring cells.

In 2D cellular automata, there are two primary types of neighborhoods: von Neumann and Moore. Neighborhoods are characterized either by their n × n dimensions or by a radius r, which defines the distance from the center cell to the neighborhood's boundary.

2.1.1 Von Neumann Neighborhood

The Von Neumann neighborhood is a concept in cellular automata named after John von Neumann, who made substantial contributions to the field. It describes a specific pattern of cells around a central cell on a two-dimensional

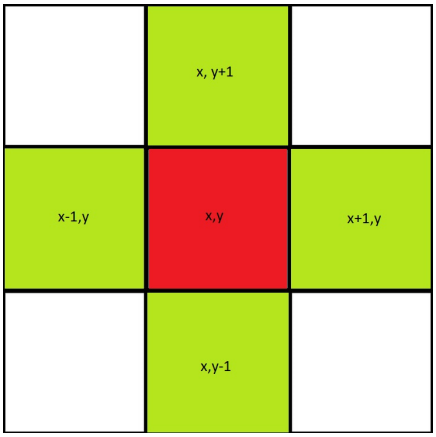


FIGURE 2.1: Von Neumann Neighborhood 3x3

square grid. This neighborhood includes only the cells directly adjacent to the center in the cardinal directions—North, South, East, and West—resulting in a cross-shaped pattern of four cells.

In contrast, the Moore neighborhood includes all eight cells surrounding the central cell, covering both direct and diagonal neighbors, making it a broader configuration than the von Neumann neighborhood. Mathematically, for a cell positioned at coordinates (x, y), the von Neumann neighborhood comprises the cells at $(x, y+1)$, $(x+1, y)$, $(x, y-1)$ and $(x-1, y)$ representing the North, East, South, and West directions, respectively .

2.1.2 Moore Neighborhood

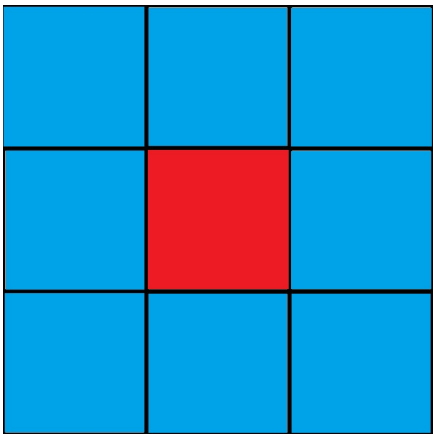


FIGURE 2.2: Moore Neighborhood 3x3

The Moore neighborhood is defined as a square region centered around a given cell within a grid. This neighborhood has an NXN area where N is an odd integer. The odd dimension is crucial as it centers the neighborhood

symmetrically around the central cell because it allows for symmetric arrangement. The square configuration of the Moore neighborhood means that it includes all cells that are at most $N - 1/2$ cells away from the central cell in any direction, including diagonally. In contrast, the von Neumann neighborhood would still retain its characteristic diamond shape, encompassing cells that are directly connected along horizontal or vertical paths, but not diagonally that is often more suitable for applications that model direct connectivity without diagonal shortcuts.

To conclude, both types of neighborhoods described above can be implemented with or without weights assigned to neighboring cells or can be extended to larger neighborhoods. Weighted neighborhoods assign different importance or influence to cells based on their distance from the center or other criteria, which can be crucial for simulations requiring differential interaction among cells.

2.1.3 Custom Neighborhood

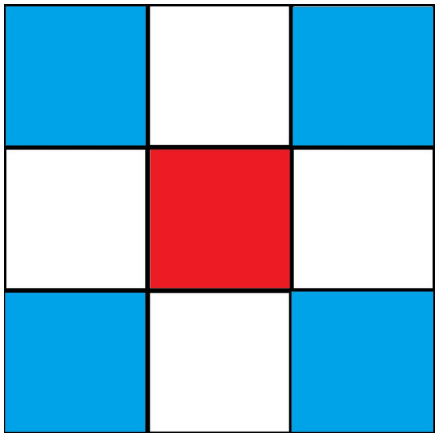


FIGURE 2.3: Custom Neighborhood 3x3

There are two mathematically defined types of neighborhoods, described above as Moore and von Neumann. Any other types of neighborhoods will be referred to as custom.Grids that are not mathematically constrained as the previously mentioned types.

2.2 Cellular Automata As Mathematical Models

In cellular automata (CA) research, two main categories are frequently mentioned: totalistic and outer totalistic cellular automata. These categories specify how the transition function for cellular automata are determined by the

states of the cells and their neighboring cells.

2.2.1 Totalistic Cellular Automata

In the context of cellular automata (CA), a totalistic cellular automaton (TCA) is a specific type where the next state of each cell is determined solely by the sum of the states of its neighboring cells[1]. To mathematically illustrate a Totalistic Cellular Automata (TCA) problem, consider a finite set of possible cell states represented by :S $=\{s0, s1, \ldots, sn\} \subseteq \mathbb{Z}0+$The total sum of each cell, calculated using predefined weights, is expressed by equation:

$$S = \sum_{x=-r}^{r} \sum_{y=-r}^{r} w(x,y) \cdot c_t(x,y)$$

where r is the neighborhood radius and w (x, y) the neighborhood weights with w (0, 0) = 0 and $a, b, \ldots \epsilon \mathbb{Z}$.
while the transition function is defined by equations :

$$c_{t+1}(i,j) = \begin{cases} s_0 & \text{if } S \leq a \\ s_1 & \text{if } a < S \leq b \\ \vdots & \\ s_n & \text{otherwise} \end{cases}$$

2.2.2 Outer Totalistic Cellular Automata

On the other hand,an outer totalistic cellular automaton is characterized by its unique rule for state transitions, where the state of each cell in the next generation is determined by the combination of its own current state and the aggregate sum of the states of its neighboring cells[1].To mathematically determine an Outer Totalistic Cellular Automata (OTCA) problem, consider a finite set of possible cell states represented by :S $=\{s0, s1, \ldots, sn\} \subseteq \mathbb{Z}0+$The total sum of each cell, calculated using predefined weights, is expressed by equation:

$$S = \sum_{x=-r}^{r} \sum_{y=-r}^{r} w(x,y) \cdot c_t(x,y)$$

While the transition function differs from that of the totalistic cellular automaton mentioned above, it is defined by the following equations:

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

In summary, Totalistic Cellular Automata (TCA) and Outer Totalistic Cellular Automata (OTCA) form a part of the models aimed at complex systems that work with simple rules based on neighboring cell states. Despite the fact that they are similar in terms of focusing on local interactions and state transitions, the methods of their individual applications are different. In TCA, the next state of a cell is determined by the total of the states of its neighboring cells exclusively. On the other hand, OTCA comprises the cell's present state as well into the transition function, providing a more intricate dynamic between the cell and its surroundings. These two mathematical models have some notable applications that are going to be analyzed above.

2.3 Applications of Outer Totalistic Cellular Automata

The Game of Life was invented by John Horton Conway, a British mathematician, in 1970. It probably represents one of the most well-known models of cellular automata.[2] It is an outer-totalistic model and is simple enough to understand the concept of cellular automata.It includes only two states ('0' or '1', 'dead' or 'alive') and the transition function is:

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

The sum S is the total of the neighbors, and $c_t(x_0, y_0)$ is the current state of the central cell.

Also same of notable examples that belong in the same category of outer totalistic rules are :The Greenberg-Hastings Model,The Hodgepodge Machine [3].

2.4 Applications of Totalistic Cellular Automata

In the context of TCA, we tested Anisotropic Rules, where we considered the anisotropy of the neighborhood. Here, the neighborhood has greater weights on the right side. The weights decrease from this maximum to 1 on the left side[3]. The transition function, which takes into consideration only the weighted neighborhood, is therefore considered as totalistic CA, is:

$$c_{t+1}(i,j) = \begin{cases} (c(i,j)-1) \mod 256 & \text{if WS} > \text{t} \\ (c(i,j)+1) \mod 256 & \text{if WS} < \text{t} \\ c(i,j) & \text{otherwise} \end{cases}$$

The sum WS is the total of weighted neighbors.

To conclude, these are the two ways to categorize cellular automata into classes, which relates to the contents of the transition function. It is important to refer to specific applications that have developed through time to understand the exact classification and use of these mathematical models.

2.5 CA Accelerators

For CA with few state per cell and small neighborhoods, general purpose CPUs are actually quite effective. This efficiency is credited to advanced algorithms, such as Hashlife. The Hashlife algorithm uses memorization to remember previous computations.In general,it is a good and fast practice for small neighborhoods [4].As the number of states per cell, the neighborhood, and weights increase, the rules of CA become more complex, leading to increased computational requirements.This complexity prevents the use of standard CPUs for running simulations, prompting the development and use of specialized hardware accelerators.Over time, a significant number of CA accelerators have been developed to enhance processing speeds beyond what CPUs can achieve. The development of this hardware technology may be FPGA-based or otherwise.

2.6 FPGA Technology

To gain an understanding of FPGA-based accelerators, it is recommended to refer to the fundamental aspects of FPGA technology.FPGAs are highly

programmable semiconductor devices that are widely used in digital circuits for various applications. This technology consists of several key components like:Programmable Interconnections,Configurable Logic Blocks(CLBs),Input and Output Resources,Digital Signal Processing (DSP) Units.

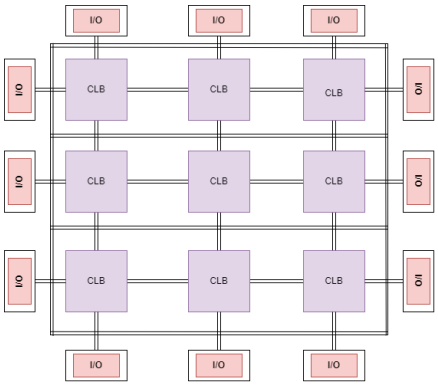


FIGURE 2.4: An FPGA Architecture Overview

1. Input and Output (I/O) Resources: These are resources that control the movement of data into and out of the FPGA. They provide interfaces between the FPGA internal logic and the external environment, so the FPGA is able to communicate with other devices or systems.

2. Configurable Logic Blocks: CLBs are the basic elements of an FPGA. CLBs can be programmed to conduct wide-ranging logical operations. Often, they consist of some logical gates and memory elements, such as flip-flops, which can be configured according to the needs of the application .
   A connection of I/O and CLB shown in Figure 2.4. .

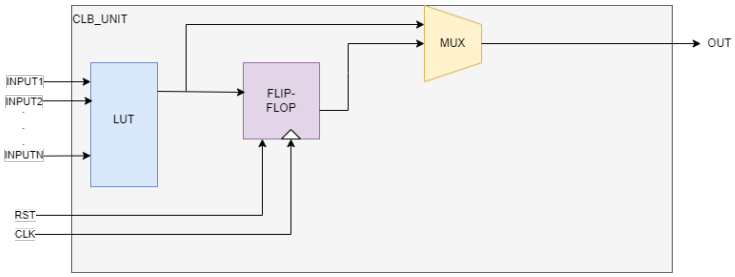


FIGURE 2.5: CLB Block Diagram

3. Programmable Interconnections: These provide flexible routing paths allowing different parts of the FPGA to connect with each other. This programmability gives the FPGA the flexibility to be configured for various tasks by altering how different blocks are interconnected.

4. Digital Signal Processing (DSP) Units: These specialized units are designed to efficiently handle mathematical functions commonly used in signal processing tasks.

The CLBs are linked through Programmable interconnections but the CLB actual contains LUTS,MUXs and Flip-flops as it shown in Figure 2.5.

2.6.1 Not-FPGA-Based Technology

Programmable Logic Devices (PLDs) also include ASICs, SPLDs, CPLDs, and microcontrollers in addition to FPGAs. SPLDs and CPLDs are not as powerful from a computation perspective as FPGAs and are generally used for encoding/decoding, data display, etc. They can also be used as an adjunct to an FPGA on a PCB to hold system configurations and let the FPGA handle the heavy-lifting processes. With respect to other PLDs, microcontrollers are quite different because they consist of both digital and analog components. They have a CPU-like pipeline, as well as RAM, ROM, and I/O ports for communication to external devices.These are the ASICs (Application Specific Integrated Circuits) and form the main class of devices against which FPGAs are competing in terms of speed and processing power. ASICs are application-specific devices and cannot be reprogrammed as FPGAs. In this class come CPUs and GPUs, known to achieve outstanding performance and to support specific user experiences. As they are hard-wired devices, developing new solutions for an ASIC requires custom software development, which in turn makes them less flexible than FPGAs. The reprogrammability of FPGAs offers a wide range of modifications and optimizations, making them more flexible for task specialization and often outperforming CPUs in niche applications.

In the above subsection discusses characteristics of technologies that are based on FPGA as well as those that are not. Up until now, we have concluded that when dealing with complex problems, the best way to make the same process faster is to create an FPGA accelerator. This was also the case for large neighborhoods and complex transition functions in Cellular Automata (CA). The next chapter will explore this further.

2.6.2 Cloud and AWS

In this thesis, the concept of Remote Execution is explored, utilizing AWS services and the cloud in the FPGA domain for implementation.AWS provides dedicated services supporting the use of FPGAs on the cloud. More specifically, AWS EC2 F1 instances provide a perfect platform for applications that can leverage FPGA accelerators: these are instances with embedded FPGA hardware that can be programmed with custom hardware accelerations to maximize performance.The AWS service give in developers the ability to develop, deploy, and test their FPGA designs, remotely, without the requirement for any kind of physical hardware on site. This is facilitated with AWS development tools the AWS FPGA Hardware Development Kit (HDK) and the AWS FPGA Software Development Kit (SDK).

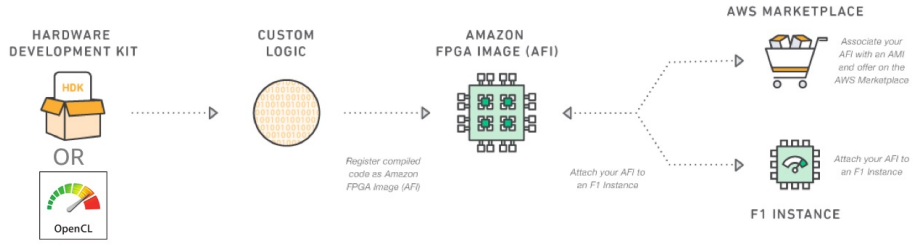


FIGURE 2.6: Development Flow AWS F1 instance (Source: https://github.com/aws/aws-fpga/tree/master)

The process of creating an Amazon FPGA Image (AFI) ,as shown in figure 2.6 ,for use on AWS F1 instances starts with developers developing the FPGA design in Hardware Description Languages (HDLs) like VHDL or Verilog .The developer during that phase have crafted the hardware design known as Custom Logic.the next phase is synthesis step that transforms HDL code to the gate-level form, which can be applied to FPGAs. At this stage, there are optimizations for performance and resource reductions.Then the next is implementation, this encompasses the processes of place and route, including timing analysis to verify the design can meet performance specifications. Finally, the design is compiled into a Design Checkpoint (DCP), which contains all necessary configuration files.Once the DCP is validated, it is developed into an Amazon FPGA Image (AFI).The Software deploy the AFI through launching an F1 instance and installing the AFI onto the FPGA, for that reason AWS provides SDKs to support AFI management for these instances.Furthermore, this AFI can be integrated into an AMI where the software environment lives. As a result, the AFIs can be shared and re-used more easily in that way.

Chapter 3

Related Work

3.1 Origins and Impact of Cellular Automata

Cellular automata originated in the 1940s, created by Stanislaw Ulam and John Von Neumann. Ulam was exploring crystal growth while Von Neumann was examining self-replicating systems . On Ulam's advice, Von Neumann concentrated on a discrete, two-dimensional approach. The Von Neumann cellular automaton, an early form of cellular automata, aimed to shed light on the logical requirements for self-replication machine. It played a key role in Von Neumann's universal automaton, featuring a sophisticated ruleset encompassing 29 states. These states were divided into five orthogonal subsets, with each state characterized by red, green, and blue color values. This automaton marked the first instance in history of a discrete parallel computational model being formally recognized as a universal computer. The research of Von Neumann and Ulam in cellular automata have significantly impacted further research in computer science, physics, and theoretical biology [5].

3.2 Exploring Implemented Cellular Automata Engines

Over the years, a variety of architectural designs have been employed to enhance the development of hardware for cellular automata simulations.Among the various architectures that have been developed, the most fundamental and widely adopted ones were selected for analysis.

3.2.1 Cellular Automaton Machine (CAM)

Tommaso Toffoli's Cellular Automaton Machine (CAM), developed in 1984, is a specialized device designed for simulating cellular automata.Outperforming general-purpose computers in these tasks, CAM boasts a speed increase of a thousand times[6]. At its core, CAM integrates a high-performance parallel processor, specifically tailored for cellular automata simulation. It features a user-friendly interface for control purposes and a monitor port to visualize the evolution of the automaton.

The architecture of CAM is built around a 256 x 256 grid, where each cell can handle 8-bit data, allowing for 256 unique cell states. The machine operates on a parallel processing framework, simultaneously updating cells in accordance with a predefined transition function.

In a novel approach, Toffoli uses a single hardware component, an SRAM LUT, to implement the transition function across all cells. This design choice means that CAM processes the cells and their surrounding environments in a sequential manner, updating the entire grid to create a new frame in the simulation. To effectively manage this process, CAM employs the technique of double buffering. This method is crucial as it retains the old state of cells, which is necessary for accurate computation, rather than using the newly updated state that emerges post-update. This careful design consideration ensures the integrity and accuracy of the simulations conducted on CAM[6].

3.2.2 CAM-6 machine

Norman Margolus, then a PhD student, collaborated with Toffoli to enhance the prototype of the Cellular Automaton Machine (CAM).They developed several reformations, culminating in the completion of CAM-6 in 1986.

The architecture of CAM-6 was fully pipelined, with its memory divided into four bit-planes. Each bit-plane contributed one bit to every 4-bit cell, and the size of each plane was 256 by 256 sites.Enabled the transition function to access all 3x3 neighborhood cells simultaneously. Like its predecessor, CAM-6 was capable of displaying the evolution of a 256x256 cell grid in real time.

One of the most intriguing aspects of CAM-6 was its flexibility in reconfiguring the interconnection of the planes. Users had the option to either simulate multidimensional cellular automata by stacking the planes or simulate a larger grid by joining the planes edge-to-edge. This flexibility did come with

a trade-off, reducing the number of states per cell to 16. Additionally, larger grids could be simulated using a technique called scooping, where the larger grid is stored in the host computer's memory, and CAM-6's internal 256 by 256 grid acts as a cache memory.[7].

3.2.3 CAM-8 machine

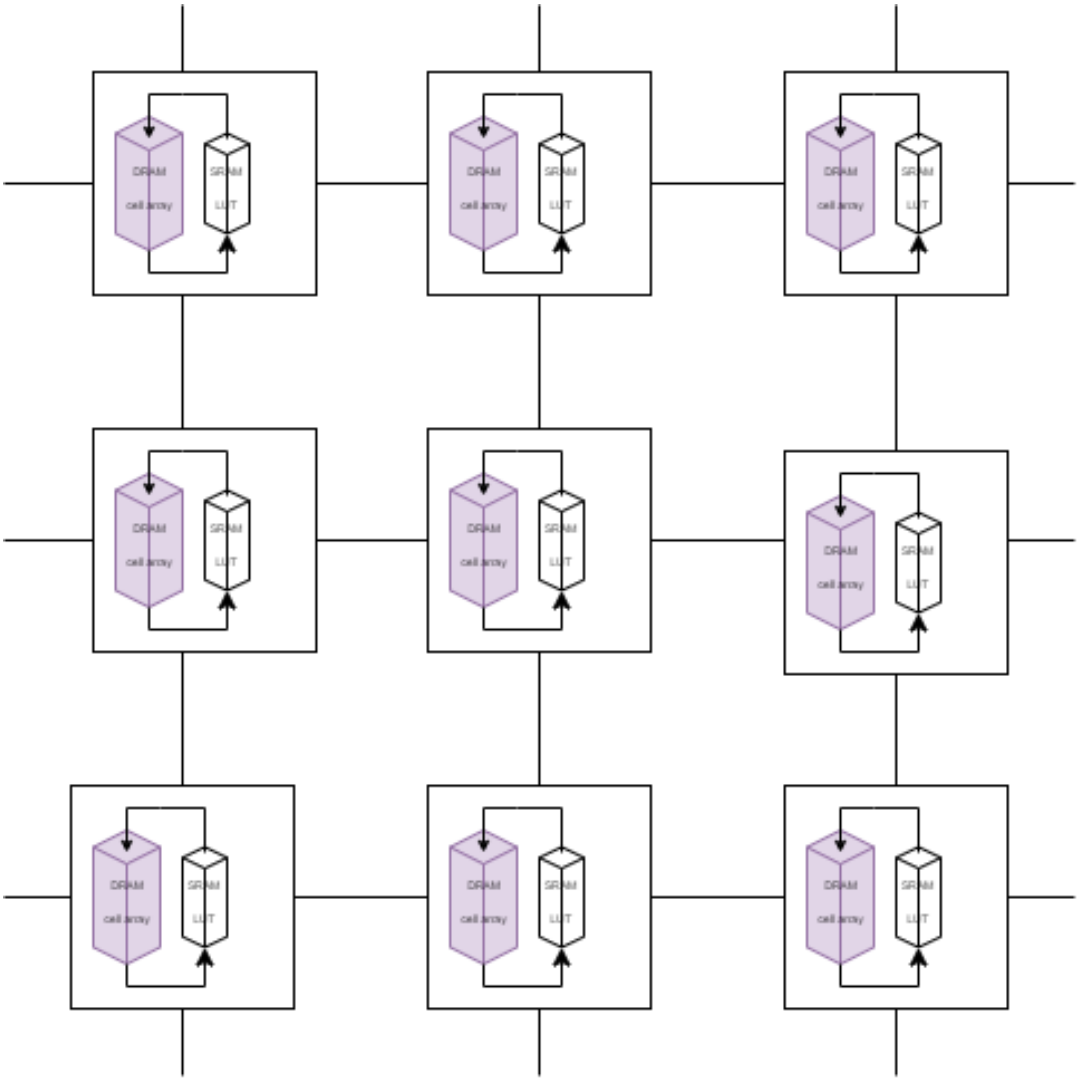


FIGURE 3.1: Cam-8 system diagram.Spatial array of cam-8 nodes, with nearest-neighbor.

The CAM-8, developed also by Norman Margolus, is a sophisticated device for simulating cellular automata systems, which are models used in computing to simulate complex systems. It's an improvement over previous models and works well for large-scale, cost-effective simulations[8].

The CAM-8 can be expanded to handle larger simulations(512x512 cells grid).The CAM-8 combines a unique architecture with efficient data handling

and connectivity to create an effective system for simulating physical spaces. It features a network of processors that coordinate by taking turns working on different parts of the simulation, ensuring comprehensive coverage and efficient handling of spatial information. The system is divided into several modules, each responsible for processing small, specific areas in sequence, which enables smooth and continuous operation across the entire system. For memory and data handling, CAM-8 utilizes standard DRAM chips and employs a specialized method for accessing and using this memory, involving the scanning of different data sections and updating them based on a preset guide, known as a SRAM lookup table. In terms of connectivity, CAM-8's design is straightforward yet efficient, with its units connected in a way that allows for the effective movement of data through the three-dimensional space of the simulation. This cohesive integration of architecture, processing modules, memory management, and connectivity makes the CAM-8 a powerful tool for large-scale, detailed simulations.The CAM-8 architecture is not an FPGA based application.[8].

3.2.4 CEPRA: Cellular Processing Architecture

The Cellular Processing Architecture (CEPRA) was a project between 1994 and 2000 at the Technical University of Darmstadt. It is an FPGA-based architecture featuring a streaming design similar to Cellular Automata Machines (CAM). Diverging from CAM, CEPRA employs pipelined arithmetic logic rather than Look-Up Tables (LUTs) for executing cellular automata (CA) transition functions. This innovative design allows CEPRA to efficiently compute complex rules in a single step, providing a distinct advantage over CAM[9].

The first member of the CEPRA family, CEPRA-8L, it showcased eight FPGA-based CA processors, enabling simultaneous access to their 3x3 neighborhood cells. CEPRA-8L achieved a notable display speed of 22 generations per second for a grid of 512x512 8-bit cells[10]. The subsequent iteration, CEPRA-1X, as an FPGA co-processor integrated into a PC expansion board. Using the host computer's memory for CA grid storage, CEPRA-1X supported real-time evolution of 1024x1024 16-bit cells. It accommodated both two-dimensional and three-dimensional CAs with neighborhoods of radius r = 1[11].

In the years following CEPRA-1X, the design team introduced a high-level Cellular Description Language (CDL)[12]. This language translates intricate

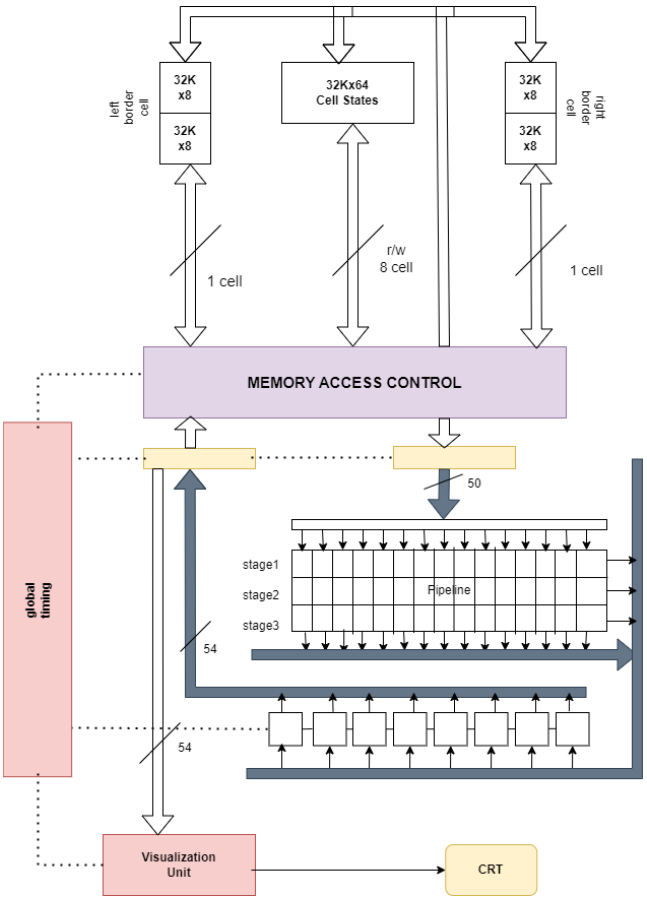


FIGURE 3.2: Block diagram of the CEPRA - 8.

CA rules into Verilog HDL, serving as a CA description language. CDL eliminates the necessity for specialized knowledge of the system architecture, offering a user-friendly tool for describing complex cellular automata.

3.2.5 SPACE :Scalable Parallel Architecture for Concurrency Experiments

SPACE explores the lattice gases as models for fluid dynamics, with a particular focus on their implementation as cellular automata fields. Traditional methods on stored program computers or lookup-table-based systems, such as the cellular automata machine (CAM), encounter computational hurdles that necessitate parallel processing for practical outcomes.This innovative architecture, based on Algotronix CAL FPGAs, exhibits speed advantages compared to memory-based simulations.

SPACE is designed with scalability as a foundational principle, employing FPGA boards with bidirectional communication between cells. The FPGAs boast a regular internal structure, enabling efficient interchip communication

and flexibility in configuring individual cell functions. A singular SPACE board comprises a 4x4 array of CAL chips, connected to create larger arrays with diverse configurations. Each board is equipped with clock circuitry and a conventional microprocessor for CAL configuration and state read-back. Input/output options contains links for serial interfaces and dual-ported memory boards for direct communication with a host computer. The architecture is suitable for various applications, including highly parallel simulation, hardware prototyping, and accelerating conventional host machines in processing fast/parallel data streams[13].

3.2.6 Large-scale Cellular Automata on FPGAs invented by Nikolaos Kyparissas

In the undergraduate thesis of student Nikolaou Kyparissas in 2020, an innovative FPGA-based architecture for cellular automata was developed. The approach that implemented in this study, involves designing a streaming architecture to process the cellular automata as a continuous flow of cells. This method proves to be more scalable, particularly when dealing with complex rules featuring extensive neighborhoods on sizable grids.This architecture facilitates the efficient computation of complex cellular automata with extensive 29×29 neighborhoods in Cartesian or toroidal grids, supporting either 16 or 256 states per cell. The implementation of the architecture describes a hardware framework for Cellular Automata (CA) simulations implemented in VHDL on an FPGA. The system comprises four subsystems: Memory Initialization and Frame Extraction, Memory Controller, CA Engine and Full-HD Graphics. The framework allows users to choose various parameters influencing the CA simulation, such as neighborhood size, cell size, memory burst size, and grid dimensions. After loading initial values into external DDR memory, the system displays the CA grid on a screen and the CA Engine processes cells in a sliding window, advancing one cell per clock cycle. The project serves as a flexible framework for users to develop their own CA hardware simulations.The CA operates in discrete time steps, utilizing double buffering for efficient state transitions.

The updated design not only achieves a substantial increase in speed compared to a high-end general-purpose CPU and similar speed enhancements as contemporary GPUs for reported 11×11 neighborhoods, but it does so with only a fraction of the energy consumption. Additionally, for larger

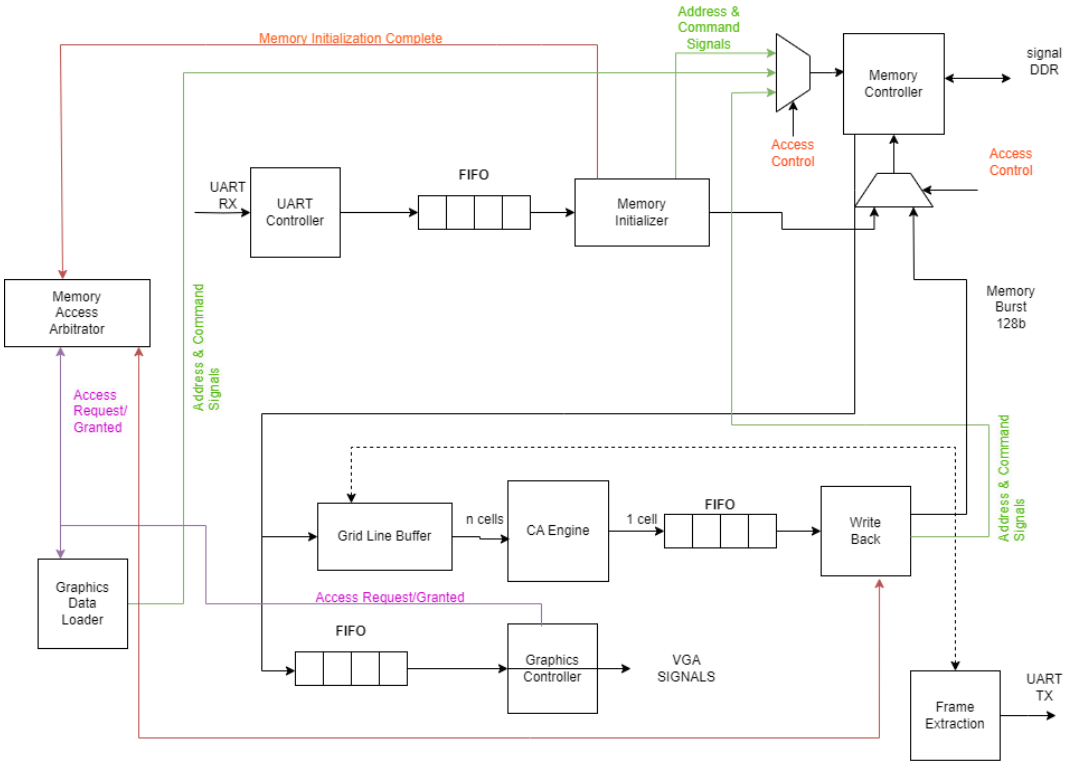


FIGURE 3.3: Block diagram of the System.

neighborhoods, it is anticipated to outperform GPUs both in terms of speed and energy efficiency[14].

3.2.7 The extension of Kyparissas's Architecture Created by Emmanouil Milonakis

As an extension of Kyparissas's project, Emmanouil Milonakis maintained a nearly identical architectural model, incorporating LUT values for the transition function. The proposed enhancements involve the development of a graphical user-friendly interface, enabling users to input desired dimensions and rules. Furthermore, a CAD tool is to be implemented, not just in terms of a user interface, but with the capability to receive user information—such as grid type, state bit count, and CA rules—and autonomously generate Xilinx Vivado files. This CAD tool will efficiently run necessary processes to produce a design, providing users with the finalized design without requiring any understanding of hardware-related intricacies.

3.2.8 Other Significant Work

In 2001, researchers Kobori, Maruyama, and Hoshino from the University of Tsukuba introduced a FPGA-based Cellular Automaton (CA) system. The system utilized a streaming architecture with an array of Processing Elements (PEs) traversing the CA grid. Each cell of the grid was processed consecutively n times, where n is the depth of the PE array, leading to a delay in generation output.The FPGA-based CA system achieved a significant speedup, simulating a 2,048 × 1,024 FHP lattice gas automaton at 400 generations per second, offering a 155× improvement over a high-end CPU at the time[15].

Between 2007 and 2010, researchers Murtaza, Hoekstra, and Sloot at the University of Amsterdam conducted a series of studies focusing on the performance modeling of FPGA-based Cellular Automaton (CA) systems. Drawing inspiration from the architectures introduced by Kobori, Maruyama, and Hoshino, they explored various topologies, sizes, and types of Processing Elements (PEs) tailored to whether a specific CA simulation was compute-bound or memory-bound. Their investigations extended to floating-point execution of lattice Boltzmann fluids on FPGA clusters[16].

In 2013, Lima and Ferreira from the University of Porto presented their reconfigurable CA architecture, which followed a similar approach to SPACE. Their processing unit comprised a PE array implementing the entire CA within the FPGA. The system's configuration was facilitated through a Graphical User Interface (GUI) on the host computer, allowing simulation of any CA with small neighborhoods. The automaton grid's size was adjustable, reaching up to 72 × 72 cells depending on the rule's complexity[17].

Chapter 4

The Hardware Architecture

Emmanouil Mylonakis has designed an architecture for cellular automata based on the architecture of Nikolaos Kyparissas [18]. This architecture includes a hardware design and a CAD software tool. The purpose of this work was to take the existing architecture of Nikolaos Kyparissas, which is presented in the following papers[3][19][20], and with some optimization and a software tool, to create an interface that would be more accessible to the user to utilize the specific CA Accelerator. Essentially, he created a user interface that helps users utilize this tool without having hardware design knowledge.

4.1 Nickolas Kyparissas's Design

In this section, we are going to look into details of the original architectural design by Nikolaos Kyparissas. This design is generally highly sophisticated and consists of neighborhoods with dimensions of up to 29 x 29. Currently, it stands as the most efficient design available.The design will be analyzed in detail to ensure a thorough understanding.

4.1.1 Top Level of the System

A simplified schematic of the design is shown in Figure 4.1.The design can show the model's progression on a 1920x1080 grid at 60 FPS in real time. In theory, a cellular automaton operates in an infinite space, but due to the limited memory of computational devices, the size of the grid must also be constrained.

In the design, Clocking Wizards are the first to be considered, which are not shown in the diagram, are used to generate clocks from an external reference clock (CLK100, operating at 100MHz) provided by an oscillator on the FPGA

board. The Memory Controller requires two different clocks: system clk and reference clk .These are used to generate an interface clock and to control the timing of the I/Os, respectively. Additionally, the FIFO modules, employing a First In, First Out methodology, manage data inputs and outputs, mainly to synchronize two distinct clock domains.

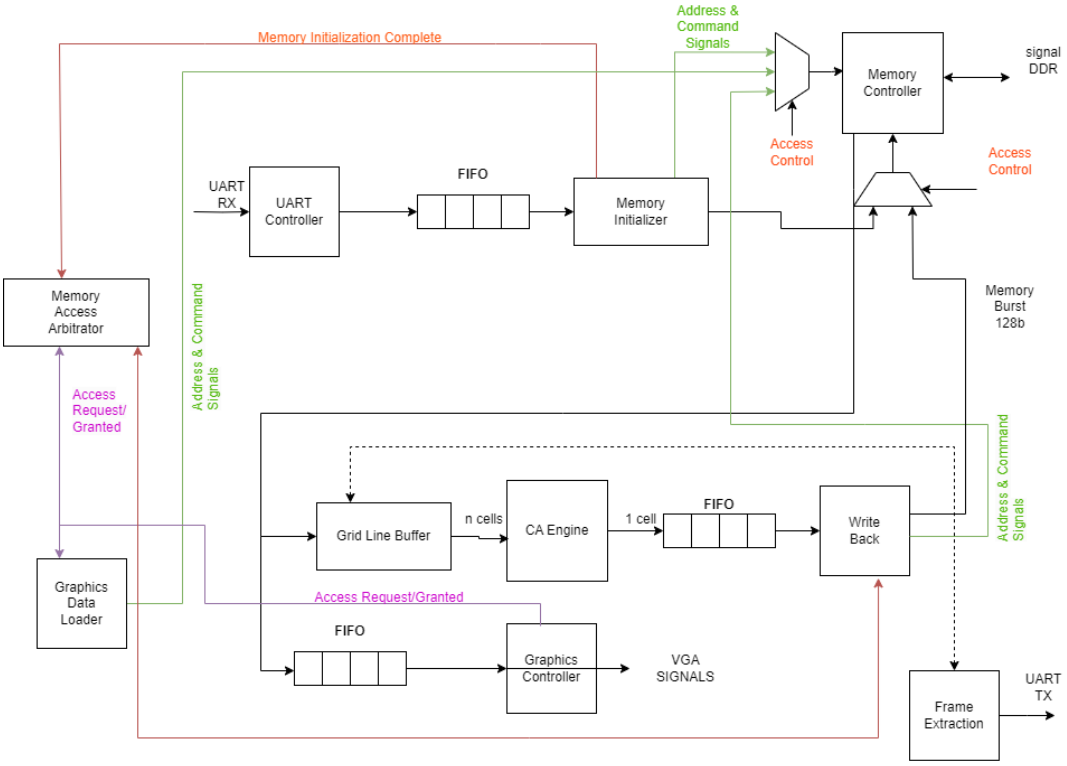


FIGURE 4.1: Block diagram of the System.

4.1.2 Memory Initialization

The process starts with creating a bitmap image using standard image editing software. This image visually represents they initial state of the grid, with each pixel matching to a cell on the grid .The color or shade of each pixel indicates the cell's state, providing a graphical interface for users to adjust and verify the grid's configuration before it is input into the FPGA system. Once the bitmap image is finalized , a Matlab script converts images into a text format suitable for a Fpga system. This script transforms each pixel value in the space-separate values in a text file, where each line corresponds to a grid row.

Next, the initial configuration is uploaded onto the FPGA system using a custom executable. This executable manages the UART connection and transfers the text file to the FPGA board via USB at a speed of 2 MBd. The transfer time

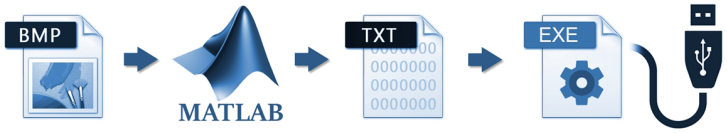


FIGURE 4.2: Process for preparation the initial state(taken from nick kyparissa's github)

varies based on the data complexity.During this initialization, a FSM oversees the data transfer, ensuring efficient data reading from the UART, data packing into 128 bits memory bursts, and writing to the system's memory. This FSM helps manage the data flow smoothly, addressing potential bottlenecks caused by the UART's slower speed compared to the system's logic. Once initialization is complete, a signal notifies that the grid has been successfully loaded with the initial configuration, and the system is prepared to start the calculations. This careful setup is crucial for accurately simulating cellular automata, allowing the system to perform complex computational models based on these initial settings.

4.1.3 Memory Controller and Double Buffering

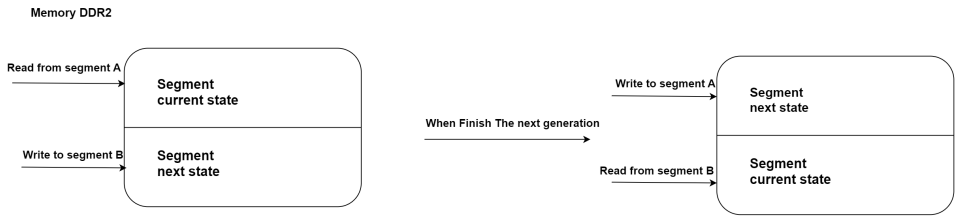


FIGURE 4.3: Double buffering concept

The memory controller, which is a critical component of this system, is designed to operate at high frequencies, ensuring fast data access and transaction speeds.The memory controller, engineered using Xilinx's Memory Interface, in our case it works with DDR2 and it is organized to handle data in 128-bits $(b = 128)$ bursts at a frequency of 325 MHz. Facilitating efficient data transfers and minimizing processing delays.The Technique that use to store data in the memory called double buffering.Double buffering is implemented within the system's memory architecture, which includes 128 MB of DDR2 memory.This method employs two separate memory segments,that are large enough to hold a frame of $XxYxCellsize$ bits each, to store frames of the cellular automaton's state, enabling one frame to be processed while

another is displayed. This dual-buffer setup is crucial for achieving real-time simulation of the automaton, which requires the system to update and display frames 60 times per second.

Grid representation in memory is described through an approach where the cellular automaton grid, consisting of 1920x1080 cells, is efficiently mapped into memory. This mapping is achieved through bursts that encapsulate multiple cells, determined by the cell's size in bits and the burst size.Each line has 60 or 120 rows, depending on the cell size. If the cell size is 4 bits, the bursts per line are 60, if the cell size is 8 bits, the bursts per line are 120 (4.1), and thus the memory is organized for the different cell sizes, while the lines in total remain 1080.

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

number of cells per memory burst, $c_b = \frac{b}{c}, \quad c_b \in \mathbb{N}$.(4.1)

4.1.4 Graphics and System's Data Loader

After transmission of the data and the writing in the DDR2 memory are complete, the system starts displaying the contents of the memory on screen.The system employs double buffering to toggle between displaying a simulation frame and processing the next one. Graphics data loading is managed by the Graphics Data Loader that access to the memory whenever it needs to load part of the frame.This loader serves as an intermediary between the graphics controller and the memory controller, facilitating communication between the two.Another component of the graphics subsystem, in addition to the Graphics Data Loader, is the graphics controller. This controller generates video synchronization pulses to manage the display's resolution, maintaining the visual integrity of the simulation on a Full-HD (1920×1080) display.Moreover, The system can read either 4-bits or 8-bits and convert it into a 12-bit color output signal in accordance with the selected color palette. This conversion allows each synchronization pulse from the Graphic controller to be associated with a specific RGB value.All These components are designed to display the changes in cellular automata from one generation to the next on the screen , achieving a rate of 60 generations per second. The graphics system needs to quickly access external memory to load parts of the frame. Since both the graphics system and the processing engine use this memory, it's important to keep access times short and give priority to graphics access

FIGURE 4.4: The color palettes supported by our system.

due to the real-time nature of screen displays.The role to load the data from memory to both graphics system and the processing engine system has the module Graphics Data Loader serves as it is the only module that requests data loading from memory.

4.1.5 Grid Lines Buffer

When the request data begins loading from memory, the Grid Lines Buffer becomes operational. The system employs two different types of grid line buffers: one for Rectangular and Cylindrical grids, and another for Toroidal grids.This choice does not require modification of the engine's base design.Need only to change the GRID TYPE in the top module in the Hardware design.

```
ENTITY TOP_LEVEL IS
    -- ALL VHDL MODULES INHERIT THEIR GENERIC VALUES FROM TOP LEVEL'S GENERIC VARIABLES
    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 BLACK-RED-WHITE GRADIENT PALETTE
        SPEED : INTEGER := 0;
        -- 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 BITS
    );
```

FIGURE 4.5: Top-Level Grid Type Specifications

4.1.6 Rectangular and Cylindrical grids

This Module temporarily stores parts of the cellular automata grid using Block RAM (BRAM) modules. These BRAMs store the states of cells within the nxn neighborhood necessary for processing each cell in a grid line. Additionally, there is another BRAM module used as a write buffer, which temporarily holds the updated cell states after processing by the Cellular Automaton Engine, before they are written back to the main grid storage or used in subsequent steps. The memory required for this setup is calculated

as $(n+1)xXxc$ (the (X x c) can be analyze also as bl × b bits). Here, n+1 accounts for the n BRAM modules storing the neighborhood data and one additional module for the write buffer. X represents the number of cells per grid line, and c is the bit-width of each cell state. This product gives the total bits stored across all BRAM modules.During the operation ,each Block RAM (BRAM) module is responsible for loading data that corresponds to a specific section of the grid.As the simulation progresses, the Cellular Automaton Engine retrieves the necessary neighborhood data from these BRAM modules to compute the new state for each cell, according to the predetermined rules of the automaton. After the new state of a cell is determined, it is stored in a write buffer BRAM.

The Grid Lines Buffer operates through two connected finite state machines (FSMs): a reader and a writer. The writer functions at a frequency of 81.25 MHz, aligning with the rate at which bursts of memory are received. It populates the buffer each time a grid line is requested by the Graphics Feeder. Simultaneously, the reader processes data from n BRAM modules at a speed of 200 MHz to supply the Cellular Automaton Engine.When a complete row in the Brams is filled up, the control system moves all the rows down by one position. The newest row that was just filled becomes ready to be emptied, while the oldest row now becomes the new place for writing data.There is a special mechanism that helps coordinate the actions of the reading and writing processes. This ensures that the system doesn't start filling a row before it has been emptied(figure4.6).

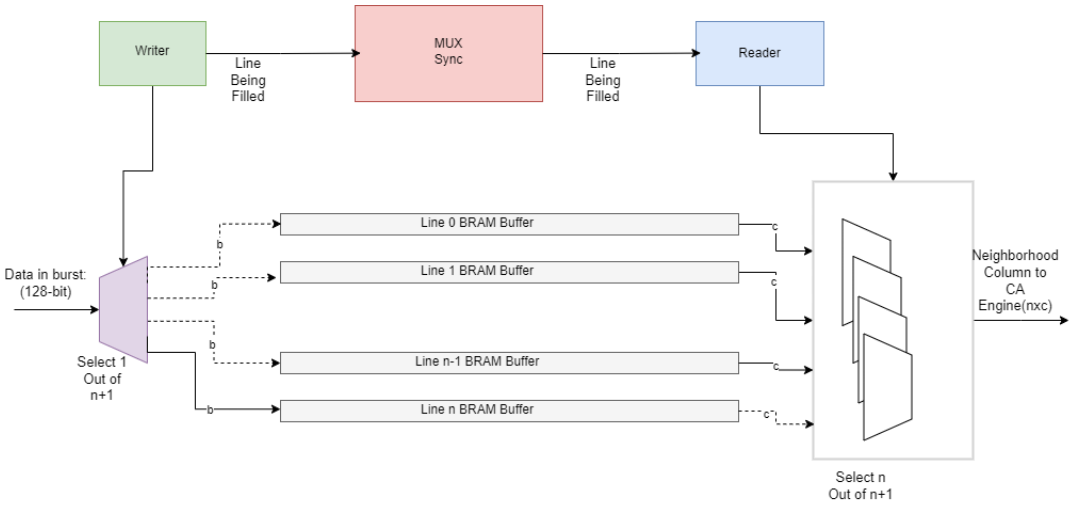


FIGURE 4.6: The Grid Line Buffer's internal structure.

In the Read FSM, there exists a state called PRELOAD NEIGHBORHOOD.

This mechanism functions for a rectangular grid in such a way that the edge cells often suffer from having incomplete neighborhoods. To address this issue, the engine can preload its neighborhood window with zero values, effectively padding the grid and ensuring that each cell can be processed as if it had a complete neighborhood. This zero-padding is pivotal for simulating cellular automata on a rectangular grid without data integrity issues at the boundaries.

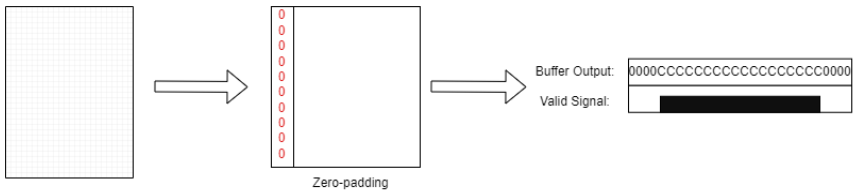


FIGURE 4.7: Zero-padding of Rectangular Grid.

The cylindrical grid configuration, on the other hand, enhances this concept by wrapping the vertical edges of the rectangular grid. This wrapping ensures that cells on the left edge of the grid incorporate cells from the right edge into their neighborhoods, and the other way around. This is accomplished by pre-loading the Cellular Automaton Engine's neighborhood window with appropriate cells from opposite ends of the buffer before the actual cells are processed. This setup allows each cell at the edge to behave as though it's centrally located within a continuous line of cells, thus avoiding edge effects.

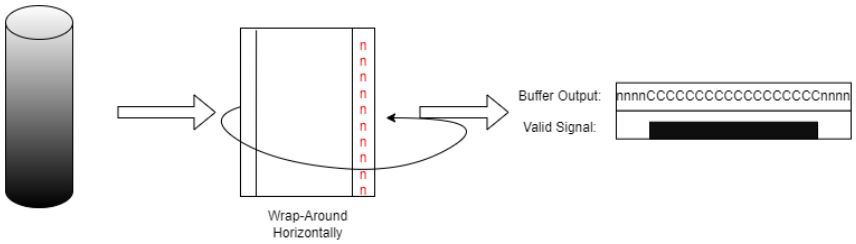


FIGURE 4.8: Wrap-Around Horizontally of Cylindrical Grid

4.1.7 Toroidal grids

A toroidal grid can preserve the wrapping around of both horizontally and vertically because its structure is like that of a torus.The implementation process begins by creating what's known as a toroidal wrap-around. This is done by first wrapping the vertical edges to form a cylindrical shape (similar to a previous cylindrical grid setup), and then connecting the horizontal edges to

complete the torus formation, referred to as a poloidal wrap-around.For the grid lines buffer , the system treats the toroidal grid similarly to the cylindrical grid during horizontal movements of the neighborhood's sliding window across the grid. This means the system preloads certain cell data before processing begins. However, for vertical movements, additional buffers and logic are necessary due to the wrap-around nature of the grid, where previously accessed grid lines are revisited and not reloaded by the typical loading mechanism.To manage these challenges, the system uses extra Block RAM (BRAM) ,The amount of BRAM resources required to implement the toroidal Grid Lines Buffer equals to $[(n+1)+(n-1)/2+(n-1)/2]xcbits = 2nxcbits = 2nb_lb$ bits.This implementation simulates environments where cells interact across the traditional boundaries of a grid.

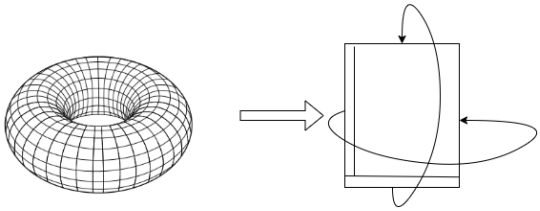


FIGURE 4.9: Wrap-Around Horizontally and Vertically of Toroidal Grid

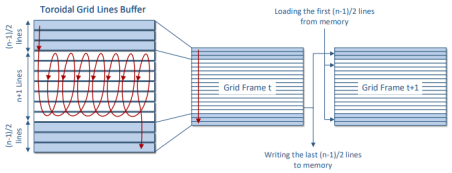


FIGURE 4.10: Toroidal Grid Lines Buffer (Source:[14])

4.1.8 CA Engine

After the Grid Lines buffer meets the CA engine, the module responsible for calculation of the next generation of each cell,the new values in the grid of a cellular automaton.It operates at 200 MHz and it produces a new cell value per clock cycle.The first step in these calculations involves loading neighborhood data into a pipelined neighborhood window, which uses an array of shift registers, it takes a neighborhood column that contains nxc bits and shifts the entire window with each clock cycle. After loading, each cell in the neighborhood is multiplied by a corresponding weight and uses an adder tree that handles the sum of all weighted cells,that gives the final sum of the neighborhood.The final step is the transition function's implementation as a simple lookup table that requires no more than one clock cycle to compute the new state of a cell, achieving the goal of updating one cell per clock cycle.

This architecture highlights several notable features. First, the use of a pipelined array of shift registers to manage the neighborhood window of the cells

significantly reduces the time complexity of loading neighborhood data from O(n^2) to O(1).Second, the absence of an input signal for data loading and the use of a 1-bit valid signal to determine the validity of the central cell in the column reduce unnecessary computations. Finally, the need to redesign the engine for each specific rule based on the automaton's needs underscores its flexibility but also highlights complexity issues in deployment. This approach allows for customization but at the cost of increased development time.The above steps are crucial, as it defines the rules of the cellular automaton, dictating how cells evolve from one generation to the next.To conclude,This architecture achieves high performance , particularly in processing 1 cell per clock cycle.

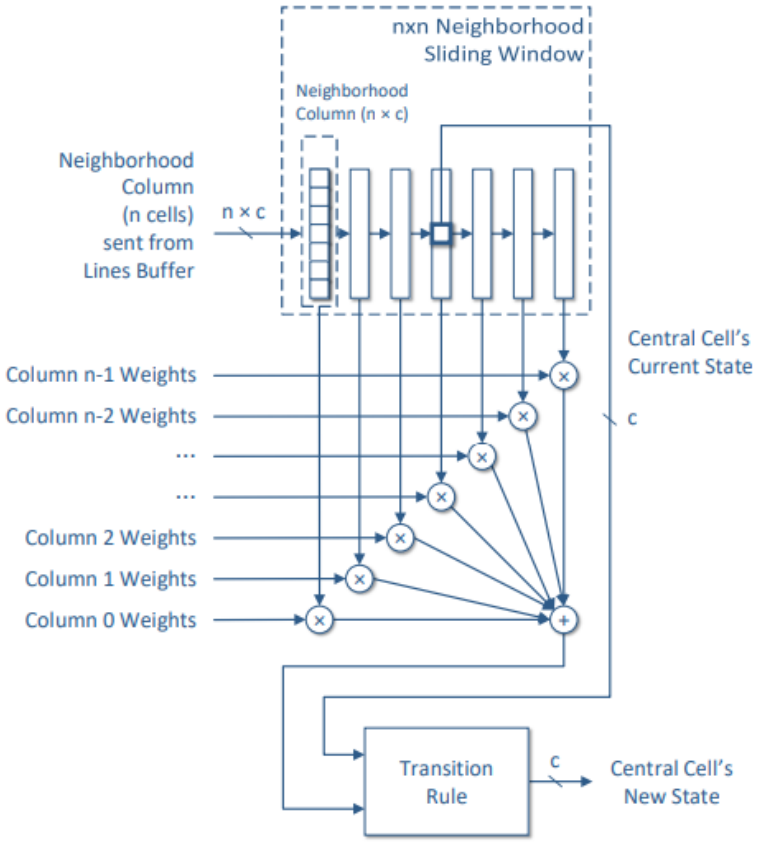


FIGURE 4.11: CA engine (Source:[14]).

4.1.9 Write-Back

As the Cellular Automaton Engine processes each grid line, a valid cell is generated every clock cycle and temporarily stored in a FIFO buffer. The

FIFO groups the cells into bursts(128-bits) that are queued for the memory write. The Write-Back module primarily acts as an arbitration point that centrally manages the control signals shared between the FIFO buffer and the Memory Controller. One such example is managing the address bus responsible for the memory to ensure that data gets correctly stored on the memory. It makes sure that every burst is written properly to memory before it sends a signal to the FIFO buffer to transmit another burst. Furthermore, the Write-Back module counts the number of bursts and lines written to the memory. This kind of operation is very useful for the module, at this point, it can properly control the operation of the writing process to the memory.This module maintains data integrity and synchronization across the system components.

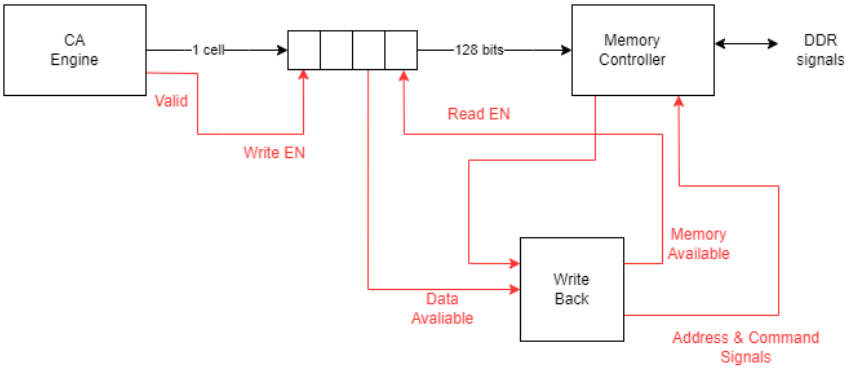


FIGURE 4.12: Write-back.

4.1.10 Memory Access Arbitrator

To manage the memory access between the read part and the write part need a module that parcel the memory between the Graphics Data Loader and the Cellular Automaton Engine's Write-Back.The main role of the Memory Access Arbitrator is to manage and prioritize access to the shared memory resource. Since the Graphics Controller and the Cellular Automaton Engine must access memory at the same time,one in order to read data and display part of the screen and the other to write back the new data,the role of the Arbitrator is to make sure that such accesses take place without conflicts.The Arbitrator works by receiving memory access requests from the Graphics Controller and the Cellular Automaton Engine. Each request includes not only the desire to access memory but also specifics about the memory blocks needed and the type of access required, whether it's read or write. The Arbitrator has a priority scheme to decide which request should be granted in case of a conflict, typically prioritizing read requests from the Graphics Controller to support continuous, real-time screen updates.After making a

decision, the Arbitrator then uses the control signals to the Memory Controller to bring about the access for the request that has been granted. It also communicates to the requesting modules, letting them know that the grant status has been accepted,so they can go on with their operations.In addition, the Memory Access Arbitrator takes care of the requests' queuing. If it is not possible to fulfill a request at a certain moment due to a running access, the request will be buffered in the Arbitrator. This buffer is also utilized dynamically in changing priorities to suit the requirements and specific conditions under which the system has to operate.All in all, the Memory Access Arbitrator is crucial for maintaining the system's performance and reliability, ensuring that memory is accessed in a managed way that supports the real-time demands of the cellular automaton system.

4.1.11 Top level

This top module incorporates various components necessary to construct the final CA accelerator, enabling the execution of cellular automata in real time at a rate of 60 frames per second. The module includes generics that allow users to adjust and parameterize the design based on specific requirements of the cellular automaton system. By using generics, the module offers flexibility in customizing both the structural and behavioral characteristics of the system to suit particular simulation needs. Below is an example of how these parameters might appear in the top module as it shown in figure 4.13. When a generic changes, the subsystem VHDL files will automatically inherit

```
ENTITY TOP_LEVEL IS
    -- ALL VHDL MODULES INHERIT THEIR GENERIC VALUES FROM TOP LEVEL'S GENERIC VARIABLES
    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 BLACK-RED-WHITE GRADIENT PALETTE
        SPEED : INTEGER := 0;
        -- 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 BITS
    );
```

FIGURE 4.13: Top Level generics.

these values, adjusting their structure and behavior without any further action needed from the user and make the final system.

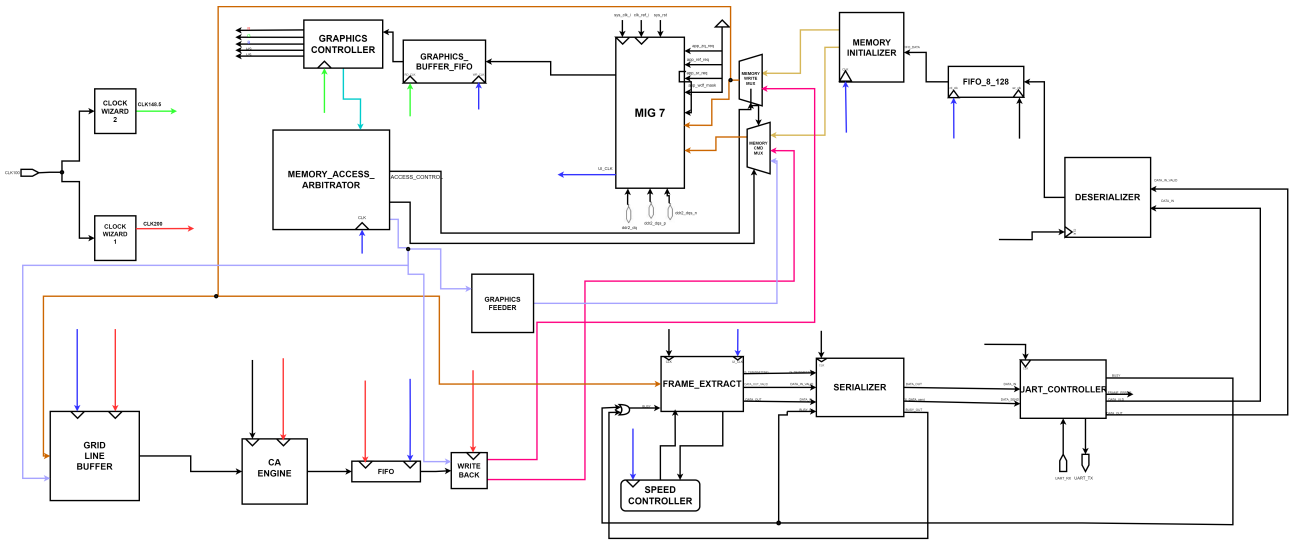


FIGURE 4.14: Milonakis Hardware Design

4.2 Emmanouil Mylonakis's Design

The graduate student, Emmanouil Mylonakis, improved upon the work of Nikolaos Kyparissas by creating a new re-programmable framework during his thesis. This framework is more adaptable to users compared to previous versions. It utilizes Xilinx CAD tools and VHDL to generate the design's bit files. Consequently, the hardware design was reformed to better integrate with the CAD tool.To understand the new framework may analyze the modifications that have been implemented.

4.2.1 Hardware Design Changes

The system architecture now includes two new components, the Deserializer and the Serializer, which handle converting structured data to and from byte format using Protocol Buffers. The Deserializer turns serialized (byte-encoded) data into a usable format , while the Serializer takes data and encodes it into bytes for sending out.Additionally, the SPEED CONTROLLER module has undergone significant changes and now functions as an independent unit.This module is able to dynamically control the speed of the circuit by adjusting from 0 up to 60 frames per second based on the time step supplied. This feature is very critical in controlling the processing speed, without necessarily involving the use of pushbuttons on an FPGA, onboard the FPGA itself, for instance.Working together with the SPEED CONTROLLER, the FRAME EXTRACT module is designed to capture continuous snapshots.Another key feature of the new architecture is the enhanced

CA (Cellular Automata) Engine, which can be reconfigured dynamically. It makes the CA Engine simulation model-dependent on the Deserializer's runtime parameter, giving it excellent adaption for many applications.

Prior to make the individual analysis of each module, it is essential to explain the Protobuf protocol.

4.2.2 Protobuf protocol

Protobuf stands for Protocol Buffers and represents a format for serializing structured data.This protocol designed to enhance the efficiency of processing, transmission, and data storage.Google developed it and made it open-source in 2008.Protobuf's serialization is a method that converts formatted data into a byte stream. It is the representation of information in a compact way. In Protobuf, it uses the Tag-Length-Value (TLV) encoding technique as its basic components to divide the serialization, the Tag, the Length, and the Value components.The Tag specifies the type of data determined in an established schema. The Length determines the amount the data contains, and Value consists of these data bytes that directly correspond to the amount.

Furthermore, the structure of the data is specified in .proto files. This definition includes the type of each field and an associated field number.Field types includes data types like int32, float, bool, string, and more.The field labels determine the role of each field within the message. An **optional** field can or may not be set, and if it is not set, then its absence does not affect the completeness of the message. A **required** field must be specified, otherwise, the message is treated by the protocol as incomplete.A **repeated** field can appear and usually does appear any number of times, including zero times, for all types of lists or arrays. Also, there is a **map** label used for a map or dictionary type or a key–value pair of structures.In the next example it appears the exact structure of Protobuf protocol :

```
1 message Grid {
2
3    repeated uint32 value  = 1;
4
5 }
```

In the above example, a simple message has been created, called Grid. The message has one attribute and is separated into four fields: the field label(repeated),the field type (uint32), the field name (value), and the field

number (1) .The Tag value is provided via the formula: $(fieldnumber << 3)|wiretype$ where wiretype shown in table 4.1. In this example the Tag is:

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

So,the lower three bits of a Tag represent the wire type, while the remaining bits indicate the field numbers.

TABLE 4.1: Wire Types

ID	Name	Used For
0	VARINT	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	I64	fixed64, sfixed64, double
2	LEN	string, bytes, embedded messages, packed repeated fields
3	SGROUP	group start (deprecated)
4	EGROUP	group end (deprecated)
5	I32	fixed32, sfixed32, float

Let's clarify the encoding process with an example. Because is repeated have an array of values for example: [724, 6].

The decimal number $(724)_{10}$ can be expressed in binary as $(1011010100)_2$.if it is split into group of 7-bits **101 1010100** , filled with zeros **00000101 01010100** then converted from big-endian to little-endian **01010100 00000101** and added the continuation bit **11010100 00000101**. The hex value of the final result is D4 05.

This process is repeated for all numbers in the array. The most significant bit (MSB) of each byte, known as the continuation bit, signals whether to continue reading bytes for the current number ('1') or to stop ('0'). Therefore, the message would be encoded as [0A03 D405 06], where T=[0A], L=[03], and V=[D405 06].This is the encoding process of a repeated attribute.The optional is handled the same way but the L=1.All these lead to well-organized user configurations as a result of using Protobufs so the hardware can identify the kind of data it received and then distribute it appropriately to the respective design modules.

4.2.3 Deserializer

Now that the protocol buffer definition has been successfully explained, opened the way for the understanding the deserializer module. This module

is crucial for the data processing phase, as it enables the reading and distribution of data serialized using protobuf.The deserializer is connected to the module UART controller and operates as a Finite State Machine (FSM) and begins its process in the **IDLE** state, where it waits the first byte of data, which is identified as the Tag value. Upon receiving a Tag, an internal register is updated, indicating the type of data that is currently being received. Depending on the information contained in the Tag, the FSM will either transition to the **LENGTH** state if the entry is repeated, or to the **OPTIONAL** state if the entry is optional.In the **OPTIONAL** state, the FSM continuously reads bytes until a stop bit is detected. Once this stop bit is encountered, the collected data, along with a valid signal, is passed on to a demultiplexer. On the other hand,for repeated entries, the FSM first moves to the **LENGTH** state, where it decodes the number of upcoming entries in the Value field. After this, it continues to the **REPEATED** state. The FSM stays in this state for a number of times as specified in the length field. At each pass, correctly deserialized value is provided at the output along with a valid signal. When the last entry is deserialized, the FSM returns to the **TAG** state, expecting to receive new data again.

All values that deserialize the initial state of the grid, the neighborhood weights, the transition rule,BRAM values and the time step are sent through one demultiplexer and distributed to other modules.

4.2.4 Serializer

The Serializer module, which is the second module added to the initial architecture, serving as the reverse of a Deserializer. This module is structured also as a Finite State Machine (FSM) and is strategically placed between the Frame Extract and the UART Controller, because its purpose is to extract the new state of the grid to the external environment.

The Frame Extract and UART Controller are ideally synchronized in the initial design. The Frame Extract dispatches a signal towards the UART Controller to commence data transmission whenever it has valid data at its disposal. In the event of its being busy, a signal is returned to stop the Frame Extract from dispatching any more data. This sort of coordination is basically important in enabling the Serializer to properly work with these interactions.

The FSM of the Serializer consists of four main states: two states for handling the Tag and Length as previously discussed, and two additional states

dedicated to the Value field. The grid being managed by the Serializer is set at a size of 1920 x 1080. For the transmission of each cell state, 2 bytes are always allocated since the worst-case scenario (transmitting the maximum cell value of 255) requires 2 bytes in Protobuf's format. For values less than 127, which in an 8-bit representation have a '0' as the most significant bit (MSB), only 1 byte would be typically necessary. However, in this design, even these smaller values are encoded into 2 bytes by appending 7 zeros followed by a continuation bit.

The operation begins when the Frame Extract indicates that the transmission has started through a transmits signal. Immediately, the Serializer sends the well-known Tag and Length fields to the UART Controller. Given that the Frame Extract delays before transmitting the cell values, there is enough time for these first two fields to be securely transferred. Moreover, due to the Serializer's slower processing rate (2 clock cycles per state), compared to the Frame Extract (which processes one state per clock cycle), a busy signal is also sent from the Serializer back to the Frame Extract to manage the data flow efficiently. Finally, once the "transmits" signal is deactivated, the FSM of the Serializer returns to the IDLE state, ready to handle the next data extraction when required.

4.2.5 Frame Extraction and Speed Control

These two modules already existed in the initial architecture. In the new reprogrammable architecture, some modifications were made to suit the new logic.

Initially, the Frame Extract module could only perform one snapshot extraction before needing a manual reset. To change this, the modifications were made so that the Frame Extract now starts in an IDLE state, waits for the simulation to pause, performs the extraction, and then returns to IDLE, ready for the next operation. It also includes a signal to show whether it is active or not.These changes removed the manual intervention and allowed the generation of exact frames based on the user's needs.

The Speed Controller ,in the initial design, used to be manually controlled through buttons to adjust the simulation speed or pause it for snapshots. Now, with updates for remote operation, the Speed Controller automatically

manages the simulation. It pauses the simulation when a set number of generations have been reached, triggers the Frame Extraction to take a snapshot, and then resumes the simulation after the snapshot is done.

4.2.6 CA engine

The re-programmable CA Engine implemented to integrate with a CAD tool,which together enhance the functionality and user friendly environment of the CA simulator.The CA Engine is designed to be used with a neighborhood size of 29 x 29. If the neighborhood is of smaller size, the system pads it with zeros in order to have a uniform frame.The engine also includes 29 x 29 fully-pipelined multipliers, which are employed to apply weights to each cell in the neighborhood.Further, the Transition Rule component, which was previously a part of the CA Engine, has been upgraded to a BRAM module.It stores the potential states of its neighborhood and is able to provide the next state of the central cell from its current state. As discussed earlier,types of rules exist the totalistic and the outer-totalistic rules. In totalistic rules, the next state of the central cell is determined only by the sum of the surrounding cells. In outer-totalistic rules, both the sum of the surrounding cells and the state of the central cell itself influence the next state.For these reason each of these two type is handled differently as we see above.For these reasons, each of these two types is managed distinctly, as will be explained in the next subsection.

4.2.7 CA Engine for Totalistic Rules

In the hardware developed for this cellular automaton, there's a special memory component called Block RAM (BRAM) that stores potential states a cell can take. For example, the sum of the neighboring cell states exists in a range ,then directly this sum will point into the memory address in the BRAM that holds the next state of the cell. In other words, the sum of the neighbors acts as an address pointer for retrieving the next state from the BRAM.Two versions of the CA Engine have been developed to address different cellular automata rule complexities: a 4-bit version for cellular automata with up to 16 states per cell and 8-bit weights, and an 8-bit version that supports up to 256 states per cell but utilizes 4-bit weights. Both engines produce a 12-bit integer product and contribute to a 22-bittotal sum in the worst case scenario $(29x29x255x15 = 3,216,825)$.With the memory available on the FPGA, worst-case BRAM requirements on the implementation of the

above two variants of the CA Engine would be about 12.8 Mbits for the 4-bit version and 102.9 Mbits for the 8-bit version. Obviously, these memory capacities are not realistic. The solution is to use reduced BRAMs as shown by both versions, that is 214 addresses for 4 bits per address = 65,536 bits, or 214 addresses and for 8 bits per address = 131,072 bits. The configuration uses the 14 least significant bits of the calculate the sum for addressing the BRAM, while the top 8 bits are used to detect overflow. In case of overflow, it increments an output select of a 2-to-1 multiplexer that selects between the BRAM data output and a fixed value that is specified in the transition function.

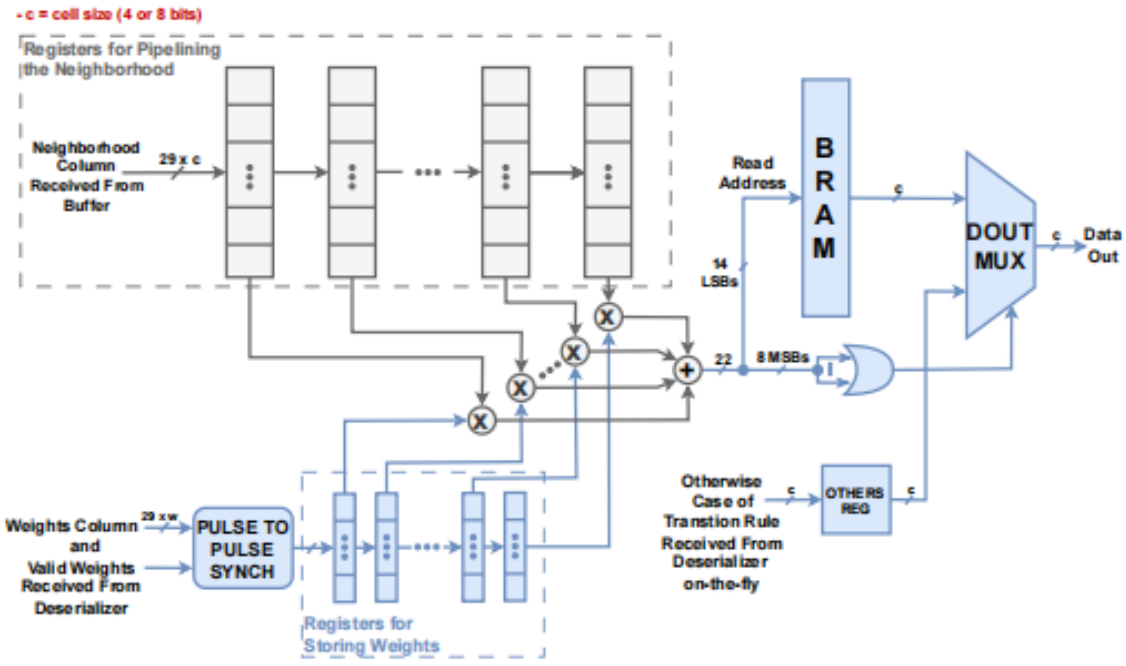


FIGURE 4.15: CA Engine's re-programmable structure for totalistic rules(Source: [18]).

4.2.8 CA Engine for Outer-Totalistic Rules

The structure proposed in the following is designed for a calculation of the next cell state in cellular automata , especially with concern to outer-totalistic rules, given the current cell state and the sum of its neighbors.The addressing method used in BRAM combines the state of the central cell with the combined state of its neighboring cells to determine the read address.The complexity depends on the cell size bit-width within the CA. For a 4-bit CA, a 14-bit address is assigned, where the 4 most significant bits encode the address of the state of the central cell and the 10 least significant bits encode the number of the total sum of the neighboring cells. For an 8-bit CA, only six least

significant bits are used for the total sum.In this case ,the BRAM divide into multiple sub-intervals, 16 or 256 sub-intervals according to required bits per cell. The central cell indicates which sub-interval should be accessed, while the total sum specifies the address within that particular sub-interval.It also requires two selection multiplexers: ADDR MUX and OVF MUX. They implement the method in which addressing modes are selected and how overflow will be addressed in the CA rules. ADDRMUX selects among various addressing plans depending on whether a totalistic or an outertotalistic set of rules is being realized, and OVFMUX deals with those overflow bits that are not intended for use inaddress calculation but can, nevertheless, influence the system.Additionally,clock domain crossing techniques are used since the Deserializer is designed to work in a different frequency domain (100 MHz) than the CA Engine (200 MHz), this technique guarantee safe data transfer between different clocking domains.

The newly integrated modules have been successfully embedded into the original hardware architecture, creating a re-configurable system that can be controlled via software. In total, six bit file images of the hardware were produced. Each type of grid (rectangular, cylindrical, and toroidal) requires three different bit files.These bitfiles are generated from Vivado and used in CAD tool that explain in the next subsection.

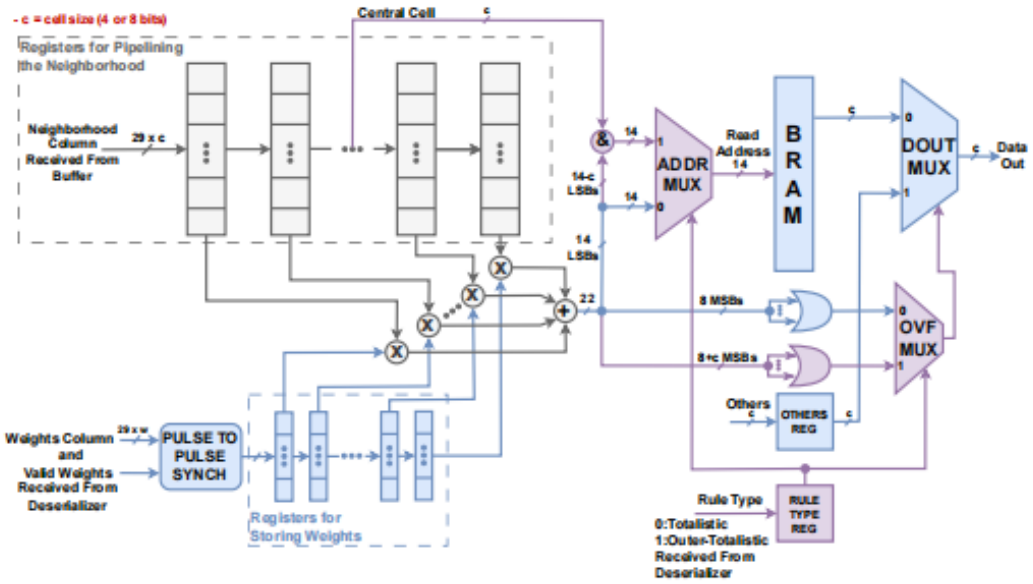


FIGURE 4.16: CA Engine's re-programmable structure for totalistic rules and outer-totalistic rules (Source :[18]).

4.2.9 CAD Tool

To facilitate the user's use of the CA accelerator, a tool was created that interacts with the tool by means of a GUI environment, this subsection mainly focuses on its back-end operations.It is a software tool that automates interaction with the hardware accelerator using a GUI. It is implemented mainly using Python due to its simplicity of use and wide range of uses in various applications. It also contains TCL script controlling the Vivado Design Suite - a family of tools for synthesizing and analyzing HDL designs.The primary purpose of the tool is to automate the initialization of a machine.The primary purpose of the tool is to automate the initialization of a machine.They may also specify parameters required for the cellular automaton such as weights, grid type and number of states. The user interaction will be through GUI that allows the user inserting the initial state for his simulation and that's done by image input. In a cellular automate the transition rule specifying how the states evolve from step to step is set within the GUI.The GUI environment can be seen in the figure4.17.It seems that the users can set the initial state by adding an image, choose the transition rule, the grid type, the number of states, the number of instances they want and the specific time moment, as well as the size and type of the neighborhood.

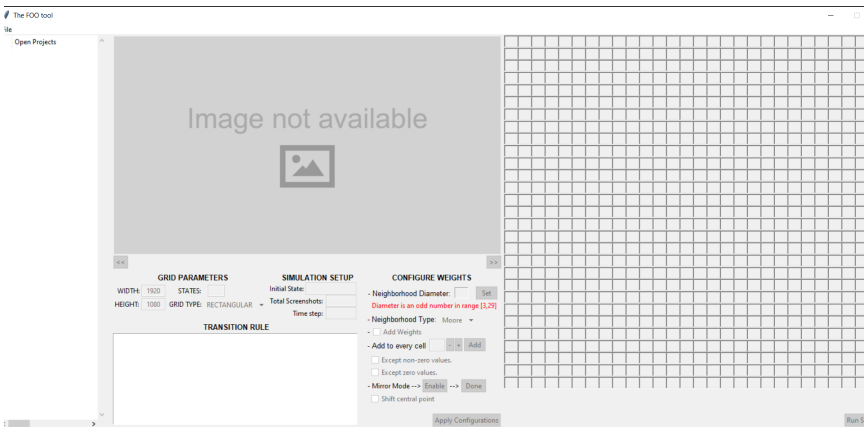


FIGURE 4.17: The GUI environment

On the operational side, the tool converts the initial state image into a data array that can be processed by the software and hardware. It handles serialization(serialized_data.bin) and deserialization of data, using Protobuf protocol that converts the data into a format suitable for transmission.The parameters that are serialized or deserialized with Protobuf are shown in the figure 4.18.

Depending on the parameters set by the user, one of six bit files is chosen and upload into an FPGA configured to carry out the specified operations.

```
syntax = "proto3";

package CAD; // Just to avoid conflicts with other project.

message DataStruct {
  optional uint32 time_step = 1;
  repeated uint32 weights = 2;
  repeated uint32 bram_values = 3;
  optional uint32 others = 4;
  optional uint32 addr_sel = 5;
  repeated uint32 grid_values = 6;
}
```

FIGURE 4.18: the parameters of Protobuf

The software communicates with the FPGA by sending and receiving data using the protocol specified by the UART. The TX for transmitting data to fpga shown in 4.19 and for RX receiving data 4.20.

```
def uart_tx():
    # Store serialized bytes to a local list.
    with open("serialized_data.bin", "rb") as file:
        serialized_data = file.read()
    file.close()

    # Instantiate/Set up serial communication.
    port_inst = serial.Serial(port='COM4', baudrate=2000000, bytesize=serial.EIGHTBITS,
                              parity=serial.PARITY_NONE, stopbits=serial.STOPBITS_ONE)
```

FIGURE 4.19: UART TX

```
def uart_rx(total_extracts, vertical_offset):

    # 1 for T flag, 4 for L flag and 2*1920*(1080-offset) bytes for the grid.
    total_bytes_to_read = 2*1920*(1080-vertical_offset) + 5

    # Instansiate/Set up serial communication.
    port_inst = serial.Serial(port='COM4', baudrate=2000000, bytesize=serial.EIGHTBITS,
                              parity=serial.PARITY_NONE, stopbits=serial.STOPBITS_ONE)
```

FIGURE 4.20: UART RX

The main purpose of this CAD tool is to help the user avoid dealing directly with the hardware. Instead, the software contains bitfiles that have been created from the mentioned re-programmable architecture. This setup allows the user to specify parameters required for the cellular automaton, while the software handles the selection of the correct bitfile,the transfer of the data and manages the communication with the FPGA.To understand the tool one example is going to be given.

4.2.10 Example Of Using CAD Tool

This example contains a totalistic rule called artificial physics.The Gui environment for that example shown in the figure4.21.

In this example the users set the initial state by adding **input1.bmp**, write the transition rule , the grid type is **Rectangular** , the number of states are **2**, the number of instances is **1 screenshot** they want and the specific time moment at **60 generations**, as well as the size of the neighborhood is**21x21** and type of the neighborhood as **concentric circle**(Custom).

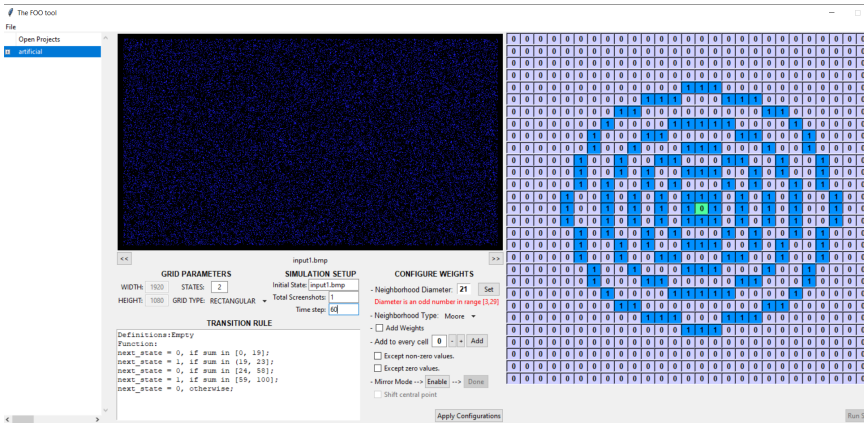


FIGURE 4.21: Artificial physics

When the user fills in the fields and runs the simulation, the CAD tool needs to select the correct bitfile and serialize the data before sending it to the FPGA. Each of these parameters helps the software system send the right data. Firstly, the grid type and the number of states aid in choosing the bitfile, in our case, it's a 4-bit rectangular one. Furthermore, all the other fields assist the protobuf in collecting data for serialization.

Each variable in the figure 4.18 represents a different type of data, and their names indicate what they contain.

1. Collect the data for the **timestep** with ID 1, including the **timestep** that user write in the interface, which in our case is 60. Therefore, the **timestep** in the serialized data is going to be 60."

2. Collect the data for the **weights** with ID 2,in the case of the **weights** that is repeated value take all the value of grid(29x29) that exists in the right side of the figure(4.21).

3. Collect the data for the **B-RAM** with ID 3, using the transition rule. Since this is a totalistic rule, the **B-RAM** values will be assigned as follows: values from [0, 19] and (19, 23] will be set to 0, values from [24, 58] will be set to 1, and values from [59, 100] will also be set to 0.

4. Collect data for the **others** with ID 4, using the transition rule as well. Assign the value of **others** according to the 'otherwise' clause of the transition rule in our example is 0.

5. Collect data for **addrsel** with ID 5 using the compiler that CAD tool contains. It Determine whether the rule is totalistic or outer-totalistic. If the rule is totalistic, set the **addrsel** to 0.

6. Collect data for the **grid values** with ID 6, starting from the initial state, which in this example is 'input1.bmp'. Use a function to convert the BMP image to an array and take this value.

All of the user configurations described above are important data that are serialized using protobuf . This data, along with the bitfile, is transmitted to the FPGA board via UART. The board performs the required calculations and sends back serialized data as a response, depending on the timestep. Subsequently, a function will convert this data back into an image according to the total number of screenshots required by the user.

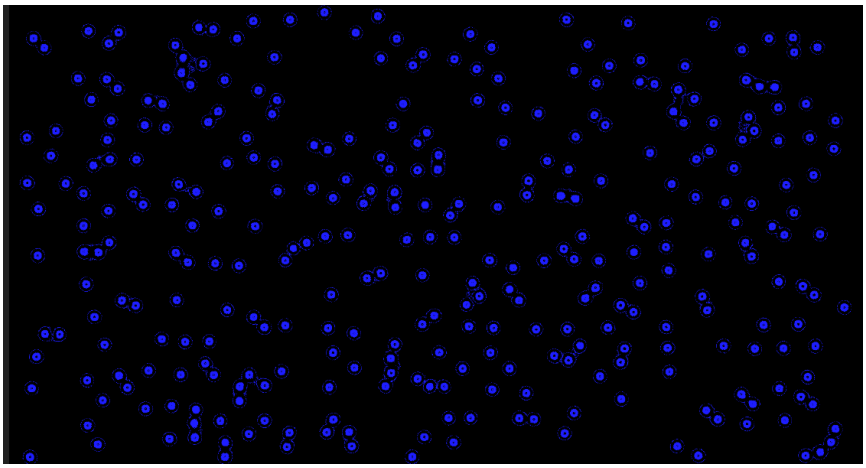


FIGURE 4.22: Artificial physics result at 600 generation

This example contains a totalistic rule. In the outer-totalistic example, the process is almost the same, with the only differences being how the values are placed in B-RAM and the setting of addrsel to 1.In this scenario,The BRAM memory is organized into sub-intervals corresponding to each possible state of a cell. If we're dealing with a system where each cell can be in one of 16 possible states (4-bit rules) or 256 (8-bits rules), the BRAM is divided into 16 sub-intervals or 256 respectively.

For example, let's say a rule specifies that when the cell's current state is 9 and the sum of the neighboring states is between 1024 and 1034, the next state should be 1. This rule would directly map to a specific range of addresses in the BRAM sub-interval for state 9, specifically addresses from 9216 to 9226 (since 9 times 1024 is 9216). All addresses within this range would be set to 1.

This is the main difference between the implementation of a totalistic and an outer-totalistic rule.

The preceding example is a basic demonstration of the tool's functionality,there's no need to explain exactly how it was coded. For those interested in exploring the code construction, it is advisable to consult Emmanouil Mylonakis's thesis [18].

A significant limitation of the Mylonakis's architecture is its inability to handle complex numerical operations required for the model's transition function, such as division. This deficiency stems from the "shrunk" design of the architecture, which, although flexible in various grid types and sizes of weights and states, fails to support complex operations defined in VHDL using Xilinx Vivado CAD tools. Also, due to limited resources in the memory segment, we have restricted the size of the weights when the Cell size is 8 bits, setting the weights to be 4 bits, meaning they can only take 16 states, which is restrictive. In contrast, the Kypparisas's architecture, although not as user-friendly, covers a broader range of operations and, by extension, more cases of cellular automata.

Chapter 5

AWS Configuration for FPGA Implementation

This chapter will address further my extension of the pioneering work initiated by Emmanouil Mylonakis in the development of a remote computational architecture using AWS hardware boards.The goal is to implement a remote system that connects to AWS boards, and with the data provided by the user, operates remotely. This makes the system even more user-friendly, as the computing user does not need to have a physical board in their space. In this chapter, it will explore in detail how we can achieve this remote implementation step by step, as well as the changes that have been made to the hardware design.

5.1 AWS Environment

First and foremost, it is essential that we familiarize ourselves with the AWS environment for FPGA implementation.AWS provides an environment tailored for FPGA (Field Programmable Gate Array) implementation through its EC2 F1 instances. These instances are specifically designed to allow developers and engineers to run FPGA designs in the cloud.

5.1.1 F1 Development Environment

The theoretical background section described exactly what an FPGA is and how it is used. In this section, we will describe how Amazon uses these reprogrammable hardware devices to allow users to create custom processors/accelerators that provide optimized compute.To understand this, we need to look at which family of FPGAs Amazon uses and their characteristics.

Amazon EC2 F1 FPGA instance family

Amazon EC2 F1 instances provide customizable hardware acceleration using FPGAs, featuring Intel Xeon processors, NVMe SSD storage, and up to 8 Xilinx UltraScale+ FPGAs per instance. These instances are ideal for applications requiring intensive computation and data processing.

Instance Size	FPGA count	FPGA memory DDR-4 (GiB)	Number of host CPUs (vCPUs)	Host memory (GiB)	NVMe instance storage (GB)	Network bandwidth
f1.2xlarge	1	4 x 16	8	122	470	Up to 10 Gbps
f1.4xlarge	2	8 x 16	16	244	940	Up to 10 Gbps
f1.16xlarge	8	32 x 16	64	976	4 x 940	25 Gbps

FIGURE 5.1: FPGA Features

FPGA Acceleration Using EC2 F1

FPGA acceleration on EC2 F1 instances is a process that achieves significant improvement in computational performance by throwing in the flexibility accorded through the programmability of FPGAs. It therefore ensued with a number of distinct steps:

1. **Deploying an OS Image:** An Amazon Machine Image (AMI) tailored for FPGA development is launched, providing the necessary operating environment.I t is the actual operating system that used to access the Fpga environment.

2. **Loading FPGA Images (AFI):** FPGA designs are compiled into Amazon FPGA Images (AFIs), which are then loaded onto the F1 instance via AWS APIs.An Amazon FPGA Image (AFI) is like to a bitstream file

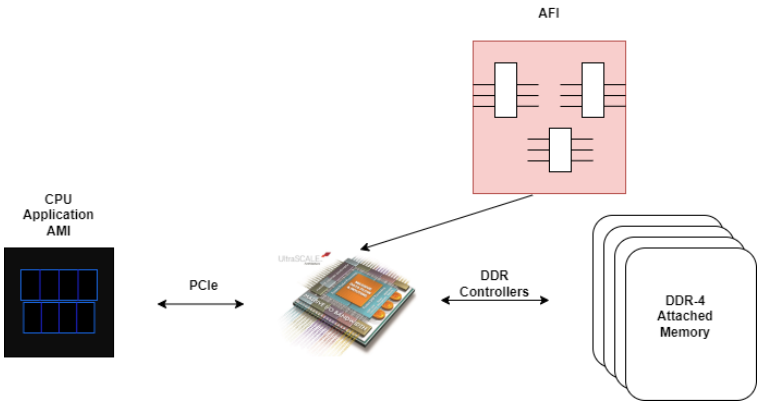


FIGURE 5.2: F1 FPGA Acceleration Process Flow

used in traditional FPGA development. It represents the compiled configuration of the FPGA design, including the specific logic and interconnections that will be implemented on the hardware.

3. **Executing Host Applications:** The host application communicates with the FPGA using the AWS FPGA SDK, enabling specific computations to be offloaded to the FPGA hardware.

5.1.2 EC2 F1 Instance Launch Requirements

Region

Firstly, It is essential to be located in a region that supports these instances. Amazon EC2 F1 instances are available for use in several locations, in Europe (Frankfurt and London), Asia Pacific (Sydney), and China (Beijing) AWS Regions. They are also available in US East (N. Virginia), US West (Oregon), Europe (Ireland), and AWS GovCloud (US) AWS regions. My implementation is located in Frankfurt(eu-central-1).

FIGURE 5.3: Region Configuration

AMI

FPGA Developer AMI has a pre-configured environment for developing FPGAs in the CentOS7 operating system. The setup is meant to make easier the whole process of developing, testing, and deploying applications that run on FPGAs in the cloud. The CentOS7 version comes installed with all

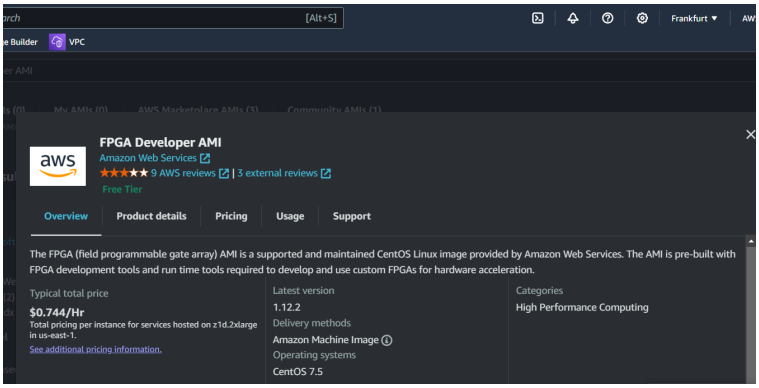


FIGURE 5.4: AWS AMI IN AWS MARKETPLACE

tools and libraries necessary for the development of FPGAs.It comes pre-installed with Xilinx Vitis and Vivado across several versions, and it's tightly integrated with AWS services.

NICE DCV

The FPGA Developer AMI does not include a graphical user interface (GUI). it is designed to be used through a command-line interface,as shown in figure 5.5.However, by using NICE DCV, adding a graphical user interface to the system to make the environment more user-friendly .NICE DCV is a high-performance remote display protocol designed for secure remote desktops and application streaming from the cloud or data center to any device. It functions by installing the server software of NICE DCV on a server, running applications on that server, and then streaming the visual output of those applications to a client device. It enables desktop sharing, video compression, multi-screen setup, collaboration tools, and therefore features make it suitable for graphics applications.

FIGURE 5.5: AMI terminal

5.2 Steps Before Launching EC2 F1 Instance

It was discussed and explained in the previous section what requirements are needed to launch an EC2 instance. To sum up, the requirements are:

1. **Region**
2. **AMI**
3. **NICE DCV**

These requirements were previously analyzed to understand the steps that are necessary for launching the EC2 instance, which will be discussed in detail above.

Steps that need to follow before launching EC2 F1 Instance:

1. **Log in to Management Console**
2. **Choose the Region drop-down menu.The region that is closest to your location and supports F1 instances.**

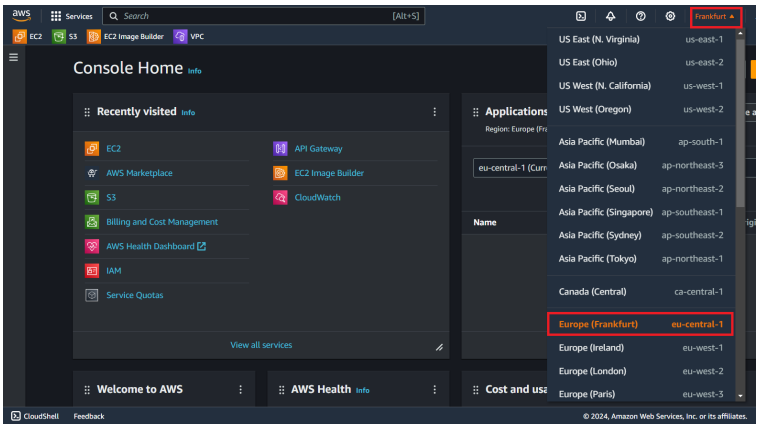


FIGURE 5.6: Regions

3. **Click "Services" in the "Compute" Category and then Click "EC2."**

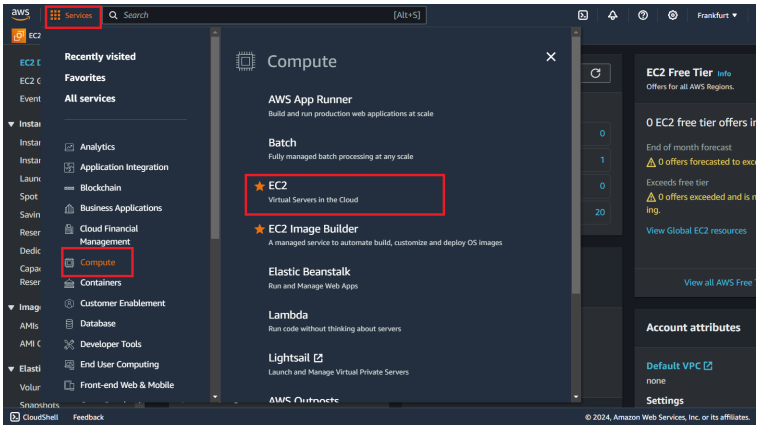


FIGURE 5.7: Find EC2 instance

4. **Create Security Group**
   - **In EC2 go to “Security Groups”**
   - **Click on “Create a Security Group"**
   - **Add a Security group name, a Description and a VPC and edit the rules in the Security Group setting Inbound.**

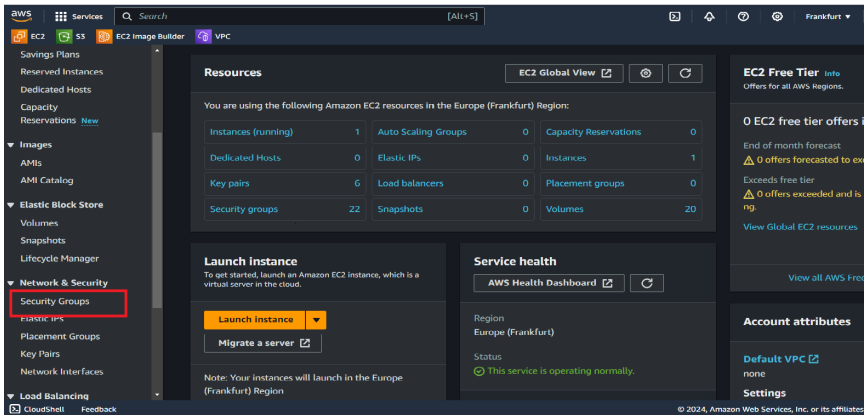


FIGURE 5.8: Find Security Group

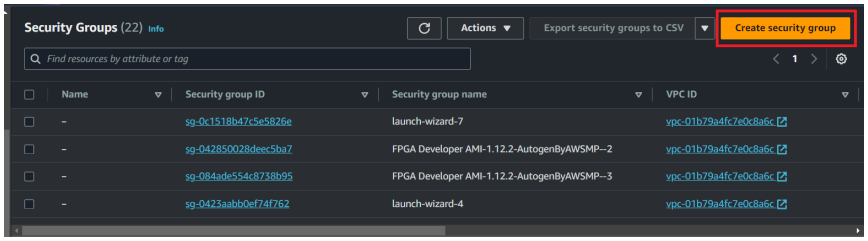


FIGURE 5.9: Find Security Group

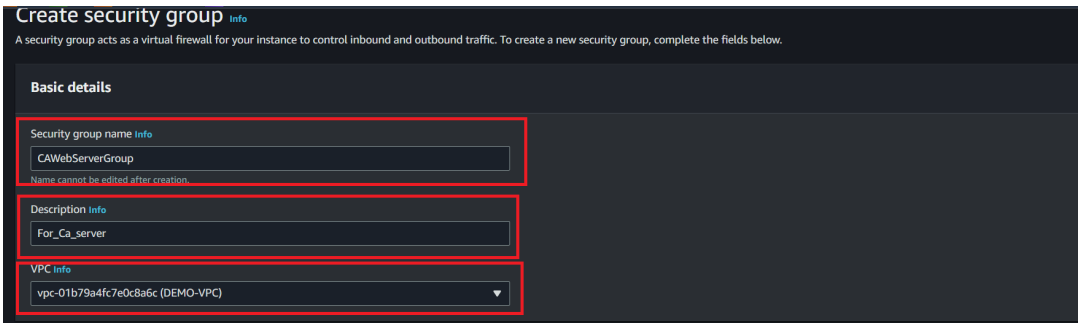


FIGURE 5.10: Example of a security group for F1 instances

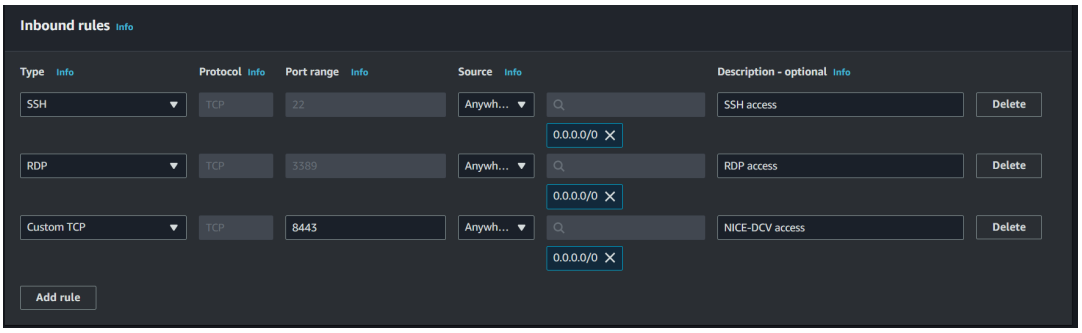


FIGURE 5.11: Inbound Rules

System administrators use SSH for managing systems and applications remotely, allowing them to log into another computer over a network, execute commands, and move files from one computer to another.From the other hand,RDP is used for remote desktop sessions, which allows users to control a desktop environment of another computer from a remote location.Final,By using the port 8443, it ensures that the GUI environment is accessible through a secure, encrypted connection.

5. **Create a Key pair** Go to **EC2 menu** again and selecting the **Key Pairs** and then **"Create Key Pair**".

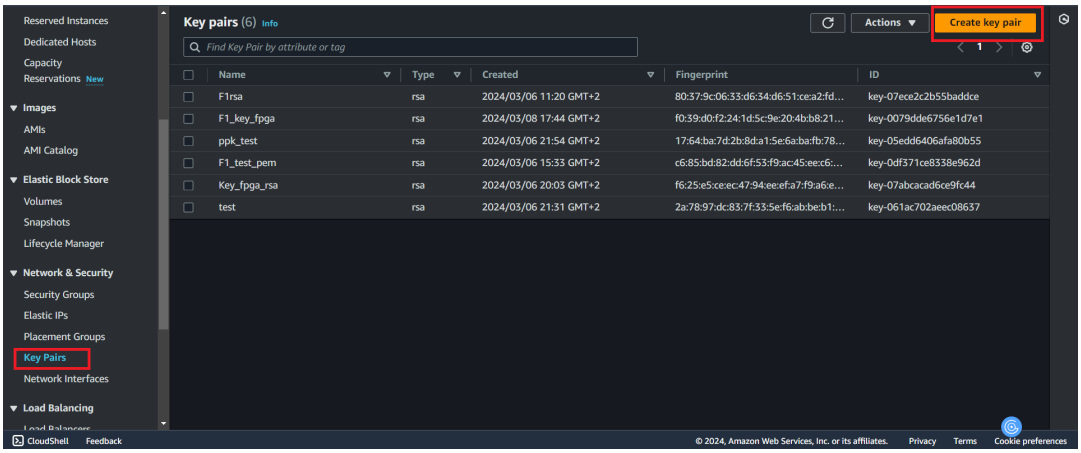


FIGURE 5.12: Key pair Creation

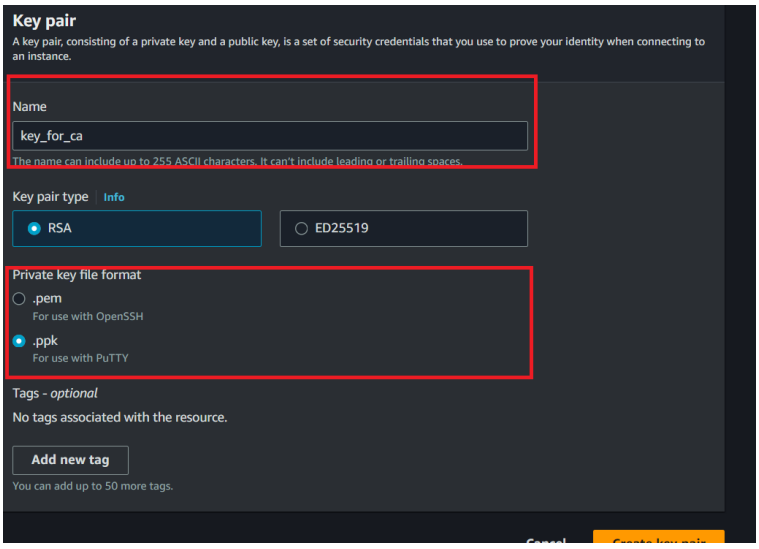


FIGURE 5.13: Key pair Creation

The private key name is for example `Key_for_ca`is typically chosen by you and can be any alphanumeric string. After creating this key pair,

the private key file will be downloaded automatically. It's important to save this file securely on your computer.Additionally, choosing between .pem and .ppk formats will depend on whether you are using OpenSSH or PuTTY.

5.3 Launching EC2 F1 Instance

1. **First of all Click "Services" in the "Compute" Category and then Click "EC2".**

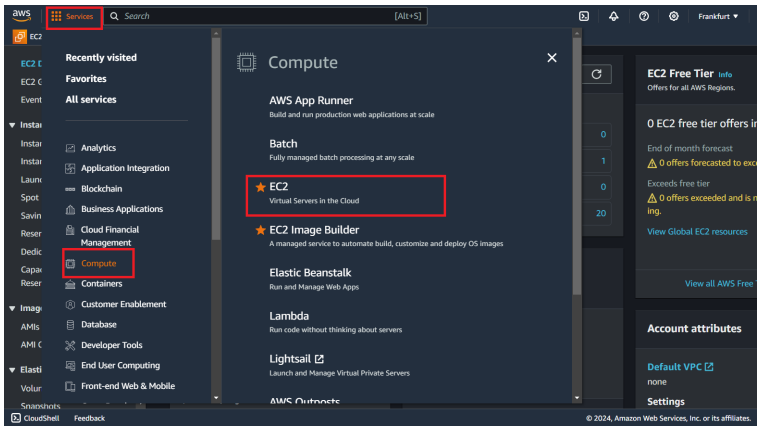


FIGURE 5.14: Find EC2 instance

2. **Click on "Launch instance".**

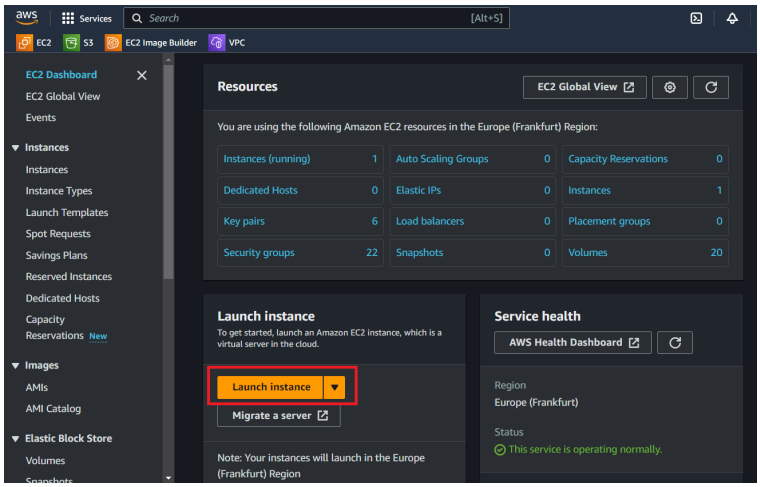


FIGURE 5.15: Launch EC2 instance

3. **Assign a name to the instance, for example,** `CA_SERVER`**.**

4. **To find the correct AMI for FPGA need to click the "Browse more AMIs".**

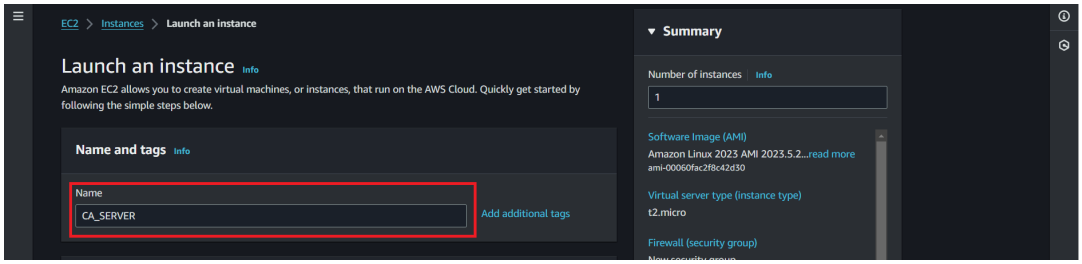


FIGURE 5.16: Assign a Name

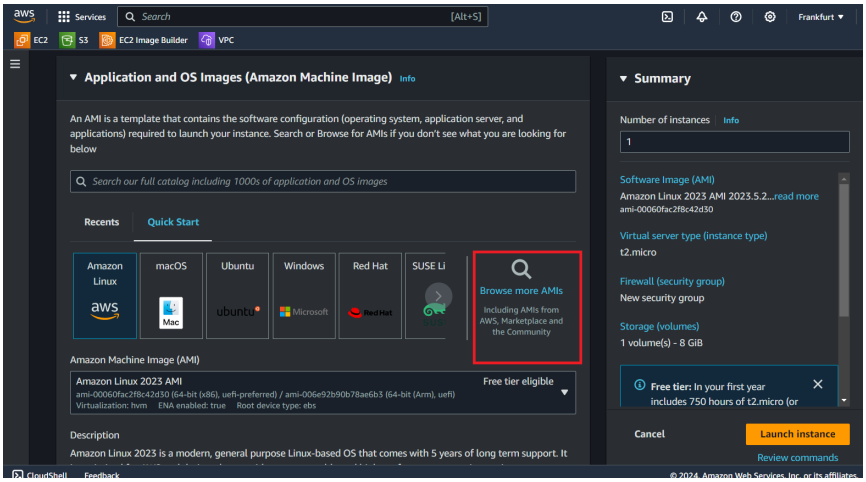


FIGURE 5.17: Browse AMI

5. **Go to the search box at the top left of the screen,Type "fpga" into the box and then look at the "AWS Marketplace AMIs" section.Once you find the AMI that suits your needs from the list, click the "Select".**

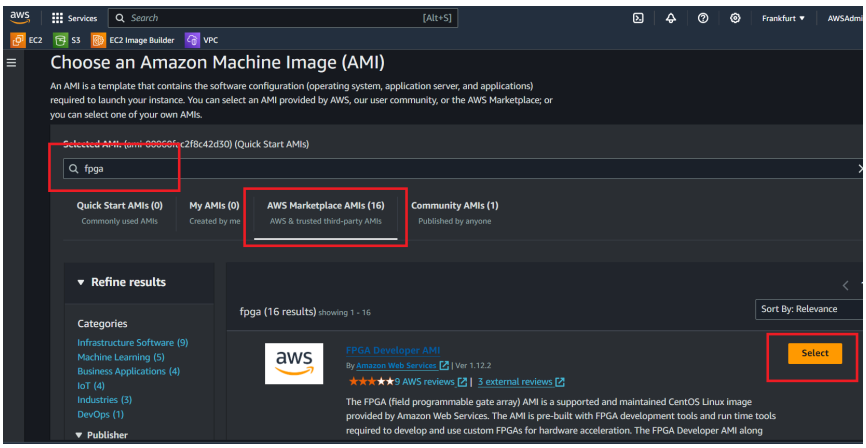


FIGURE 5.18: Find the AMI that suits you

6. **Navigate to the 'Instance Type' search box labeled 'F1' and find the 'f1' with the specific configurations related to FPGA instances as shown in Table 5.1. Observe the compatibility message stating, 'The zone (eu-central-1a) does not support this instance type.**

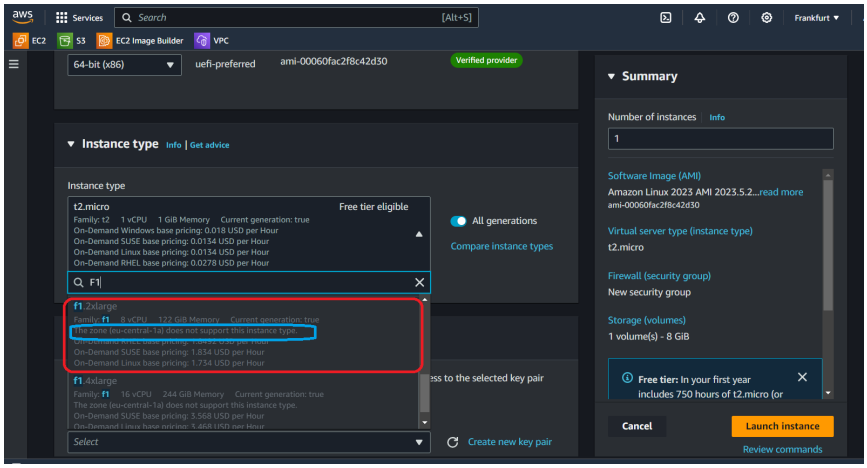


FIGURE 5.19: Instance type

If this problem occurs in the zone you have selected, resolve it by navigating to the subnets and creating a public subnet in another zone of this region.Many tutorials exist on the internet for creating subnets correctly.

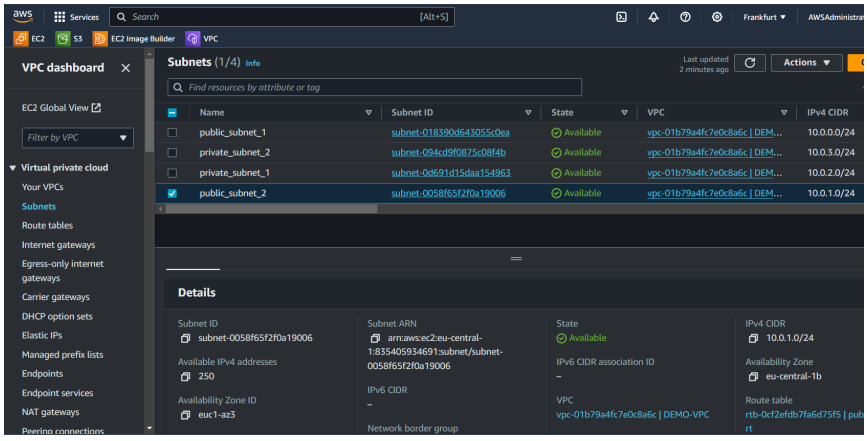


FIGURE 5.20: subnets creation

7. **After configuring the instance type, proceed to the "Key pair" section. In the field labeled "Key pair name," type the name of the key pair you wish to use, such as `Key_for_ca`.This name refers to the key that was created in subsection 5.2 step 5.**

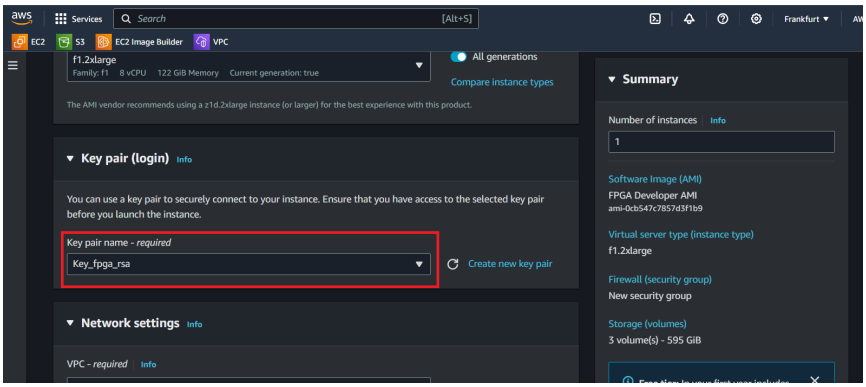


FIGURE 5.21: Key pair section.

8. **Network settings**

   - **VPC (Virtual Private Cloud):select "DEMO-VPC" with a specific VPC ID. This sets the network within which your instance will operate.**
   - **Subnet: There are two public subnets available, and you've selected one of them, `"public_subnet_2`." Subnets determine the IP range and availability zone that your instance will operate within, and choosing a public subnet usually implies that instances placed here can be directly accessible from the internet.More details are provided above on how to create the subnets.**
   - **Auto-assign public IP: You have enabled this option, which means that AWS will automatically assign a public IP address to your instance, allowing it to be reached from the internet.**
   - **Firewall (security groups): You are prompted to either create a new security group or select an existing one.In the 'Select Security Group' section, use the security group created in Step 4 of Subsection 5.2.**

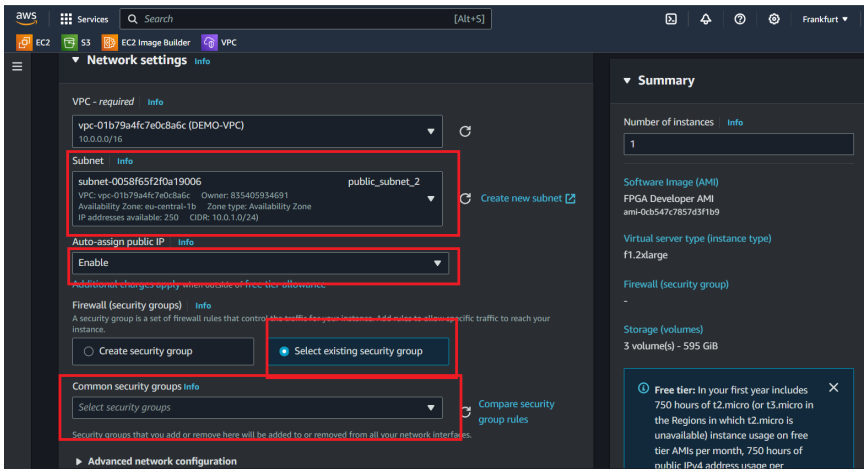


FIGURE 5.22: Network Settings configuration.

9. **Configure storage**

   - **Root Volume (120 GiB, gp3): This is the primary storage for the operating system and applications. The volume size is adjustable based on your needs.**
   - **Additional EBS Volume (5 GiB, gp2): This smaller, general purpose (gp2) volume is likely used for less intensive storage needs.The volume size is also adjustable based on your needs.**

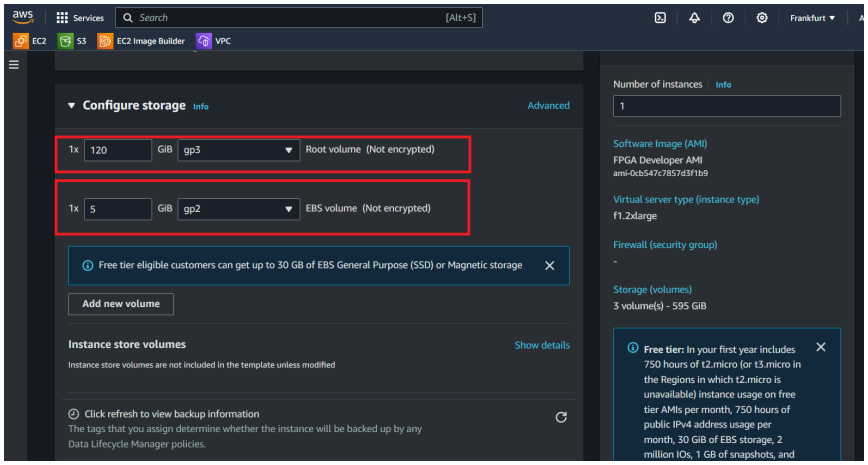


FIGURE 5.23: Storage configuration.

Gp3 is a General Purpose SSD, suitable for a balance of performance and cost. Ideal for most general-purpose workloads.On the other hand, gp2 is an Older type of General Purpose SSD, typically cheaper but with potentially lower performance than gp3.

These are the basic configurations needed to create an F1 EC2 instance that works with FPGA. There are more advanced configurations available if you need something more specific, but for a simple implementation, these should be sufficient.After that Click on **"Launch instance"**.

5.3.1 Connecting EC2 F1 Instance

1. **Select the "`CA_SERVER`" instance.**
2. **Right-click on the instance and select "Start Instance".**
3. **Wait until the instance state changes to "Running" and then use the "Connect" button to establish a connection to the instance. Depending on the instance configuration, this could involve SSH for Linux/UNIX systems or RDP for Windows systems.**

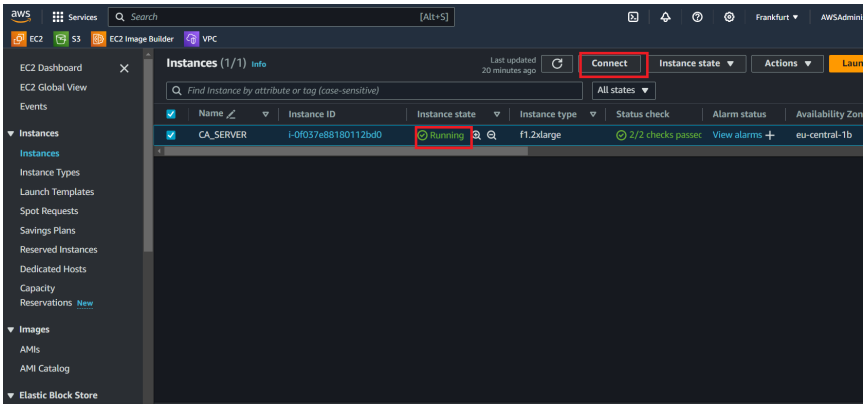


FIGURE 5.24: Connect to Instance

4. **In the "Connect Instance" Window select "SSH Client" and follow the Steps.**

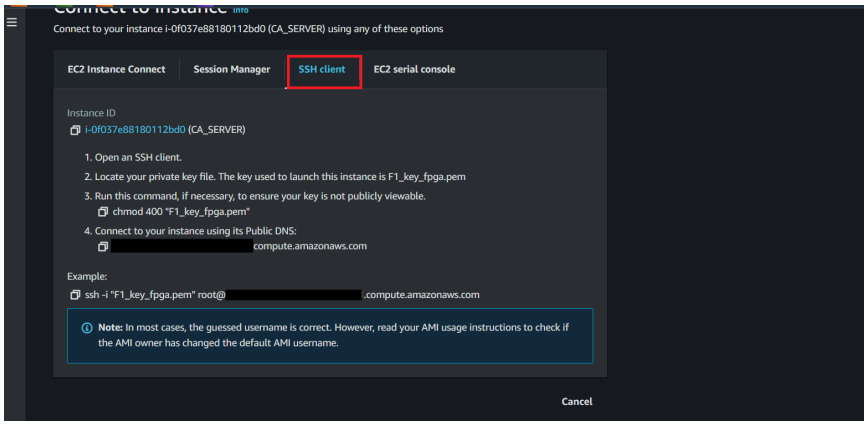


FIGURE 5.25: SSH Client

5.3.2 Putty Configuration

For establishing SSH connections, there are several methods available, one of which is PuTTY. This subsection details the configuration process of PuTTY for connecting to an instance that has been created.

1. **Download the PuTTY application in your Desktop.**
2. **Start the PuTTY application on your computer.**
3. **Configure the Private Key:**
    - **In the "Auth" section, you'll see a field labeled "Private key file for authentication".**
    - **Click the "Browse..." button next to this field.**
    - **Navigate to the directory where your private key file (.ppk) is saved.**

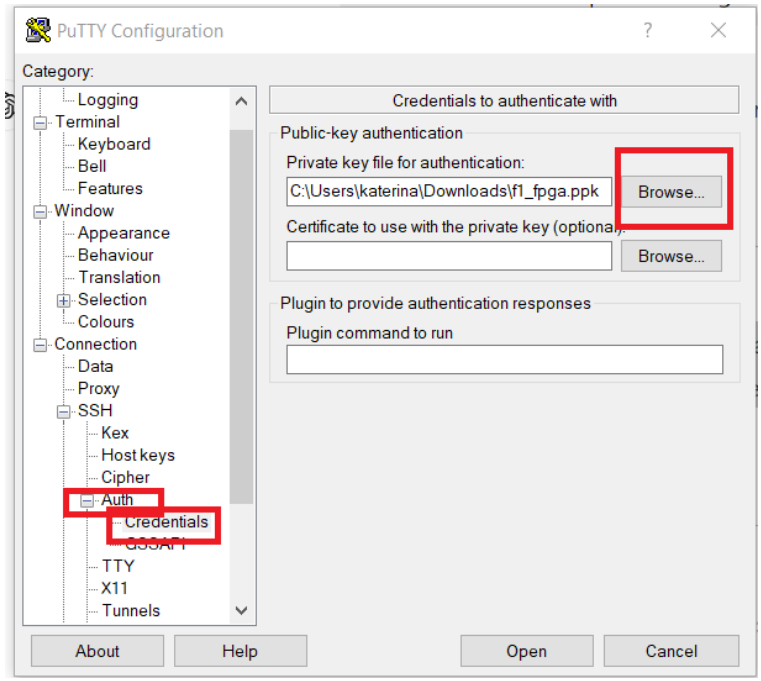


FIGURE 5.26: Putty Authentication

4. **Openning an SSH Connection**

   - In the "Session" section, In the "Host Name (or IP address)" field, enter the username and IP address or hostname of the server you want to connect to, in the format **username@hostname**(centos for these kind of instances). For example, centos@ec2-35-159-33-183.eu-central-1.compute.amazonaws.com. The **ec2-35-159-33-183.eu-central-1.compute.amazonaws.com** is the Private IP DNS name of the instance.
   - **Ensure the "Port" is set to 22, which is the default port for SSH connections.**
   - **Verify that the "Connection type" is set to "SSH". This specifies the protocol PuTTY will use to connect to the remote server.**
   - **In the "Saved Sessions" field, you can type a name for this session configuration, such as "aws-test".**
   - **Click 'Save' to store these settings. This allows you to quickly load them in the future by selecting the session name and clicking 'Load'. If the Private IP DNS name has changed, it may also change in the saved settings."**

- **Once all settings are configured, click "Open" to initiate an SSH connection to the specified server.**

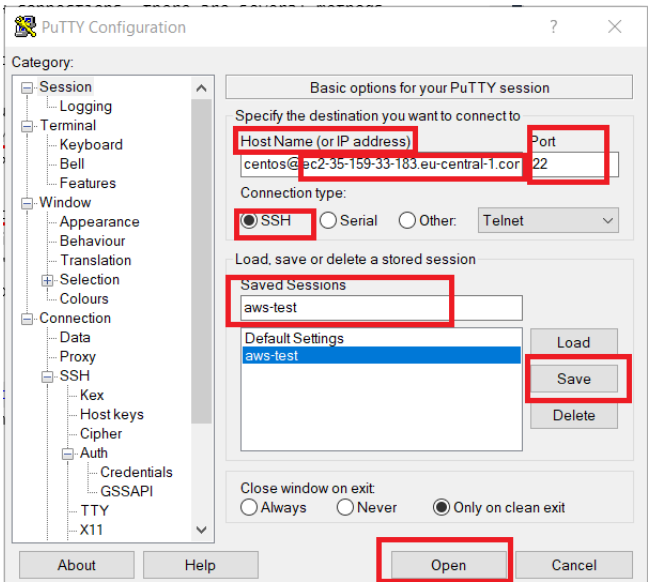


FIGURE 5.27: Putty Authentication

The terminal should look like the one shown in Figure 5.5 after all the above configurations are completed.

5.3.3 Setting up an AWS CLI environment

1. Create IAM Policies.
   First, two types of IAM Policies are needed: **one for FPGA Image Creation, FPGA Image Description**, and another for **allowing users, applications, or services to retrieve (read) objects from a specific S3 bucket**.Above, we will provide step-by-step instructions to access and manage Identity and Access Management (IAM) settings.

   - Click on the "Services" menu at the top left corner of the AWS console.
   - In the list that appears under "Services," look for the "Security, Identity, Compliance" category.
   - Click on "IAM" to open the IAM dashboard.
   - Once in the IAM dashboard, click on "Policies" in the navigation pane on the left side.

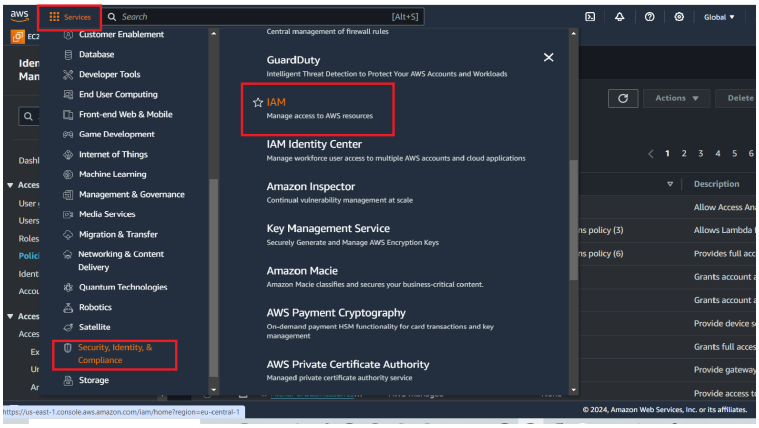


FIGURE 5.28: IAM

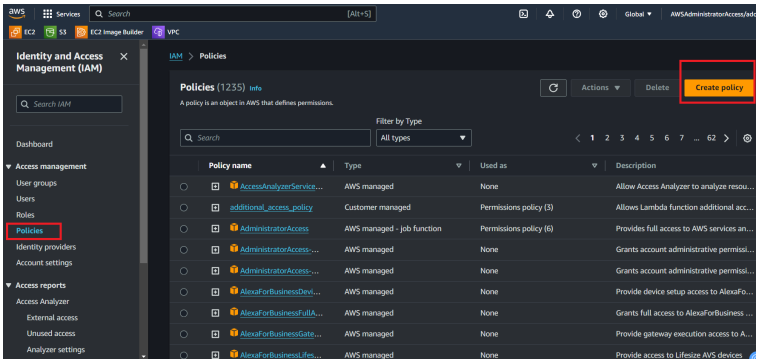


FIGURE 5.29: IAM policies

- On the policies page, click the "Create policy" button located at the top of the page. This will start the process of creating a new access policy.
- In the "Specify permissions" interface, click on the "JSON" tab. This allows you to directly enter the JSON policy document.

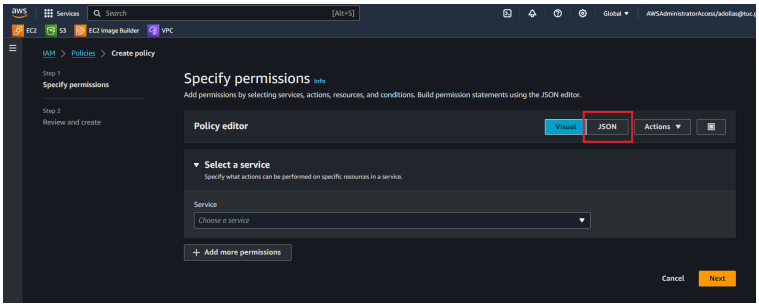


FIGURE 5.30: Specify Permissions

- In the JSON editor, you will see the policy structure where you can add or modify permissions.
- Update or modify the JSON code as necessary.

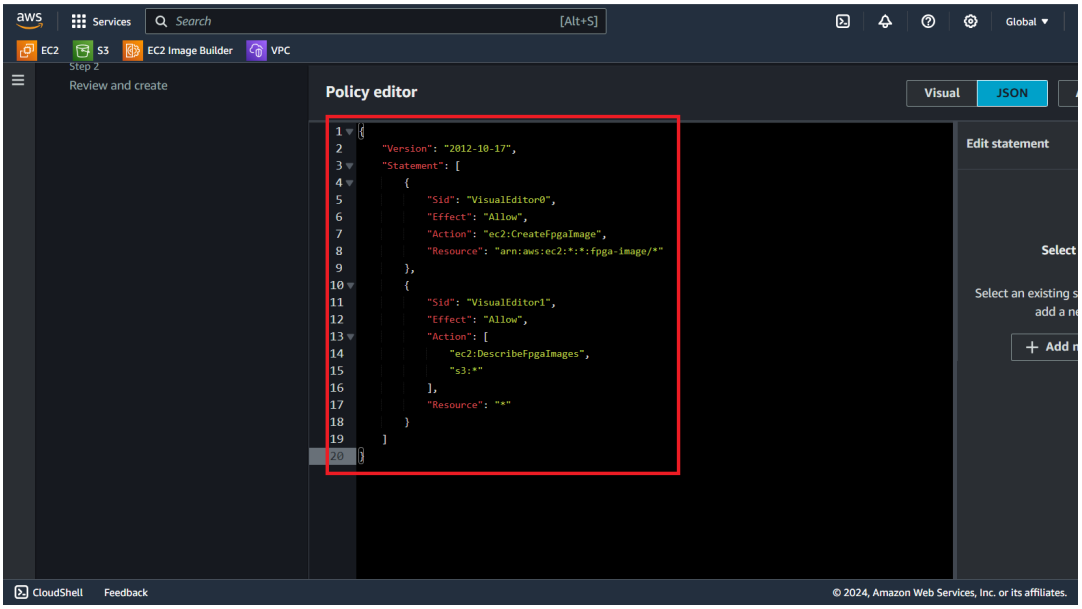


FIGURE 5.31: JSON Code for FPGA Image

This code enables the creation of FPGA images in the EC2 service and applies all FPGA images across all regions and accounts. Additionally, using DescribeFpgaImages allows viewing the details of FPGA images. This permission is necessary to create FPGA images and use the F1 instance.

- After that, enter a name for your policy in the "**Policy name**" field, such as **S3access**.
- Optionally, add a description for the policy in the **"Description**" field.

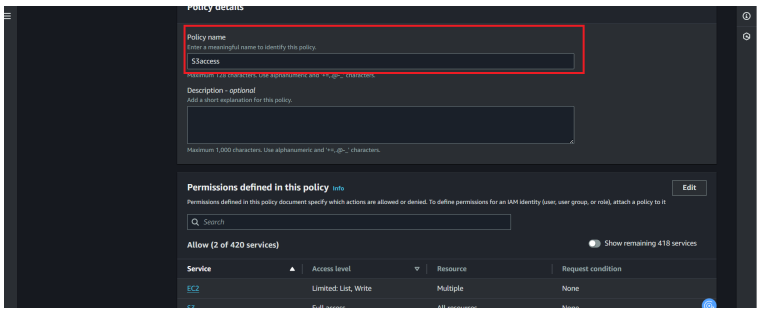


FIGURE 5.32: Create Policy

- After that Click on "**Create Policy**".
  A new policy needs to be created that allows the user, role, or service it is attached to, to retrieve any objects from the specified S3 bucket in the a specified region, following the same steps as the

previous one and writing the code below:

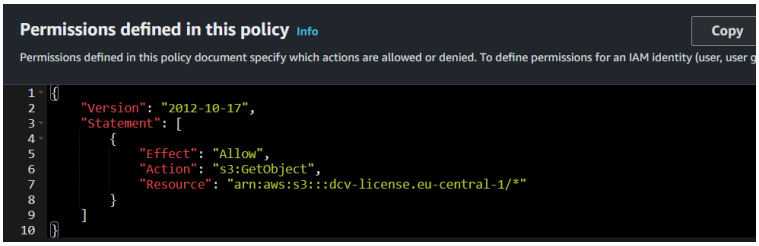


FIGURE 5.33: JSON Code for S3 bucket

Also, enter a new name for this policy ,such as `NICEDCV_Linc` and click on "**Create Policy**".

2. Create IAM Roles.
   - In the IAM dashboard, click on "Roles" in the navigation pane on the left side.
   - Click the "Create role" button located at the top right.
   - From the "**Choose a service or use case**" menu, select a service that will use this role. For instance, if you are creating a role for an EC2 instance to perform specific actions, select "**EC2**" from the "Commonly used services" list and then click on "**Next**".
   - After specifying the trusted entity type and reaching the "Add permissions" step , browse through the list of available policies.
   - Attach the policies that were created before.
   - Click on "**Next**".
   - In the "**Role name**" field, enter a name for your role. For example, "**S3accessRole**" suggests this role has permissions related to Amazon S3.
   - In the **"Description**" field, provide a detailed explanation of the role. This should include information on what the role is used for and why it's needed.
   - Then,click on "**Create Role**" button.
3. Attach IAM roll to EC2
   - Select "Instances" from the left menu of the EC2 screen and select the F1 instance you created before. From the actions menu, select Actions > Security > Modify IAM Role.

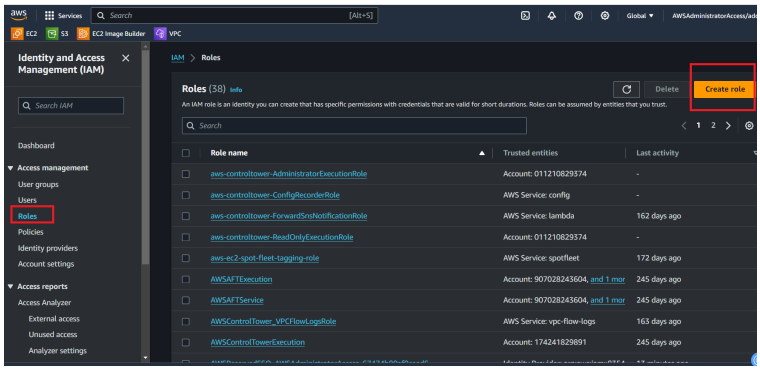


FIGURE 5.34: IAM Roles

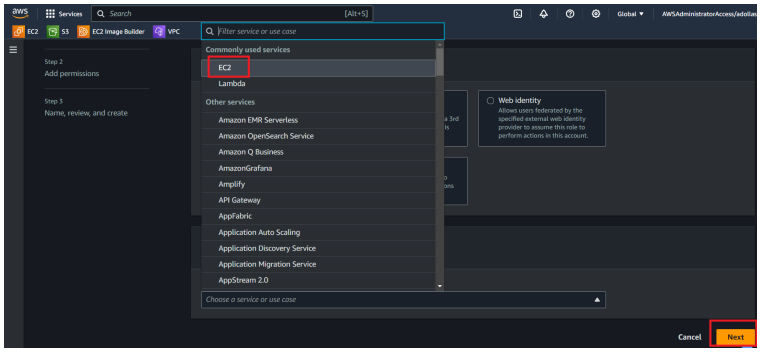


FIGURE 5.35: Create IAM Roles

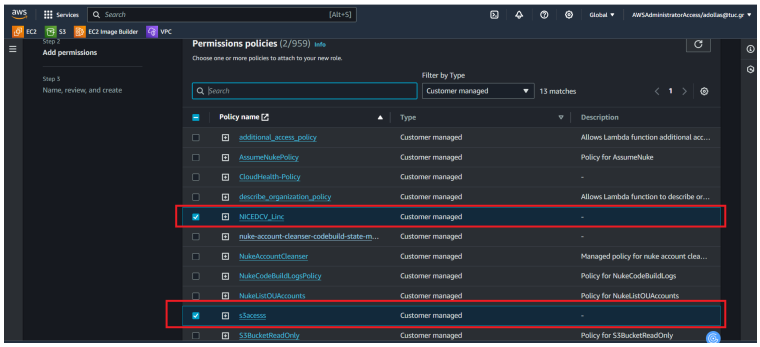


FIGURE 5.36: Select IAM Policies

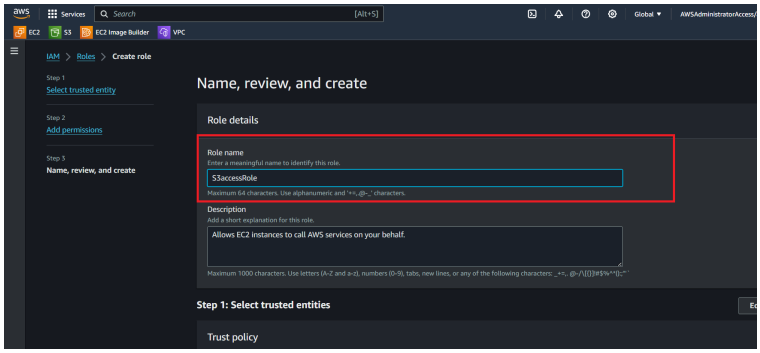


FIGURE 5.37: Name IAM Role

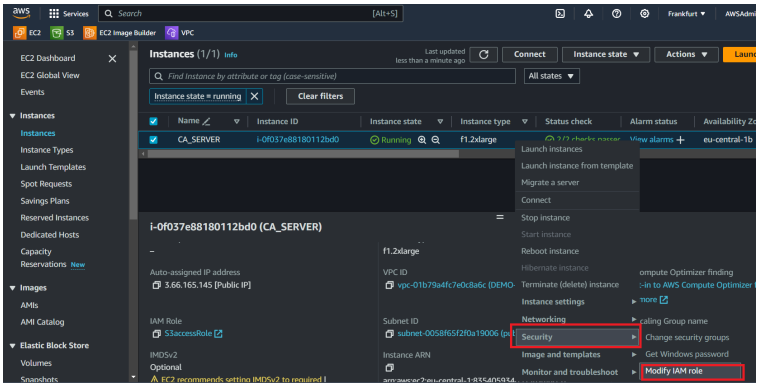


FIGURE 5.38: Attach IAM roll to EC2

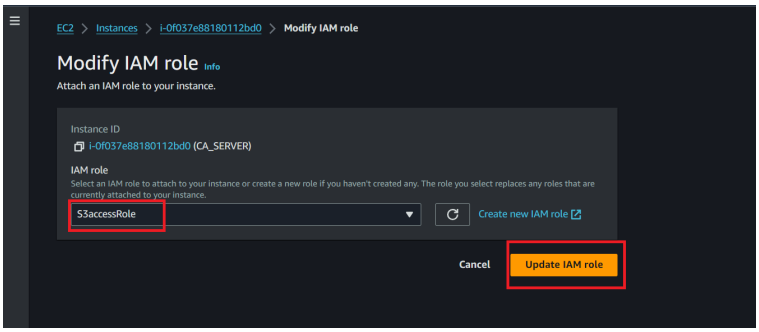


FIGURE 5.39: Attach IAM roll to EC2

- On the Change IAM Role screen, select the IAM role you just created and Click on "**Update IAM Role**" button.

The previous steps are for making an IAM Roles for the EC2 instance,The reason that need a IAM role is that IAM roles in AWS let you set up permissions for accessing AWS services without tying them to a specific user. Instead, entities like users, applications, or AWS services (like EC2) can take on a role temporarily to get the permissions they need to work with AWS resources. This way, they can do their tasks securely without needing a permanent link to a specific user.

5.4 GUI FPGA Development Environment with NICE DCV

In Section 5.2, the NICE DCV was explained. Now, its installation and usage will be described step by step.

1. In the AWS FPGA Terminal (Image 5.5), install the **NICE DCV prerequisites** for CentOS 7 by typing the following commands in the terminal:

```
$ sudo yum -y install kernel-devel
$ sudo yum -y groupinstall 'Server with GUI'
$ sudo yum -y groupinstall "GNOME Desktop"
$ sudo yum -y install glx-utils
$ sudo systemctl isolate multi-user.target
$ sudo systemctl isolate graphical.target
```

2. Install NICE DCV Server:**Download and unpack**.

```
$ sudo rpm --import https://d1uj6qtbmh3dt5.cloudfront.net/
    NICE-GPG-KEY
$ wget https://d1uj6qtbmh3dt5.cloudfront.net/2021.1/Servers
    /nice-dcv-2021.1-10598-el7-x86_64.tgz
$ tar -xvzf nice-dcv-2021.1-10598-el7-x86_64.tgz
$ cd nice-dcv-2021.1-10598-el7-x86_64
```

3. **Install and enable**

```
$ sudo yum install nice-dcv-server-2021.1.10598-1.el7.
    x86_64.rpm nice-xdcv-2021.1.392-1.el7.x86_64.rpm
$ sudo systemctl enable dcvserver
$ sudo systemctl start dcvserver
```

4. Now change a configuration setting in dcv in order to login using **username:password** combo.

```

$ sudo sed -i 's/#authentication="none"/authentication="
    system"/' /etc/dcv/dcv.conf
$ grep 'authentication=' /etc/dcv/dcv.conf
```

5. And finally, it needs to restart the service to propagate the changes

```
$ sudo systemctl restart dcvserver
$ sudo systemctl status -f dcvserver
```

6. Setup the password for the "**centos**" user so that we can login through NICE DCV.

```
$ sudo passwd centos
```

7. In addition to the Security Groups rule,**Create a firewall rule**.

```
$ sudo yum install firewalld
$ sudo systemctl start firewalld
$ sudo systemctl enable firewalld
$ sudo firewall-cmd --zone=public --add-port=8443/tcp --
    permanent
$ sudo firewall-cmd --reload
$ sudo firewall-cmd --list-all
```

8. Create a virtual session to connect to. This needs to be done every time you want to access the NICE DCV server.

```
$ dcv create-session --type virtual f1-workshop-nicedcv-
    session
```

9. Connect to the DCV Remote Desktop session

   - Using the NICE DCV Client:
     Download and install the DCV Client, then use the Public IPv4 address of the EC2 instance to connect.

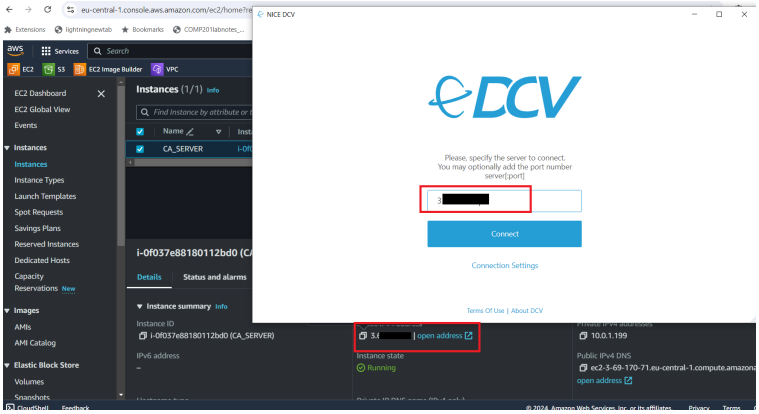


FIGURE 5.40: NICE DCV IP

   - login with **Username** and **Password** that setup in Step 6.

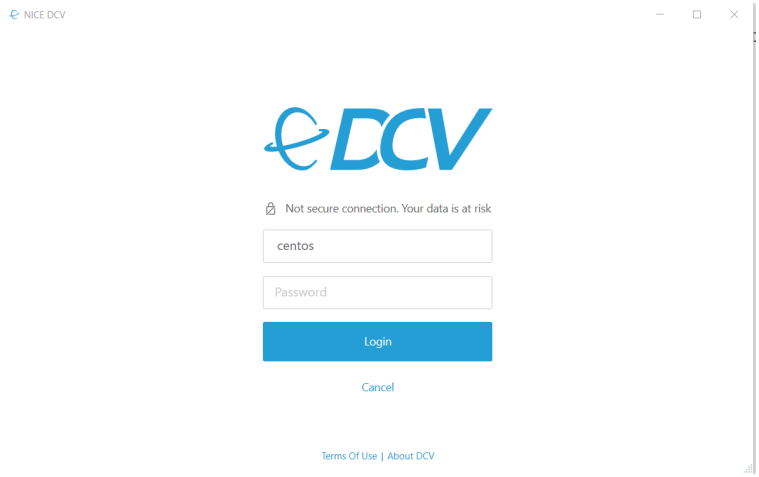


FIGURE 5.41: NICE DCV login

10. Logging in should show you your new GUI Desktop:

FIGURE 5.42: NICE DCV GUI

5.5 Setting up the HDK development environment

Setting up an HDK (Hardware Development Kit) development environment involves several key steps, each crucial to successfully developing, testing, and deploying hardware-based projects.It needs to create, configure, and Test the HDK development environment.

1. Configuring the Instance for Working with HDK

   - Open a new terminal by right-clicking anywhere in the Desktop area and selecting Open Terminal.Configure the AWS CLI as follows:

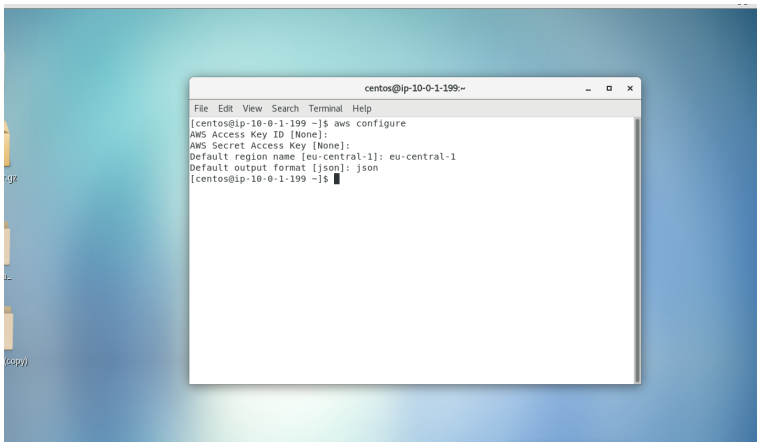


FIGURE 5.43: AWS configuration

2. Install the HDK and setup environment.
   To clone the AWS FPGA HDK to your instance using the FPGA Developer AMI, execute the following commands:

```
1 $ AWS_FPGA_REPO_DIR=/home/centos/src/project_data/aws-fpga
```

```
$ git clone https://github.com/aws/aws-fpga.git
   $AWS_FPGA_REPO_DIR
$ cd $AWS_FPGA_REPO_DIR
$ source hdk_setup.sh
```

Sourcing `hdk_setup.sh` configures your environment to use the HDK by:

- Setting required environment variables used throughout the HDK examples.
- Downloading DDR simulation models and DCP(s) from S3.

After that, it will create a folder named `project_data` containing guides, code, and reference materials to help you understand and utilize AWS FPGA technology. It may be helpful to navigate and familiarize yourself with these resources to enhance your understanding.

3. Configuring Vivado Tcl Initialization and Environment Variables
   - Step 1: Open or Create the Tcl File
     (a) In the same terminal as the one above.
     (b) Navigate to the Vivado Directory:
     ```
     $ cd ~/.Xilinx/Vivado
     ```
     (c) Check for the file using:
     ```
     $ls
     ```
     (d) If neither 'init.tcl' nor '`Vivado_init.tcl`' exist, create one:
     ```
     $touch Vivado_init.tcl
     ```
     (e) Open the file in a text editor like nano:
     ```
     $ nano Vivado_init.tcl
     ```
   - Step 2: Get the Absolute Path of $HDK_SHELL_DIR
     (a) Print the environment variable:
     ```
     $ echo $HDK_SHELL_DIR
     ```
   - Step 3: Add the Source Line in the Tcl File
     (a) Add the following line to the Tcl file:
     ```
     $ source $::env(HDK_SHELL_DIR)/hlx/hlx_setup.tcl
     ```

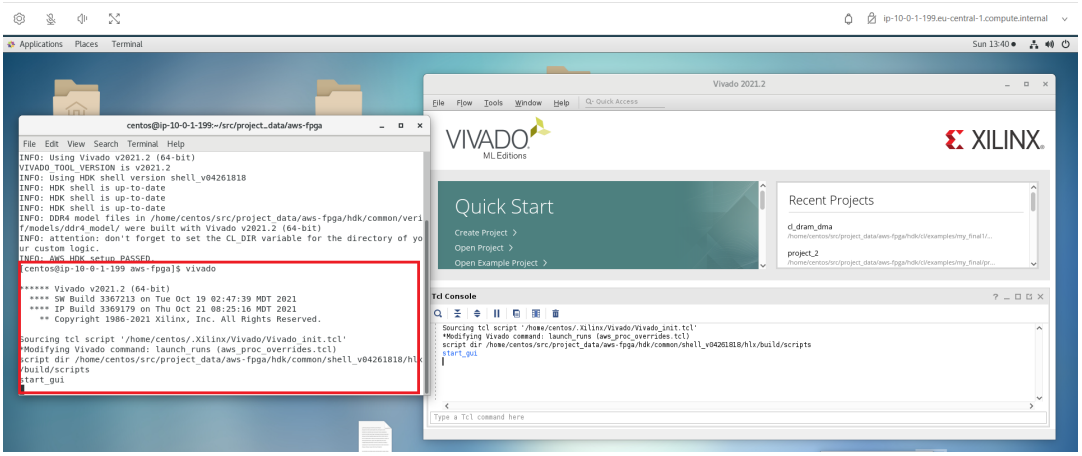


FIGURE 5.44: AWS configuration

(b) Save your changes and close the editor.

- Verify Configuration

  (a) Run Vivado to ensure the initialization file is sourced correctly and the environment is set up as expected.

```
1 $ vivado
```

If the installation is correct, you should see the following in your terminal: the software versions, the status of HDK shell updates, mentions of DDR4 model files built with Vivado, and a reminder to set an environment variable for custom logic. It also confirms that the AWS HDK setup has passed and includes the command used to start Vivado with specific scripts, likely for setting up or configuring FPGA development environments.

5.6 AFI generation

After make your chances and create the design follow the next step to create the AFI about the AWS

- Run both synthesis and implementation.
- When the implementation completed,the `.tar` file is located in `.runs/faas_1/build/checkpoints /to_aws/Developer_CL.tar`.
- To create an AFI, go to the website: https://github.com/aws/aws-fpga/tree/master/hdk in step 3 and follow the instructions.

Chapter 6

Design of the re-programmable Framework in AWS

The purpose of this thesis is to transform the hardware design created by Emmanouil Mylonakis which was mentioned in Section 4. This transformation aims to enable the tool that has been developed to run on the remote boards provided by Amazon. To achieve this, modifications were necessary in the existing VHDL code, which will be discussed in the following sections.

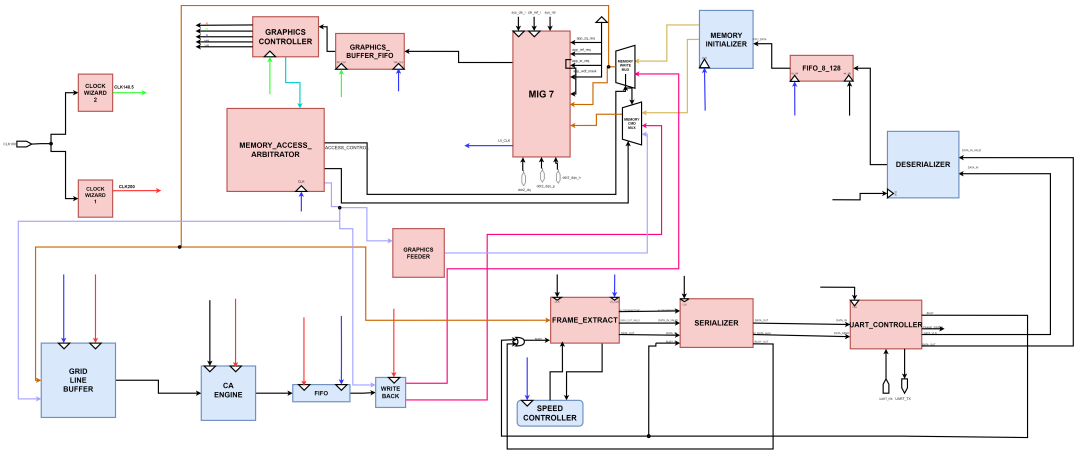


FIGURE 6.1: Transformation of Milonakis Hardware. Red:Removed and replaced,Blue:Changed.

Above is an image of Milonaki's hardware design. In blue, the parts kept from the initial idea and slightly modified to fit with the new architecture. The parts in red were removed and new ones took their places.

6.1 Connection to AWS Board

6.1.1 General Approach Of A Transaction

It is also important to note that the Shell (SH) and Custom Logic (CL) are the two parts of an FPGA environment that are relatively well defined on AWS F1 instances. The Shell is a static part of the environment, coming pre-supplied by AWS. It is the same for all F1 instances and contains the infrastructure that allows us to manage, interface, and scale well the FPGA resources. This part handles external memory interfaces, PCIe interfaces, and network connections. It is not modified by users, allowing AWS to maintain a stable environment while users focus on their specific algorithms in the Custom Logic.The Custom Logic ,on the other hand, is the user-defined portion of the FPGA. This is where users can develop and implement their own hardware designs.

To provide a clearer explanation of the data path as highlighted by the red lines in the diagram in figure 6.2.The data transfer begins from the Application-Specific Part of the PCIe and is then directed through Base Address Register 4 (BAR4). BARs are used in PCIe devices to specify the memory locations in system memory that the device will use for I/O operations. From there, the data travels to the PCIe to AXI Bridge, enters the board, and moves through the AXI Interconnect, a bus interface that handles data transfers between different components using the AXI protocol.As the data arrives at the AXI interconnect, it can be routed to different DDR (Double Data Rate) memory controllers such as M1, M2, M3 and M4, each responsible for handling data to and from its respective DDR memory module. This routing is based on the address where each piece of data will be stored.As shown in the diagram, DDR4-A, DDR4-B, DDR4-D, these memory modules are part of the user-defined FPGA area where the Custom Logic is implemented. Meanwhile, DDR4-C, this memory module is part of the Shell, the fixed infrastructure provided by AWS. To summarize:

- PCIe is the bridge that connects the host to the FPGA.
- DMA is the method for transferring data between the host and FPGA's memory efficiently.
- BAR4 is the control interface that the host uses to tell the FPGA what actions to take, like setting up and starting a DMA transfer.

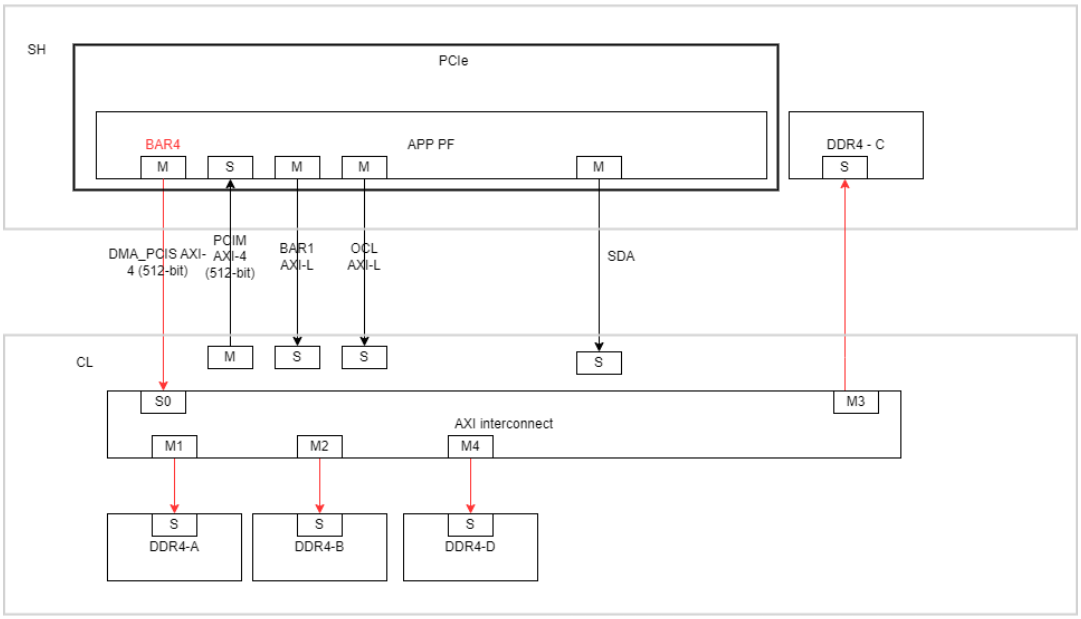


FIGURE 6.2: General Approach of transaction

6.1.2 More Detailed Approach Of A Transaction

To replace the UART protocol, including the UART controller in a hardware design, a module that enables wireless connections to the board is necessary, as cable connections are no longer used. In the Official GitHub repository of the AWS EC2 FPGA Hardware and Software Development, the CL_DRAM_DMA example provides how various interfaces and components are interconnected and function together in a typical FPGA setup. The following is a step-by-step explanation of both the hardware and software aspects of this example and how it interacts with the XDMA driver.

Hardware Side

- **CL_INT_SLV and CL_OCL_SLV Componets**: These appear to be slave interfaces for handling interrupts and possibly other control logic functions. The "INT" stands for interrupt, and "OCL" handles adjustments to settings or controls for parts of the system that don't deal directly with emergencies but are essential for the system's normal operations, like setting up a test for a device.
- **CL_DMA_PCIS_SLV Component**: The design integrates a cl_pcim_mstr.sv module for managing PCI Express Master functionalities, which involves data transfers between the host and the DDRS via DMA. The DMA paths are designed to efficiently handle large volume data transfers, crucial for applications requiring high throughput.

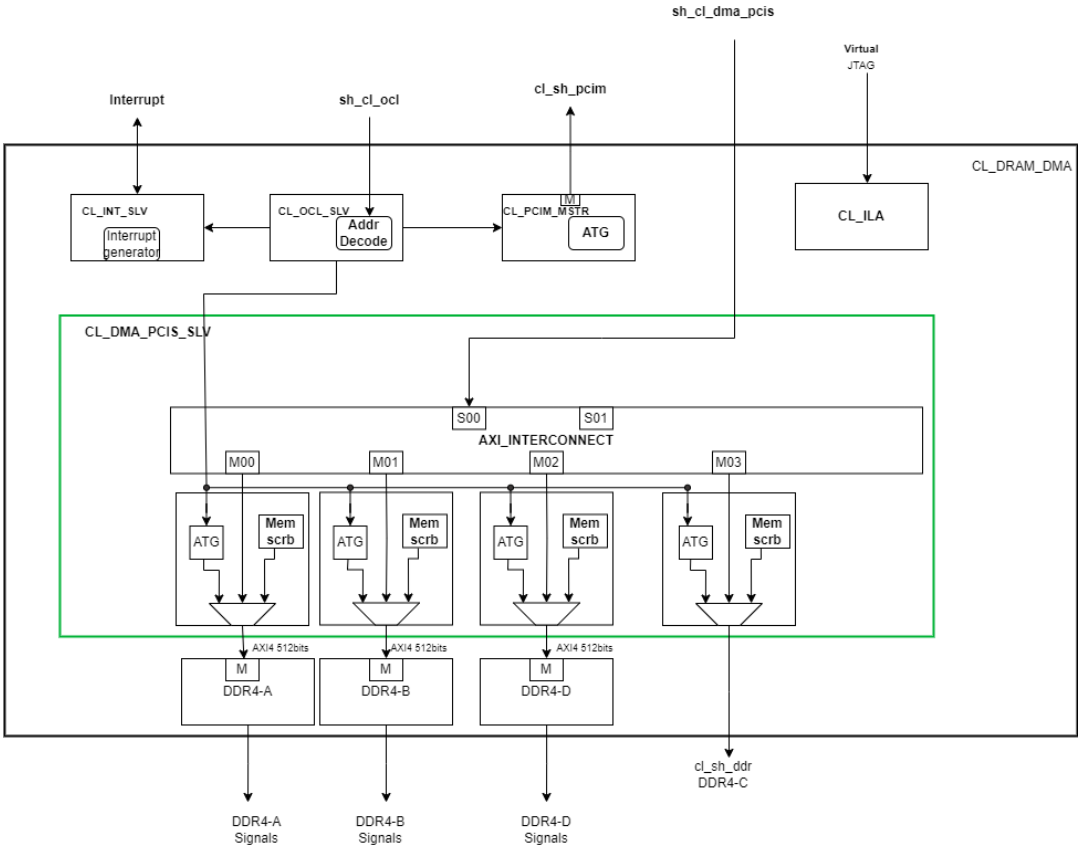
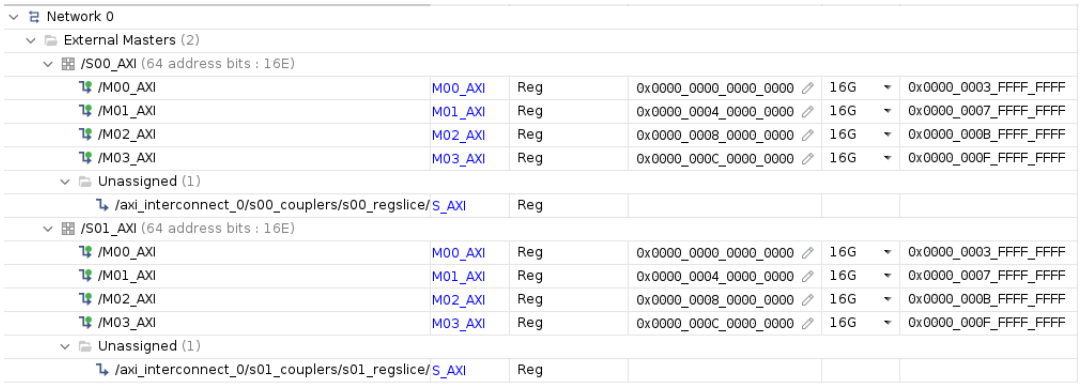


FIGURE 6.3: Block Diagram of CL_DRAM_DMA

- **CL_PCIM_MSTR Component**: Likely a PCI master module, controlling transactions over a PCI bus. This would typically handle requests and responses between the PCI interface and the system's internal components.
- **CL_ILA**: This subsystem likely supports testing and debugging operations. It could allow for monitoring system performance and diagnosing issues.
- **AXI_INTERCONNECT**:This component is part of the `CL_DMA_PCIS_SLV` and serves as a bus. It facilitates high-speed data transfer among multiple masters (M00 to M03) and slaves (S00, S01). Additionally, it connects to DDR memory to manage data transfers. To correctly address the DDR memory, an address map must be established as follows: DDR_A with a base address of 0x0_0000_0000 and a range of 16GB, DDR_B at 0x4_0000_0000 with a 16GB range, DDR_C at 0x8_0000_0000 also with a 16GB range, and DDR_D at 0xC_0000_0000 with a range of 16GB.he boundaries for each block do not overlap and are contiguous, which is typically necessary to ensure correct memory access.



Network 0					
External Masters (2)					
/S00_AXI (64 address bits : 16E)					
/M00_AXI	M00_AXI	Reg	0x0000_0000_0000_0000	16G	0x0000_0003_FFFF_FFFF
/M01_AXI	M01_AXI	Reg	0x0000_0004_0000_0000	16G	0x0000_0007_FFFF_FFFF
/M02_AXI	M02_AXI	Reg	0x0000_0008_0000_0000	16G	0x0000_000B_FFFF_FFFF
/M03_AXI	M03_AXI	Reg	0x0000_000C_0000_0000	16G	0x0000_000F_FFFF_FFFF
Unassigned (1)					
/axi_interconnect_0/s00_couplers/s00_regslice/	S_AXI	Reg			
/S01_AXI (64 address bits : 16E)					
/M00_AXI	M00_AXI	Reg	0x0000_0000_0000_0000	16G	0x0000_0003_FFFF_FFFF
/M01_AXI	M01_AXI	Reg	0x0000_0004_0000_0000	16G	0x0000_0007_FFFF_FFFF
/M02_AXI	M02_AXI	Reg	0x0000_0008_0000_0000	16G	0x0000_000B_FFFF_FFFF
/M03_AXI	M03_AXI	Reg	0x0000_000C_0000_0000	16G	0x0000_000F_FFFF_FFFF
Unassigned (1)					
/axi_interconnect_0/s01_couplers/s01_regslice/	S_AXI	Reg			

FIGURE 6.4: Vivado Address Editor

- **Role of ATG and Mem Scrb**: The combined role of the ATG and Mem Scrb is to ensure that data is accurately accessed and remains reliable as it is transferred between the memory and other components of the system.
- **DDR4-A, B, C, and D**:They are four memory channels on the AWS F1 FPGA that store and access data. The key difference is that DDR4-C is connected differently than the others. DDR4-A, B, and D are directly connected to the FPGA, making them faster to access. DDR4-C, on the other hand, is accessed through a special interface called the Shell, which can make it a bit slower.

Consider an example to understand how this works. Take for example data is arriving at the sh_cl_dma_pcis interface and has a destination address of 0x4_1000_0000. The address is then sent to the AXI_INTERCONNECT block, which controls the routing of real data to the correct memory controller. Based on the address, the interconnect sends data to the DDR_B-matching memory interface. This is visible on the diagram as the memory controller labeled M01.

From here, when the data hits the DDR_B memory controller, there could be more processing by this controller component in the form of the ATG and Mem scrb. That is the last error check and decode of addresses.After processing, the data is then written to the DDR_B memory. To summarize, the important modules that require attention for connecting to the AWS board are the CL_DMA_PCIS_SLV component and the DDRS. The CL_INT_SLV, CL_OCL_SLV, and CL_ILA components are not necessary for a simple DMA transaction. Similarly, the CL_PCIM_MSTR component is not required for this transaction.

XDMA

Before analyzing the software, it is important to understand what XDMA is. The XDMA interface facilitates data transfer between host systems and FPGAs on AWS EC2 instances.XDMA is a technology optimized to enable effective data transfers between the host system and FPGA, in this case representing the programmable hardware applied to execute an application. In this setup, FPGA is the hardware onto which certain functionality may be configured,and the XDMA software driver installed on the host manages the data transfer between the host's memory and the FPGA.

Here is how the master-slave relationship works in this setup:

- **Master**: The XDMA on the host side is the master, it starts all transactions and governs data transfers.
- **Slave**: The FPGA acts like a slave that either transmits or receives on command from the master.For that reason the module that use is `CL_DMA_PCIS_SLV` component.

The XDMA driver handles all the phases of data transfer: preparation of the DMA channel, queue handling,and interfacing the FPGA to assure proper communication of data.

- **Installation Steps**

  Open a terminal in Nice_DCV and follow these commands sequentially to remove any existing drivers, install the XDMA driver, and verify the installation.

```
# Stop the MPD process
sudo systemctl stop mpd

# Remove existing XRT and XOCL drivers
sudo yum remove -y xrt xrt-aws

# Navigate to the XDMA directory
cd ~/aws-fpga/sdk/linux_kernel_drivers/xdma

# Compile the XDMA driver
make

# Install the XDMA driver
sudo make install

# Verify that the XDMA driver is installed
```

```
17 lsmod | grep xdma
```

Software Side

The software for the cl_dram_dma example can be found within the AWS FPGA repository, specifically under the hdk/cl/examples/cl_dram_dma/software directory. This directory contains the host software, which is written in C/C++, and is responsible for interacting with the FPGA hardware.

The host software serves as a "master" in this setup, controlling the flow of data between the host's DRAM and the FPGA via DMA (Direct Memory Access). The primary role of this master software is to initiate and manage the DMA transfers, which involves several steps: initializing the FPGA and setting up the environment, allocating memory buffers on the host side, configuring the DMA engine on the FPGA, and then triggering the actual data transfers. In the repository hdk/cl/examples/cl_dram_dma/software in the test_dram_dma.c may show attention in dma_example function that is:

```
1  int dma_example(int slot_id, size_t buffer_size) {
2      int write_fd, read_fd, dimm, rc;
3
4      write_fd = -1;
5      read_fd = -1;
6
7      uint8_t *write_buffer = malloc(buffer_size);
8      uint8_t *read_buffer = malloc(buffer_size);
9      if (write_buffer == NULL || read_buffer == NULL) {
10         rc = -ENOMEM;
11         goto out;
12     }
13
14     read_fd = fpga_dma_open_queue(FPGA_DMA_XDMA, slot_id, /*
       channel*/ 0, /*is_read*/ true);
15     fail_on((rc = (read_fd < 0) ? -1 : 0), out, "unable to open
       read dma queue");
16
17     write_fd = fpga_dma_open_queue(FPGA_DMA_XDMA, slot_id, /*
       channel*/ 0, /*is_read*/ false);
18     fail_on((rc = (write_fd < 0) ? -1 : 0), out, "unable to open
        write dma queue");
19
20     rc = fill_buffer_urandom(write_buffer, buffer_size);
21     fail_on(rc, out, "unable to initialize buffer");
22
```

```
23      for (dimm = 0; dimm < 4; dimm++) {
24          rc = fpga_dma_burst_write(write_fd, write_buffer,
    buffer_size, dimm * MEM_16G);
25          fail_on(rc, out, "DMA write failed on DIMM: %d", dimm);
26      }
27
28      bool passed = true;
29      for (dimm = 0; dimm < 4; dimm++) {
30          rc = fpga_dma_burst_read(read_fd, read_buffer,
    buffer_size, dimm * MEM_16G);
31          fail_on(rc, out, "DMA read failed on DIMM: %d", dimm);
32
33          uint64_t differ = buffer_compare(read_buffer,
    write_buffer, buffer_size);
34          if (differ != 0) {
35              log_error("DIMM %d failed with %lu bytes which
    differ", dimm, differ);
36              passed = false;
37          } else {
38              log_info("DIMM %d passed!", dimm);
39          }
40      }
41      rc = (passed) ? 0 : 1;
42
43  out:
44      if (write_buffer != NULL) {
45          free(write_buffer);
46      }
47      if (read_buffer != NULL) {
48          free(read_buffer);
49      }
50      if (write_fd >= 0) {
51          close(write_fd);
52      }
53      if (read_fd >= 0) {
54          close(read_fd);
55      }
56      return (rc != 0 ? 1 : 0);
57  }
```

LISTING 6.1: DMA Example Function

In the Function:

- Buffer Allocation: "**write_buffer**" for holding data to be written to the DDR and "**read_buffer**" for reading data back from the DDR to verify

the write operation.

- DMA Queue Handling: Open DMA queues for both reading and writing. This is done using the "**fpga_dma_open_queue**"function which sets up the DMA channel for either reading or writing.
- Buffer Initialization:The write buffer is filled with random data using the "fill_buffer_urandom" function.
- DMA Write Operations:Data is written to each DDR using the "`fpga_dma_burst_write`" function, which is used to perform burst write operations to the specified addresses in the DDR.
- DMA Read and Verification:Data is read back from each DDR DIMM using "**fpga_dma_burst_read**" to ensure that the write operations were successful. The data read back is compared to the data written to verify the integrity of the data transfer.
- The function includes robust error handling to manage and report any issues that occur during the DMA operations.

This is the example provided by Amazon on how to connect and transfer data, which we used for our connection. The hardware and software provided were adapted into our own implementation. The CL_DMA_PCIS_SLV has replaced the UART controller. Now, there is essentially a master that operates on the software side, and the slave CL_DMA_PCIS_SLV that receives the data within the hardware. This setup allows for control over where to store the data, when to read it, and from where.All that concept with master-slave may be preserved in each data transfer in this implementation.

Setting up the example cl_dram_dma in Vivado

To put this example in Vivado in NICE_DCV and start the implementation,

- To set the `AWS_FPGA_REPO_DIR` environment variable, run the following command:

```
1 export AWS_FPGA_REPO_DIR=/home/centos/src/project_data/aws-
    fpga
```

- Navigate to the AWS FPGA Repository Directory Change to the directory where the AWS FPGA repository is located:

```
1 cd $AWS_FPGA_REPO_DIR
```

- Run the Setup Script Run the setup script to configure the environment:

```
source hdk_setup.sh
```

- Navigate to the `hdk` Directory:

```
cd hdk
```

- Navigate to the `cl` Directory

```
cd cl
```

- Explore the Examples Directory

```
cd examples
```

- Create a New Project Directory for example my_final2

```
mkdir <your_directory_name>
```

- Start Vivado

  After creating your project directory and navigating into it, you can start Vivado by running:

```
cd my_final2
vivado
```

- After opening the Vivado click on "Create Project".

FIGURE 6.5: Create Project in AWS

- In the "Project Name" field, enter the name of your project, and specify the directory where the project files will be stored. Use the folder you created earlier (for example: my_final2) and you have filled out the necessary details, click on the "Next >" button to continue with the project setup process.
- After finish the project setup,In the `Tcl Console` specific environment variables are being set, such as:

```
set ::env(CLOCK_A_RECIPE) "2"
set ::env(CLOCK_B_RECIPE) "0"
set ::env(CLOCK_C_RECIPE) "0"
set ::env(device_id) "0xF001"
set ::env(vendor_id) "0x1D0F"
set ::env(subsystem_id) "0x1D51"
set ::env(subsystem_vendor_id) "0xFEDC"
```

  - **Clock Recipes A,B,C:** These define how the clocks within the FPGA design are configured. Amazon provides some clocks for use, and the frequency can be found in the GitHub repository.
  - **Device ID, Vendor ID, Subsystem IDs:** These IDs are used during the synthesis and implementation process to ensure that the FPGA AFI is generated for the correct hardware. AWS FPGAs in the cloud are identified by these IDs, ensuring that the design is compatible with the target device.

- In the `Vivado TCL Console`, type in the following command to create the `cl_dram_dma` example.

```
aws::make_rtl -examples cl_dram_dma
```

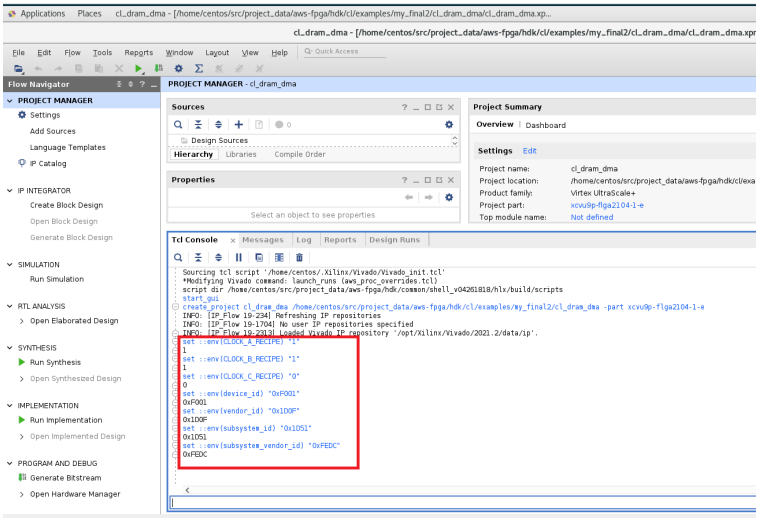


FIGURE 6.6: project setup

After running the aws::make_rtl -examples cl_dram_dma command in the Vivado Tcl Console, the example project for cl_dram_dma is created and the project is targeting the AWS F1 Platform, specifically the xcvu9p-flgb2104-2-i FPGA part, which is a Xilinx Virtex UltraScale+ FPGA.

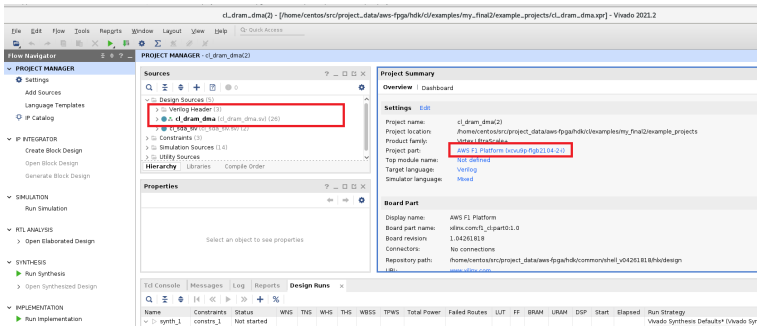


FIGURE 6.7: project configuration

In the project directory, there may be an "**example_projects**"folder that seems ready for changes.

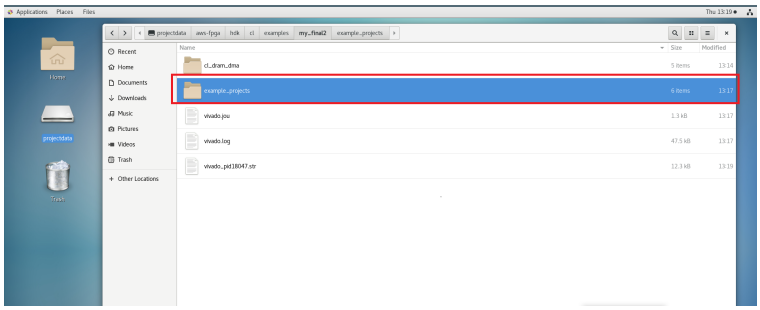


FIGURE 6.8: project configuration

6.1.3 CL_DMA_PCIS_SLV Component

As mentioned in the above subsection, this example is the best way to transfer data because it utilizes the XDMA protocol, which is used for large data transfers from the system memory to the FPGA without involving the CPU. Amazon includes this method as a recommended example and encourages its use in a workshop for FPGA implementation that it has organized in the past.

For this reason, it would be useful to focus on and understand the data transfer component of this example. The main part that facilitates this is called cl_dma_pcis_slv, which seems to involve the management and communication with various DDR (Double Data Rate) memories. The primary module used is AXI buses (axi_bus_t) which allow the connection between the FPGA and the DDR memory units. Regulatory functions such as AXI register slices and AXI interconnects are used to optimize and manage the flow of data and commands through the AXI buses. Additional functions include control and diagnostic operations, for managing and configuring the

DDR memories through different buses (cfg_bus_t), as well as implementing functions like ATG (Address Translation Gateway) and scrubbing to verify and maintain the integrity of the data in the DDR memories. In conclusion, the system aims to create an efficient and reliable memory management system for applications that require high performance and fast communication, which will aptly replace the UART controller of the original system.

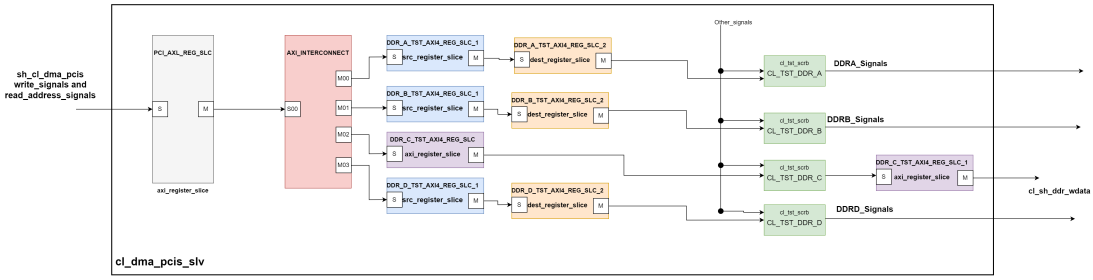


FIGURE 6.9: CL_DMA_PCIS_SLV Block diagram

6.2 DDR controllers

The sh_ddr.sv file, provided by Amazon, contains the logic for the DDR controllers.The example uses DDR controllers as part of cl_dram_dma integral controllers. These are designed to handle the interface between the FPGA and the attached DDR memory. The controllers handle memory transactions(read and write operations)to the DDR memory, providing high-bandwidth storage necessary in many different data processing tasks.Furthermore, many FPGA-based designs on the Amazon platforms, including the cl_dram_dma example, will often feature having multiple DDR controllers enabled to increase as much as possible the available memory storage. Each DDR controller will deal with a part of the memory space, which is mapped through different address ranges on the AXI4 bus. For example, DDR_A may deal with addresses starting from 0x0_0000_0000, DDR_B from 0x4_0000_0000, and so on, each controlling 16 GB of memory space.In case that a DDR controller is not in use,it can be disabled by modifying design files. For example, updating the `cl_dram_dma_ defines.vh` file to disable specific DDR controllers and ensuring that the address space associated with disabled controllers is not accessed during tests is one such modification. Although this

implementation does not currently disable any of the DDR controllers, the capability to do so is available.

The code inside the sh_ddr.sv file is encrypted.Encryption is used to secure data and ensure that sensitive information is not exposed. In cloud environments, multiple users can share the same physical hardware, making it critical to ensure that one user's data remains isolated and secure from others. For security reasons, Amazon provides ready-to-use DDR controllers that are more secure and simpler to use.

6.3 Re-programmable Framework in AWS

In this implementation, the system initialization does not occur as in the original. Here, all data will initially be written to memory and then distributed to the appropriate components for the execution of computations. For this reason, several changes have been made within the code and will be explained step by step below.

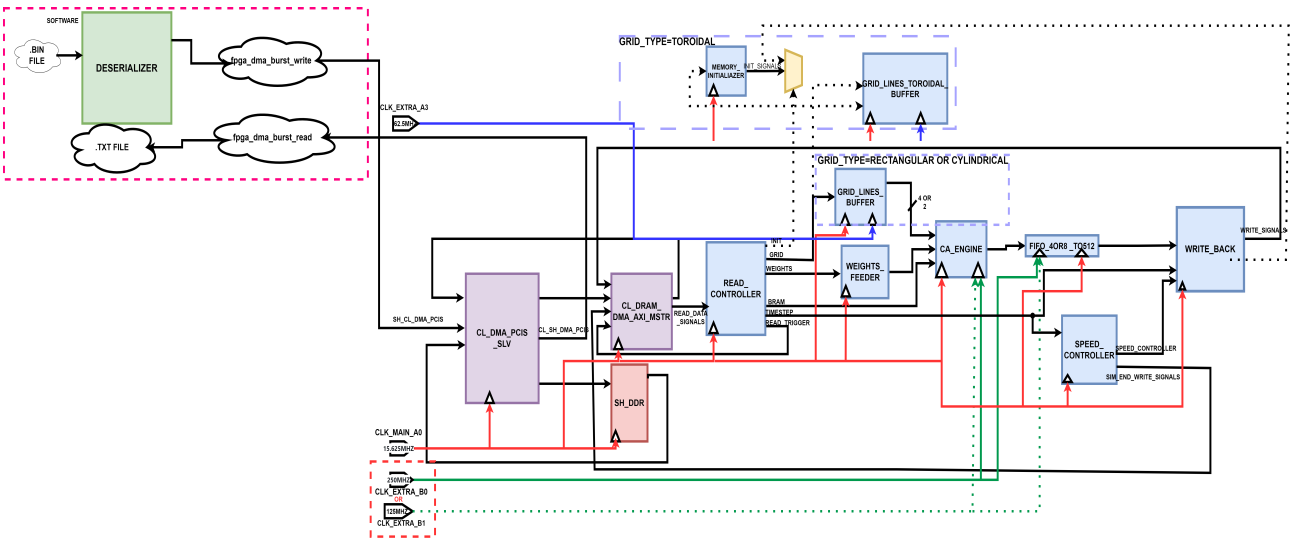


FIGURE 6.10: Re-programmable Framework Block diagram

6.3.1 Grid Representation in Memory

In updated setup, we have transitioned from using 128 MB DDR2 memory ,in the original system, to a 64 GiB DDR4 system provided by AWS, consisting of four 16 GiB modules. This upgrade includes a change in burst size from 128 bits to 512 bits, which AWS supports. This increase in burst size significantly impacts how we handle data, particularly in the way memory

bursts accommodate cellular automaton cells.The configuration changes as follows:

- Burst Size: In prior system, 128-bit bursts were employed, with each burst comprising eight 16-bit words. Transitioning to a 512-bit burst size, and assuming that the size per cell (e.g., 4 or 8 bits) remains constant, this change implies that each 512-bit burst can accommodate four times the number of cells as the 128-bit bursts.In the memory system of AWS, uses eight 64-bit words.
- Cells per Burst (**cb**): This value now recalculates based on the new burst size and the cell size. The formula remains:

$$cb = \frac{b}{c} = \frac{512 \text{ bits}}{4 \text{ bits/cell}} = 128 \text{ cells/burst} \quad (\text{if } c = 4 \text{ bits})$$

  or

$$cb = \frac{b}{c} = \frac{512 \text{ bits}}{8 \text{ bits/cell}} = 64 \text{ cells/burst} \quad (\text{if } c = 8 \text{ bits})$$

  but but with b now equal to 512 bits. This alteration allows for more cells per burst, enhancing the efficiency of data transfers and reducing the number of memory accesses required per frame.
- Memory Bursts per Grid Line (**bl**):The number of memory bursts required to represent a single line of the cellular automaton grid (1920 cells per line) will decrease. This is calculated by

$$b_l = \frac{x \times c}{b} = \frac{1920 \times 4 \text{ bits/cell}}{512 \text{ bits}} = 15 \text{ bursts/line} \quad (\text{if } c = 4 \text{ bits})$$

  or

$$b_l = \frac{x \times c}{b} = \frac{1920 \times 8 \text{ bits/cell}}{512 \text{ bits}} = 30 \text{ bursts/line} \quad (\text{if } c = 8 \text{ bits})$$

  with x being the total number of cells per line. The increase in b means fewer bursts are necessary per line, which can improve the performance of memory accesses and the overall processing of cellular automaton frames.

The 1920 × 1080 grid is represented in memory as shown in the figure 6.11. Each line consists of 1920 cells organized into memory bursts, with each burst containing cb cells. The number of cells per burst is variable, which means

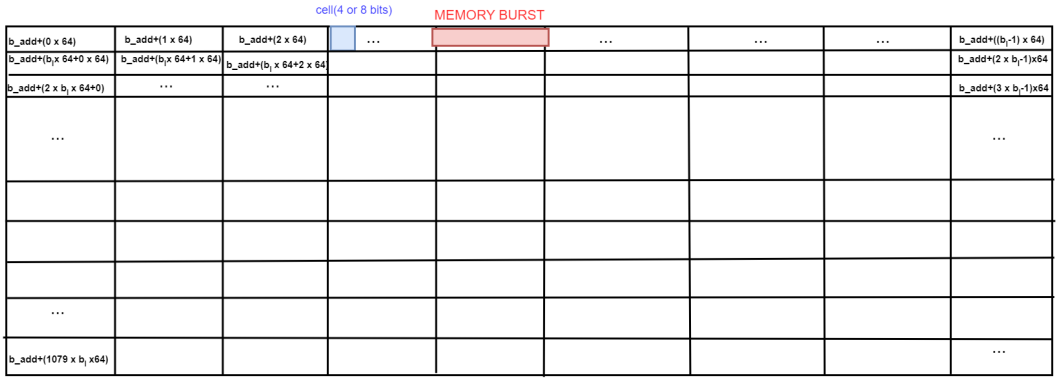


FIGURE 6.11: Grid representation in memory and burst addressing

the number of memory bursts per grid line (bl) is also variable that for the new architecture is 15 or 30 as calculated before. The memory layout begins with a base address of 0x00004500, and the entire grid is aligned according to the DDR4 memory system's 64-byte boundary.This means each burst must start on an address that is a multiple of 64 bytes.

The memory is designed to hold two consecutive frames of the grid for double buffering purposes. Both memory segments use the same addressing pattern as depicted in the figure, distinguished only by the most significant bit of the address.

6.3.2 Memory Initialization

The initialization of the memory has changed significantly from the original implementation, which was based on the CAD tool developed by the student named Emmanouil Mylonakis. To transfer and store data in the memory, an external Deserializer was developed in software and implemented in C. This Deserializer takes the array created using the Protobuf protocol, as mentioned in the 4.2.2 subsection, and decodes the data so that it is ready to be written to memory. The software implementation of the Deserializer is quite similar to the original hardware component in terms of functionality, but it is written in C and is no longer part of the hardware design.

Now, the memory initialization is no longer done through a hardware component like the memory initializer used in the previous implementations. Instead, it is handled by the functions discussed in the earlier subsection 6.1.2(Software Side), and the Deserializer is no longer a hardware component but a software program. As shown in the figure above, the Deserializer is no

longer present in the block diagram, and the memory initialization process has been removed from the VHDL code of the memory initializer, retaining only the part that supplies the grid_line_buffer.

Additionally, a new feature of the implementation is that during initialization, it must be predetermined which memory addresses will be used to store each different element that will later be useful for computations. As shown in the code below, we have pre-decided the memory addresses. By knowing the starting address of each element we want to store, we ensure that we can read the correct addresses later to retrieve the information.

```
    }

    // Define addresses for various data
    uint64_t time_step_addr    = 0x0000000000000000;
    uint64_t others_addr       = 0x0000000000000040;
    uint64_t addr_sel_addr     = 0x0000000000000080;
    uint64_t weights_addr      = 0x0000000000000100;
    uint64_t bram_value_addr   = 0x0000000000000500;
    uint64_t grid_value_addr   = 0x0000000000004500;
    uint64_t grid_value_addr1  = 0x0000000000204500;
    uint64_t generation_addr   = 0x0000000000301800;

// Define completion signal and address
    uint8_t completion_signal  = 1;           // The single byte to write
    uint64_t completion_addr   = 0x00000000001fe940;  // Address to write the completion signal
```

FIGURE 6.12: pre-decided the memory addresses

Deserialization Mechanism in C

The code contains an FSM that has states such as TAG, LENGTH, `VALUE _REPEATED`, and VALUE_OPTIONAL, which determine the action to take based on the current state and incoming data. The FSM starts in the TAG state, where it identifies the type of data chunk(`time_step,others,adde _sel, weights,bram_values or grid`) ,it's dealing with by reading a tag byte. It then moves to the LENGTH state to determine how much data follows this tag, often using continuation bits to support variable-length data encoding. Depending on the tag, the FSM might move to VALUE_REPEATED to process repeated value bytes or to VALUE_OPTIONAL for one value processing. For instance, a tag indicating a time step might imply that the following bytes need to be written to a specific address.The data can be stored temporarily in a buffer (write_buffer) until it's ready to be written out via DMA.

Writing Data to FPGA using DMA in C

The function fpga_dma_burst_write() is used to send data from a buffer to specific locations on an FPGA, which are determined by predefined addresses like time_step_addr, weights_addr, bram_value_addr, etc 6.12.There is also a completion signal at address 0x001FE940 that indicates the write process has been completed and is going to be used for system initialization.

After all this, the data is ready and being sent to be stored in specific memory addresses in the Fpga and more specific in the DDRs. Once the memory writing process is completed, the next step is to analyze how the data will be distributed among the components for computations.

6.3.3 Read$_{C}$ontroller*Module*

This new element manages memory reading, works at 15.625MHZ(UI_CLK). The controller begins by initializing and waiting for an initial signal(the completion signal) before starting the read process.Then,It reads various types of data sequentially from memory, including timestep, weights, BRAM data, other data, address selection, and grid data. For each data type, the controller sets the appropriate memory address, waits for valid data, reads it, and stores it in corresponding output signals.The component contains an FSM,to control the process of reading different types of data from memory. It begins in the RESET state, where all internal signals are initialized . After the reset, the FSM transitions to the INIT state,where it sets up the initial address for reading and waits for a specific condition (when init_read is '0' and read_data(7 downto 0) equals x"01" from the memory address 0x001FE940) to be met. Upon satisfying this condition, the FSM moves to the READ_TIMESTEP state. In the READ_TIMESTEP state, the FSM reads timestep data from memory when the rvalid signal is asserted, indicating that the data on the bus is valid. Once the timestep data is successfully read, the FSM transitions to the`READ _WEIGHT` state. Here, the FSM enters a loop to read weight data from memory, repeating the process until 14 weight blocks are retrieved. After all the weights are read, the FSM proceeds to the READ_BRAM state.In the READ_BRAM state, the FSM reads data intended for Block RAM (BRAM) and writes it to BRAM as soon as the rvalid signal indicates valid data is present. Once all necessary BRAM data has been read, the FSM transitions to the READ_OTHER state, where it reads another segment of data from memory. After completing this, the FSM moves to the READ_ADDRSEL state to read the address selection data.Following this, the FSM transitions

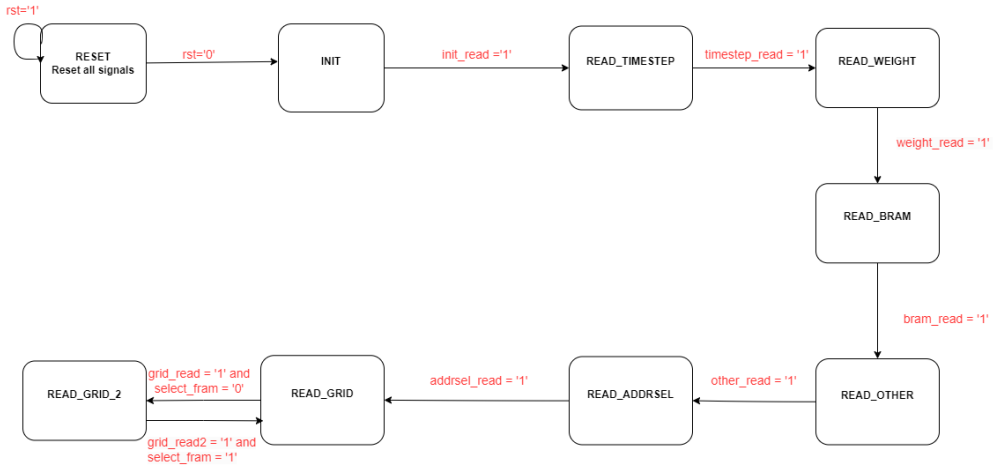


FIGURE 6.13: FSM IN READ CONTROLLER

to the READ_GRID state, the system reads grid data from a specified memory address, continuing this process based on a specified count , calculated as 16200 * CELL_SIZE / 4.If in the end of this process the select_fram input became low, indicating a requirement to switch frames and the state machine transitions to the READ_GRID_2 state to read data from the other frame.The READ_GRID_2 state handles reading operations from a different memory base, denoted by grid_address_2. The operations in this state are similar to those in READ_GRID. if the select_fram input is high, the FSM transitions go back to the READ_GRID state.The switch between READ_GRID and READ_GRID_2 is a double buffering method for reading and writing from different frames, then switching between them.

6.3.4 Cl_Dram_Dma_Axi_Mstr Module

This module, developed by AWS, is connected to memory via an AXI_INTERCONNECT. It functions as a master on an AXI bus, primarily designed to manage memory access for both read and write operations. The module uses a finite state machine (FSM) to control the sequence of AXI transactions, handling specific states that include issuing write and read commands, managing data transfers, and receiving acknowledgments from the bus.

The module has been modified to facilitate reading and writing of the calculations ,that need for CA, in memory. It now includes new input signals named read_trigger and write_trigger, which allow memory operations to be initiated by external signals rather than through the usual control register interface. This modification enables operations to start automatically based on specific triggers from the calculation modules, which will be discussed in the next sections.

Additionally, new inputs such as TRIGGER_READ_ADDR_HI, TRIGGER_READ_ADDR_LO, TRIGGER_WRITE_ADDR, and TRIGGER_WRITE_DATA specify the exact memory addresses for read and write operations. The module also features a new output register, output_rdata, which captures data directly from the AXI bus and sends it to the Synchronizer.

A new state machine (trigger_sm_states) has been added to handle the states of triggered read and write operations separately from the main AXI transaction state machine. Two new FSMs were created to manage the lifecycle of read or write operations initiated by external triggers. In the write operation FSM, starting from TRIGGER_IDLE, the FSM waits for a write_trigger. Then, it transitions to TRIGGER_WR. In this state, if the AXI bus is ready (awready is high), the FSM moves to TRIGGER_WR_DATA, sending the data to the bus. Following the successful acceptance of data (wready is high), it progresses to TRIGGER_WR_RESP, where it waits for a completion confirmation (bvalid is high) from the bus before returning to TRIGGER_IDLE.

For the read operation, the FSM also starts in TRIGGER_IDLE, waiting for a read_trigger. When triggered, it shifts to TRIGGER_RD, moving forward to TRIGGER_RD_DATA if the AXI bus signals arready is high. It then waits for the read data (rvalid is high), stores the received data in output_rdata, and returns to the idle state upon completion.Since there are separate channels for reading and writing, there's no need for memory sharing as in the original, which reduces a lot of complexity. Essentially, when writing and

reading to different memory addresses simultaneously, there's no need for a control signal to manage memory sharing. Additionally, writing and reading in completely different memory frames, as explained in the original, simplifies multiplexing and speeds up the process.

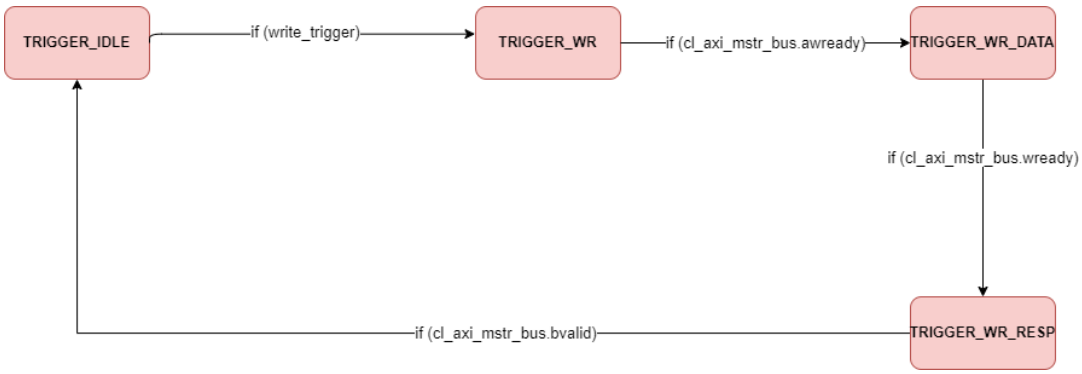


FIGURE 6.14: WRITE FSM

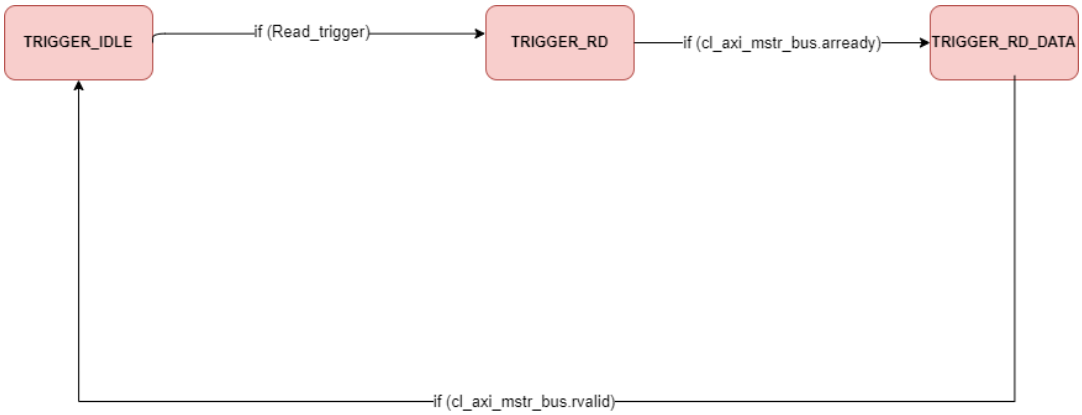


FIGURE 6.15: READ FSM

6.3.5 Weights_Feeder Module

The `WEIGHTS_FEEDER` component is between a `Read_Controller` and `CA_ENGINE`, suggesting that it is responsible for feeding weights to the `CA_ENGINE` in a controlled manner.The `WEIGHTS_FEEDER` component is designed to manage the transfer of weight data from a larger 512-bit input to the `CA_ENGINE` in smaller, more manageable chunks (`weights_row`). The component carefully handles partial data processing, ensuring that only valid and complete rows of data are passed on to the next stage.The component stores this data in a 1024-bit register and processes it in smaller chunks. It uses internal signals to track how much data has been processed and when to output valid rows. During operation, it captures incoming data, processes it if enough is available, and outputs it in rows while managing the validity of the output.

6.3.6 Grid_Line_Buffer Module

Since, as mentioned, the burst size has changed from 128 to 512, some parts of the code must be modified to support the new input, ensuring there is no loss of information. For this reason, the following adjustments have been made. First of all, as mentioned, the BURST SIZE changed from 128 to 512, which necessitates reducing the number of bursts per line from 60 to 15 when the CELL size is 4, and from 120 to 30 when the CELL size is 8. Additionally, the clock frequencies were altered: the ui clock, which is the write clock for the block memory, is now set to 15.625 MHz, and the clock responsible for reading from the block memory is now 62.5 MHz. Due to the change in burst size, the block memory diagram itself will now accept an input of 512 bits and produce a minimum output of 16 bits in both cases(**CELL SIZE= 4 OR CELL SIZE=8**). This contrasts with the original design, which had an input of 128 bits and outputs of 4 or 8 bits. This new implementation allows us to have four data outs or two for cell sizes of 4 and 8 respectively, instead of just one data out as in the original Grid Line Buffer.This is because 16 bits are now read at a time from the block memory, which can be calculated as **16/4 (cell size=4) and 16/8 (cell size=8)**. **These changes made in the grid lines buffer and the grid lines toroidal buffer but does not significantly alter the remaining logic**, the main difference is that more LINE_DATA tables will now be added to store data for the new neighborhoods, which will now be either 4 or 2, with obviously 4 or 2 DATA_OUTs accordingly. In the new architecture, for cell size 4 there will be 4 outputs at a time, meaning the first, second, third, and fourth will appear first, followed by the fifth, sixth, seventh, and eighth, and so on, as opposed to one in the original.Adding three additional output ports—DATA_OUT1, DATA_OUT2, and DATA_OUT3—alongside the existing DATA_OUT port. This means that the new version can handle four or two separate output streams in total and make the process faster.

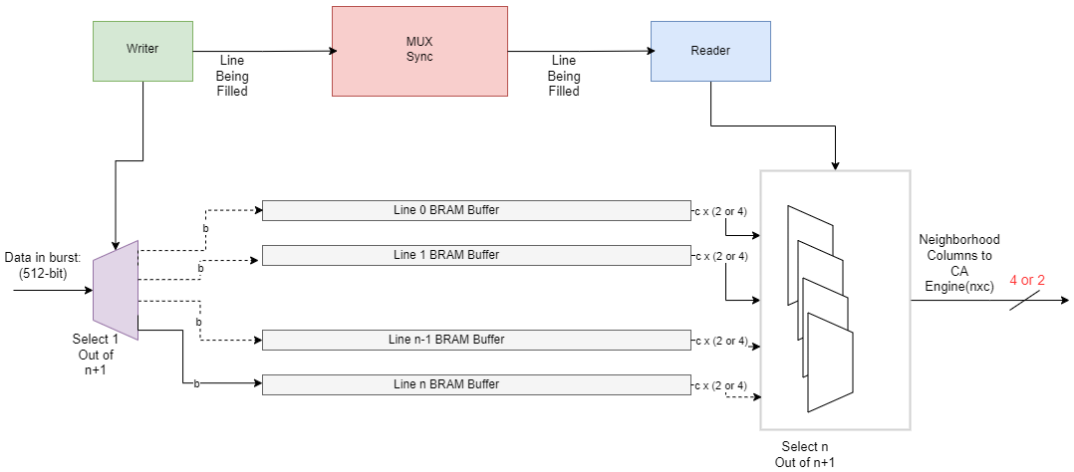


FIGURE 6.16: GRID_LINE_BUFFER_UPDATED

6.3.7 Ca_Engine Module

Ca_Engine WITH CELL SIZE=4

Another module that was affected by the change in burst size, as well as the changes mentioned in the grid lines buffer module, is the CA ENGINE. **Since it is connected to the Grid Lines Buffer and its output serves as the input for the CA ENGINE, the latter's inputs must also change to accommodate four, based on the cell size. The main change, however, concerns the clock frequency:** because the Grid Lines Buffer operates at 62.5 MHz and provides four inputs simultaneously, the CA ENGINE must operate at 250 MHz in order to correctly store the inputs and prevent data loss 6.18. This ensures that information is not lost and the CA ENGINE now operates at **250** MHz. Additionally, a significant change was made in addressing and the data size in the BRAMs, since 512 bits are now read from memory, necessitating a change in the input data size to 512 bits. This brought changes in the memory addressing in the BRAMs and in the adder tree, which is analyzed in Section 4 on how it operates. Now, to obtain the correct result with the new data, the sum from the adder tree must be divided by 64 and the mod 64 of the sum calculated. This is done because the data now comes out in much larger blocks, and while in the original implementation, **if the sum from the adder tree was, for example, 3, the read address from the BRAMs would be 3, in this implementation we do not want to look at the third because the output is now 512 and we want to look at address 0. This will result in a 512-bit output and we must select the appropriate byte, and for this reason, we use mod 64.** The division by 64 is performed because the values of the BRAM are stored per 8 bits(1 byte), and dividing 512 by 8 is 64. Further down, there

will be a detailed explanation in fig 6.17. Also, minor changes are made to ensure that this is synchronized and correctly adjusted, however, the logic remains the same as in the original.

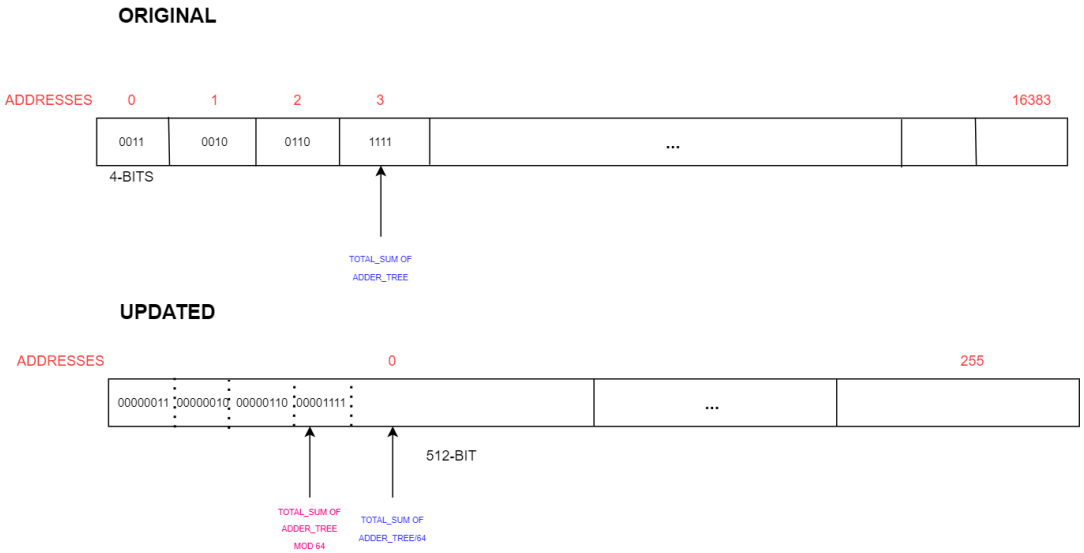


FIGURE 6.17: Updated BRAMs

Ca_Engine WITH CELL SIZE=8

The main difference between the new CA MODULE for CELL SIZE=4 and the one for CELL SIZE=8 is that in this case, the **Grid Lines Buffer only outputs 2 signals at a time, thus it will have only 2 inputs, and the clock frequency will be 125 MHz.** This is because, as explained above, the Grid Lines Buffer outputs data at a frequency of 62.5 MHz, so to correctly write the data to the neighborhood, given that it has only 2 inputs, the clock must be at 125 MHz and not 250 MHz as in the previous version. The rest remains almost identical to the previous version.

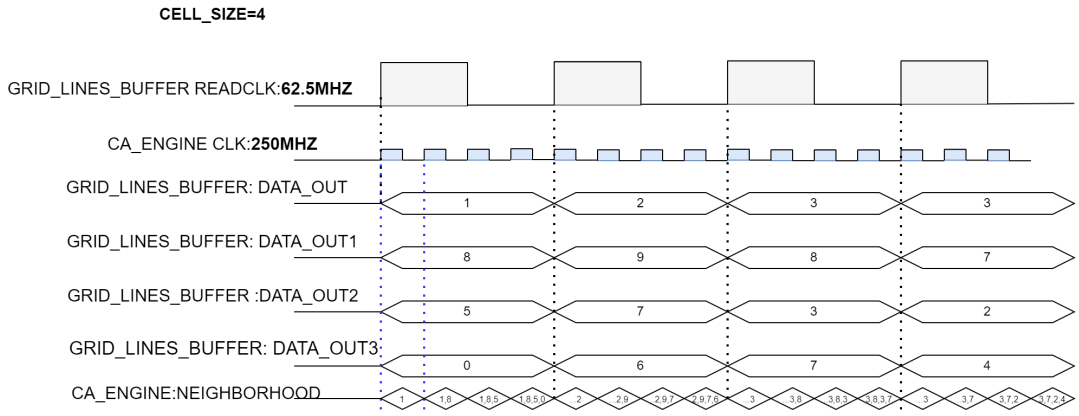


FIGURE 6.18: Time multiplexing between GLBuffer and CA Engine

6.3.8 Write_Back Module

This module appears to be used for managing the writing back of data to memory after processing cellular automata. It likely handles synchronization between the computational core and the memory system, ensuring that data is written correctly and at the right address. The module has not changed much from the original. First of all, the APP_CMD signal was removed because the DDR4 on AWS supports separate channels for reading and writing operations, and a command signal is not needed for binding the memory for writing. As mentioned before, the burst size and the address size have changed, and all the necessary adjustments have been made to support the new architecture, but the logic remains almost the same as the original analyzed previously in Chapter 4.

6.3.9 Speed_Controller Module

The updated version of the SPEED_CONTROLLER module introduces significant functional enhancements over its predecessor, notably in its ability to interact with memory. It can now store the number of generations at the address X"0000000000301800," a feature specifically used to confirm the completion of generation computations, which was absent in the original version. The use of SIM_ENDED allows for a controlled and predictable end to a simulation, ensuring that all dependent or subsequent operations are based on accurate and complete computation results. When SIM_ENDED is set to 1, all computations stop and no further reading or writing to the memory occurs. The reason for this freezing is to extract the correct generation data from

the software at that point, and the computations do not restart. this ability has been removed.

6.3.10 Read Software Side

After the writing, in a software program, the code begins by allocating a buffer, timestep_read _back_buffer, to hold 64 bytes. It then uses fpga_dma _burst_read to read 64 bytes of data from the address time_step_addr into this buffer. After reading, the contents of the buffer are stored in the result variable.Next, the code enters a loop that repeatedly reads from another memory address, generation_addr= X"0000000000301800, into a buffer named read_back_buffer to poll for data changes. The loop continues reading until the value from this address (stored in result_back) matches the value of the variable result. To prevent system overload from continuous polling, the code pauses for 100 milliseconds between each iteration using usleep(10000000).

Once result_back equals result, the code proceeds to read grid data. It works with two base addresses, grid_value_addr and grid_value_addr1. Depending on whether the result variable is an even or odd number, the code reads data from one of these addresses, incrementing the address by 64 bytes for each iteration, suggesting that it reads the data in 64-byte chunks. The loop runs for a number of iterations defined by grid_length_in_burst, reading 64 bytes per iteration from the current address into the read_back_buffer.

If a DMA read operation fails during this process, the code prints an error message and exits the loop. After each successful read, the contents of the buffer are printed, and the data is written to a .txt file.Then manually transfer the .txt file to the CAD tool to create the final image.

Chapter 7

System Verification, Examples of Use and Results

In the previous chapters, an extensive analysis of cellular automata as mathematical models was conducted, and the construction of a CA Accelerator for computing and visualizing these mathematical models was explored. My task was to remove the Nexys 4 DDR board and enable remote execution. In this chapter, after discussing the transformations made to achieve this, the results will be presented and system validation will be performed.

7.1 System Verification

The first verification step is to ensure that the connection with the AWS board is established and that it can read and write data. After loading the AFI, as analyzed in Chapter 5, and implementing the C functions for reading and writing, as discussed in Chapter 6, it is essential to confirm that the software application can successfully connect to the board.The figure shows a process of writing and reading from memory, likely for an AWS F1 FPGA using the cl_dram_dma interface. The steps indicate memory addresses being accessed and written to, which proves the capability to interact with the FPGA's memory via direct memory access (DMA). In the first instance, we see data being written to the memory addresses time_step_addr (0x0) and others_addr (0x40). The buffer contents for time_step_addr show data like 3C 00 00 00, which likely represents meaningful values specific to the application. The other address (others_addr) has buffers filled with zeroes. Next, a completion signal is successfully written to a specific memory location (0x1fe940), and the buffer content 01 confirms the signal. This indicates

that the write process has finished, and the completion signal is acknowledged by the FPGA system.

Finally, a read operation retrieves the data from the same addresses (time_step _addr and others_addr), confirming that the data written earlier is available and intact. This read operation shows the same buffer content that was written, which proves that the FPGA can read from and write to memory correctly.

The overall process demonstrates successful read and write operations to the FPGA board, which confirms that the board can communicate with memory

```
----------------------------------------WRITING TO MEMORY----------------------------------------
time_step_addr: (0x0),
Buffer contents:
3C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

----------------------------------------WRITING TO MEMORY----------------------------------------
others_addr: (0x40),
Buffer contents:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

----------------------------------------WRITING TO MEMORY----------------------------------------
Completion signal written successfully to address 0x1fe940
Buffer contents:
01
----------------------------------------READIG FROM MEMORY----------------------------------------
time_step_addr: (0x0),
Buffer contents:
3C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

others_addr: (0x40),
Buffer contents:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

FIGURE 7.1: Connection to the board

7.1.1 A Simple Adder

To ensure effective internal read and write operations at the cl_dram_dma_axi_mstr, create a component named adder_512. This component is designed to operate within the memory, with its primary function being to read two 512-bit values from specified memory addresses, add them together, and then write the result to a different specified memory address.

For this purpose, a C program is developed wherein the function performs two write operations to different memory addresses using the contents of the write buffer, which stores a value (for example 5). This step is intended to store values to be processed by the FPGA. A third write operation modifies the first byte of the write buffer to '1' and writes it to another address, signaling the completion of the write process. Following this, the program initiates reading operations that commence in a loop, polling a specific memory address every 100 milliseconds to check for a result. This loop continues until the data at this address changes from zero, indicating that the FPGA has completed an operation, likely an addition, and written the result back to memory. The loop utilizes buffer comparison to detect changes, comparing the read buffer against a zeroed buffer to determine when the FPGA writes a non-zero result. The results are displayed below.

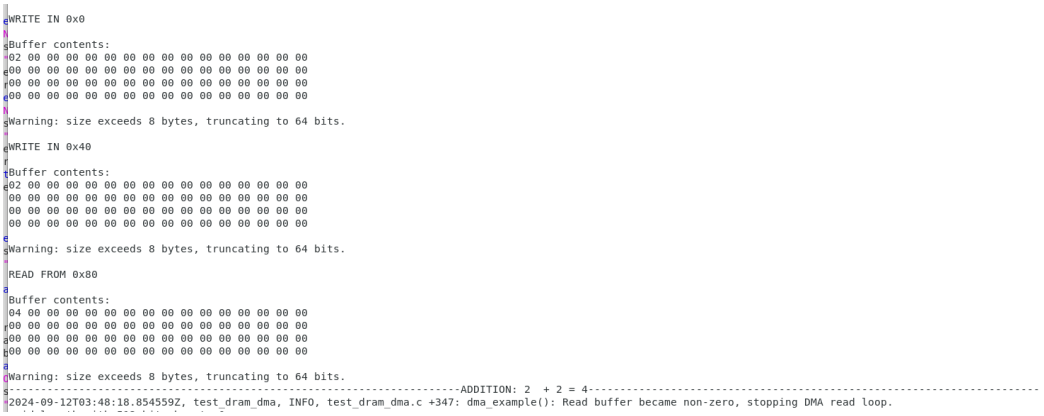


FIGURE 7.2: ADD 2+2

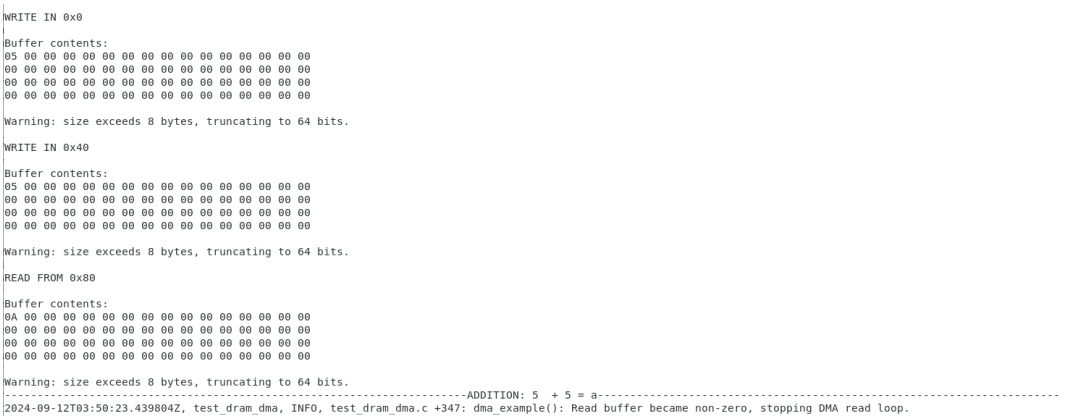


FIGURE 7.3: ADD 5+5

From the screenshots above, it is clear that the adder is functioning correctly, and both the external and internal read/write operations are working as expected.

7.2 Examples of Use

7.2.1 CA models

The primary objective of this thesis, in addition to learning and understanding the AWS F1 Cloud, is to successfully run a Cellular Automata model on it. To achieve this, the serialized file (serialized_data.bin) mentioned in Chapter 4, which is used by the CAD tool, must be manually uploaded to the Amazon operating system. Using fread, deserialization, and the functions fpga_dma_burst_read and fpga_dma_burst_write, as described in Chapter 6, the initial grid, transition rules, and all parameters necessary for system initialization can be loaded. The initial grid used for both examples in the next subsections is as follows:

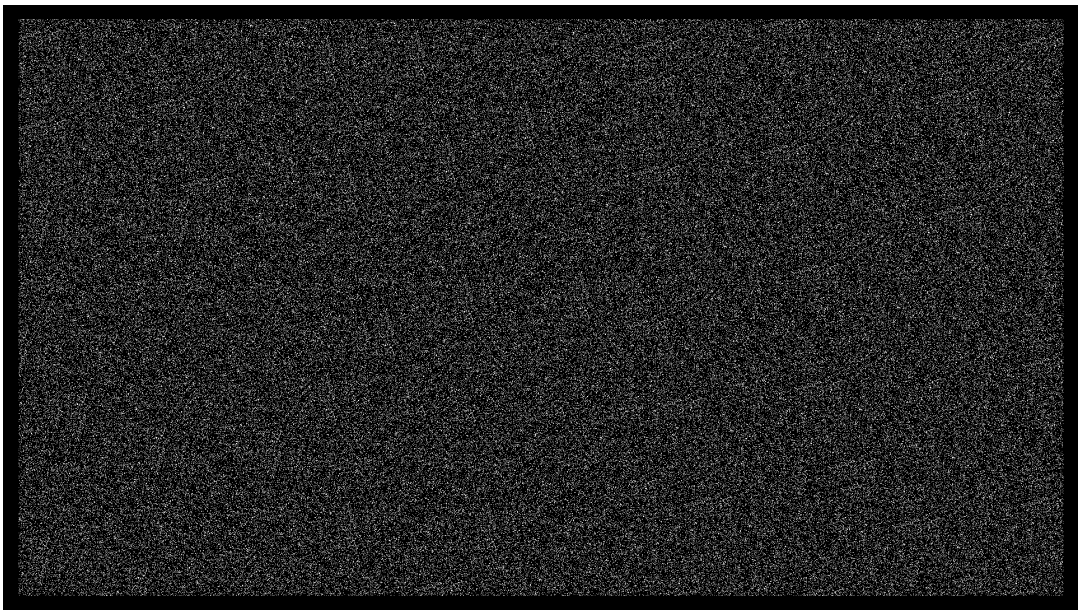


FIGURE 7.4: initial grid

Artificial Physics

The Artificial Physics model is a type of totalistic cellular automaton (CA) rule, where each cell in a grid can have one of two possible states: "dead" or "alive." It uses a large neighborhood of cells, arranged in a 21 × 21 grid, to determine how each cell will evolve. The neighborhood forms a pattern resembling concentric circles, and the initial state of the model consists of randomly distributed "alive" cells, with a ratio of 1 alive cell for every 7 dead cells.

In this model, the state of a cell at position (i, j) at time t is determined by summing up the weighted states of the neighboring cells within a radius r. The formula for this summation is:

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

where $c_t(x,y)$ is the state (alive or dead) of the neighboring cell at (x,y) at time t, and $w(x-i, y-j)$ is the weight of each neighboring cell based on its position relative to (i,j).

Once the sum $S_t(i,j)$ is calculated, the next state of the cell $c_{t+1}(i,j)$ is determined using the following transition rules:

- If $S_t(i,j)$ is between 0 and 19, the cell becomes "dead" (0).
- If $S_t(i,j)$ is between 19 and 23, the cell becomes "alive" (1).
- If $S_t(i,j)$ is between 23 and 58, the cell becomes "dead" (0).
- If $S_t(i,j)$ is between 59 and 100, the cell becomes "alive" (1).
- For any other value of $S_t(i,j)$, the cell becomes "dead" (0).

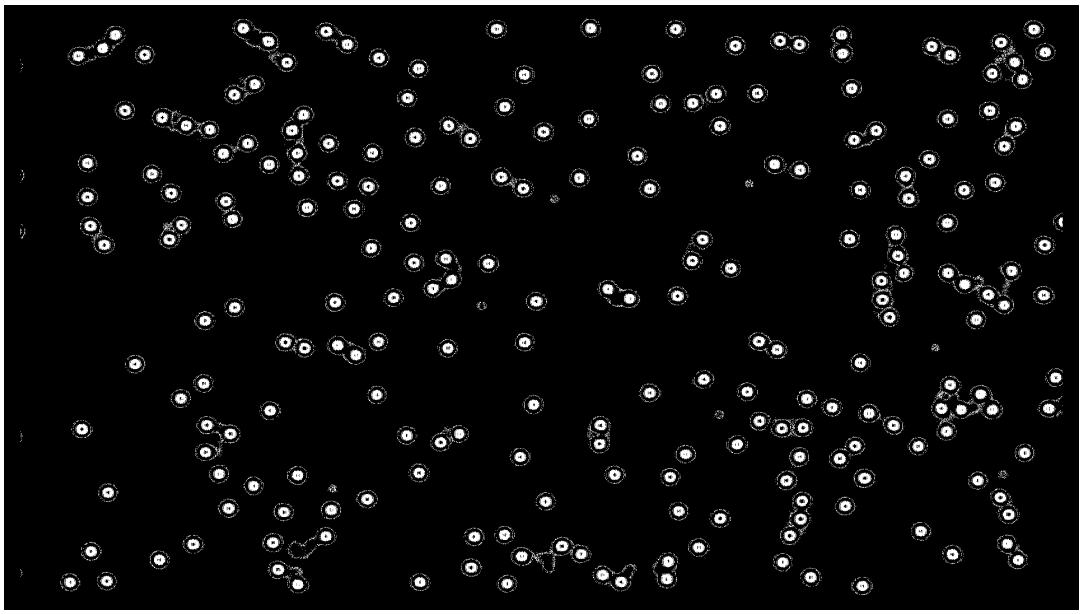


FIGURE 7.5: generation 60

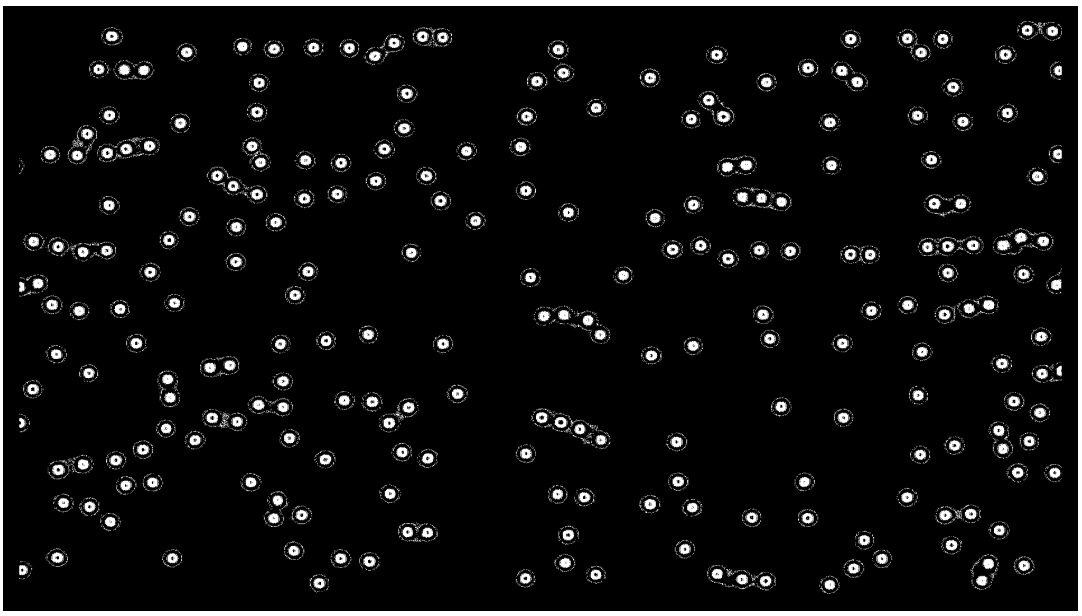


FIGURE 7.6: generation 1000

It calculates a total influence on a central cell due to the weighted contributions coming from the states of its neighbors. Depending on the computed total influence, the state of the central cell is updated based on some predefined transition rules. The local interactions of this kind between cells are iterated in time so that the developments of complex behaviors and structures arise from simple, individual cell rules.When these rules are applied to the grid over time, the simulation generates patterns similar to those Figures 7.5 and 7.6. As the simulation advances, structures resembling atoms start to emerge within the automaton's "universe." These atoms can bond with one another when they come close. However, after the simulation runs for a long enough duration, same atoms eventually disappear.

To summarize:

- The model evolves based on the states of nearby cells, using a large neighborhood and specific weighted rules.
- Over time, the simulation gives rise to atom-like structures that may combine into molecules before eventually disappearing after enough time passes. It finds applications across physics, biology for exploring emergent phenomena.

The Game Of Life

This model utilizes an outer-totalistic rule and demonstrates the second case supported by this configuration, where the central state also influences the transition conditions. The most common outer-totalistic model is the Game of Life, which has been applied to this new architecture. The results show several well-known patterns, primarily falling into two groups: Still Lifes and Oscillators.

Still Lifes are patterns that remain unchanged as the simulation progresses, while Oscillators return to their original state after a set number of steps, cycling repeatedly. In the case of Oscillators, after a certain number of generations, the pattern returns to its initial state and repeats cyclically. For Still Lifes, the pattern remains stable and unchanged regardless of how many generations the simulation runs, making the image resemble the initial configuration after a lot of generations . The transition function is :

$$\text{next\_state} = \begin{cases} 1, & \text{if sum} \in [2,3] \text{ and current\_state} = 1; \\ 1, & \text{if sum} \in [3,3] \text{ and current\_state} = 0; \\ 0, & \text{otherwise.} \end{cases}$$

As it happens in the simulations, in the 100th and 500th generation in the following images7.7 and 7.8, the oscillators change over time. After many generations, as shown in the image 7.9, they return close to their original state, as we expected based on the model mentioned earlier.

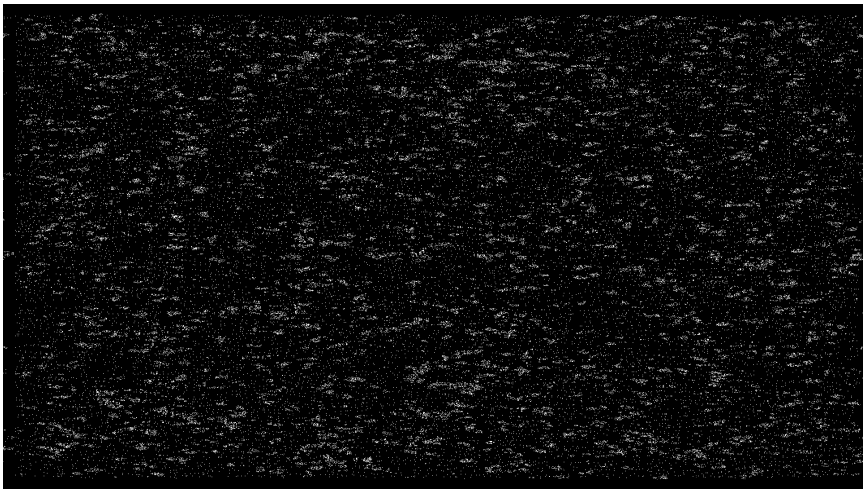


FIGURE 7.7: generation 100

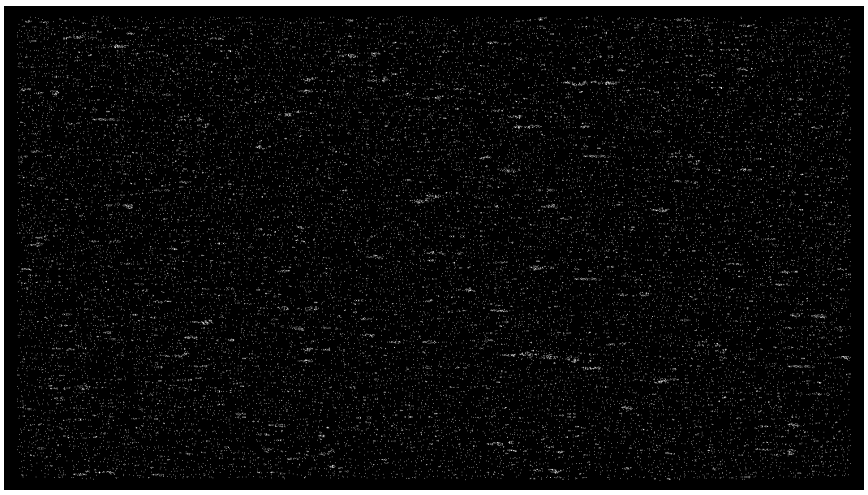


FIGURE 7.8: generation 500

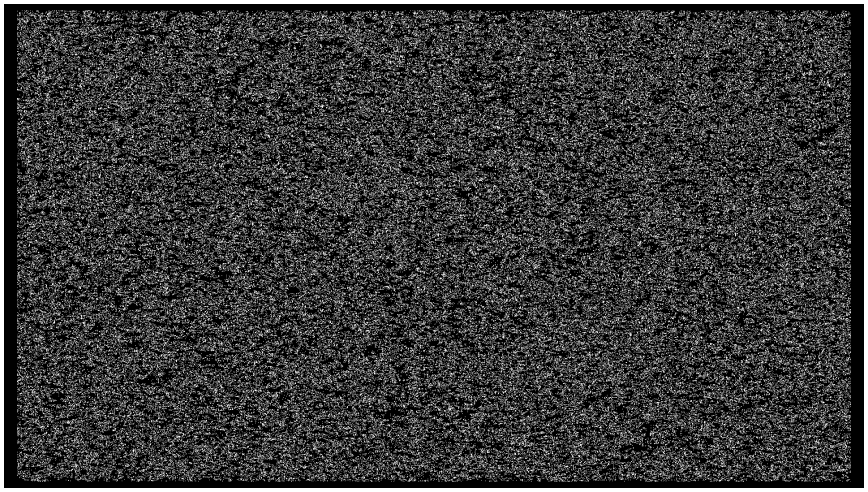


FIGURE 7.9: generation 10000

7.3 Results

Both of the above results represent basic models of the two categories of Cellular Automata (CA) models and are very similar to the results of the two pre-existing architectures. These results can logically be explained by the transition function and the neighborhood topology. In both cases, a rectangular grid is used, but the architecture is designed to also support cylindrical and toroidal grids. What has not been fully implemented yet, to make the system even more user-friendly (since the need for the user to have a physical board has already been eliminated), is to connect the CAD tool directly with AWS so that manual file transfers are no longer required. However, future expansions will be discussed later.

7.4 Performance Results

As illustrated below, our system effectively overcomes the computational and memory constraints typical of general-purpose CPUs. It achieves a significant speedup, up to 21 times, when compared to an Intel Core i7-7700HQ CPU (1 core) running highly optimized (-O3) software written in C. On the

Cellular Automaton	**i7-7700HQ, 1000 generations**	**Our Design, 1000 generations**	**Speedup**
Artificial Physics, n=21	538.77 sec	25.4 sec	21.21x

TABLE 7.1: Comparison of performance between the i7-7700HQ and our FPGA-based design

other hand, a GPU consumes more power than an FPGA. At the system level, a contemporary GPU requires 170 W, whereas our AWS FPGA-based system consumes approximately 37 W. Therefore, for similar speeds, the power consumption of our FPGA-based system is typically less than one-fourth the power of a GPU, and hence the total energy required for the computation scales accordingly.

Architecture	**Neighborhood Size**	**Performance**
Margolus, 1993-2001, CAMs	experimented with up to 11x11	10 gen/sec for a 512x512 grid with 3-bit cells
Gibson et al., 2015, Workstation with Nvidia GTX 560 Ti	experimented with up to 11x11	≈65x over serial for Game of Life on a 2048x2048 grid
Millan et al., 2017, Nvidia TitanX GPU	experimented with up to 11x11	21.1x over serial for Game of Life on a 4096x4096 grid
Current Work, 2024, AWS Board	experimented with up to 21x21	21.2 x over serial for Artificial Physics on a 1920x1080 grid

TABLE 7.2: Performance comparison of different architectures.

Chapter 8

Conclusions and Future Work

This chapter presents the future work, focusing on the steps needed to enhance the Cellular Automata tool and deploy it on Amazon Cloud. It begins by discussing the creation of a Python script to automate the generation and uploading of configuration files to cloud storage, removing the need for manual intervention. This will be done using Amazon's boto3 library to dynamically select and load AGFI bitstream files based on the cellular automaton's parameters, such as CELL SIZE and GRID TYPE. Additionally, hardware upgrades are suggested for more advanced models like anisotropic cellular automata and the Greenberg-Hastings model, which require more complex logic. The chapter also highlights the key steps to develop a scalable cloud service, including security, infrastructure automation, continuous integration, and monitoring.

8.1 Conclusions

In this thesis, we have successfully developed and implemented an FPGA-based framework for cellular automata on the Amazon F1 Cloud. The purpose of the thesis was achieved by removing the board from the use of CA accelerator, allowing users to easily utilize the CA tool without the need for a physical board.The architecture was simplified by removing several components, which enhanced the speed of calculation and data transportation. These improvements were further supported by adjustments in burst size, the use of time multiplexing, and the implementation of two separate channels for reading and writing.Despite this, the architecture requires further changes to accommodate even more models. Although the initial steps have been taken to remove the physical board, additional measures are necessary

to make it usable for users. These will be discussed in the following subsection.

8.2 Extended CA Tool

To create a proper service on Amazon Cloud, the first step is to develop a Python script that performs all the necessary tasks. The CAD tool mentioned in the previous chapter must be properly configured. The UART component should be removed, and the corresponding AGFI (bitstream files for Amazon Cloud) should be generated. Using Amazon's boto3 library, the appropriate image will be dynamically loaded based on the selected CELL SIZE (4 or 8) and GRID TYPE (RECTANGULAR, CYLINDRICAL, or TOROIDAL). This process will automate the upload and initial configuration on Amazon, eliminating the need for the manual intervention currently required.

Python Code Example for AGFI Selection and EC2 Client Interaction:

```
import boto3

# Initialize boto3 client for EC2
ec2_client = boto3.client('ec2')

# Function to select AGFI based on CELL SIZE and GRID TYPE
def get_agfi_id(cell_size, grid_type):
    # Define AGFI mappings for different configurations
    agfi_mapping = {
        '4': {
            'RECTANGULAR': 'agfi-rect4-id',
            'CYLINDRICAL': 'agfi-cyl4-id',
            'TOROIDAL': 'agfi-tor4-id'
        },
        '8': {
            'RECTANGULAR': 'agfi-rect8-id',
            'CYLINDRICAL': 'agfi-cyl8-id',
            'TOROIDAL': 'agfi-tor8-id'
        }
    }

    # Return the corresponding AGFI ID
    return agfi_mapping[cell_size][grid_type]

# Example input
cell_size = '4'  # or '8'
grid_type = 'RECTANGULAR'  # or 'CYLINDRICAL', 'TOROIDAL'
```

```
28
29 # Get AGFI ID based on user input
30 agfi_id = get_agfi_id(cell_size, grid_type)
31
32 # Launch the EC2 instance with the selected AGFI
33 response = ec2_client.modify_fpga_image_attribute(
34     FpgaImageId=agfi_id,
35     Attribute='loadPermission'
36 )
37 print(f"AGFI {agfi_id} loaded successfully!")
```

8.3 Extended Hardware

In future work, several extensions are necessary to enhance the current reprogrammable framework to support more complex cellular automata models. First, the **Greenberg-Hastings** model, originally developed with a small von Neumann neighborhood and a limited number of cell states, was expanded by Kyparissas to handle larger neighborhoods (such as 29 × 29) and multiple states per cell (up to 16). This requires the implementation of dynamic transition rules, where the state of each cell is determined by the excitations of its neighbors.

$$\text{c\_t+1(i, j)} = \begin{cases} 1, & \text{if } c_t(i,j) = 0 \text{ and the total excited neighbors} > T, \\ c_t(i,j) + 1, & \text{if } c_t(i,j) > 0, \\ c_t(i,j), & \text{otherwise,} \end{cases}$$

3 states per cell: "quiescent", "excited" and "refractory"

Additionally, the framework should incorporate **anisotropic models**, which simulate directionally dependent behavior. These models utilize a weighted Moore neighborhood, where each cell has a distinct weight based on its distance from the center, and can support 256 states per cell.

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

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

To accommodate these complex models, it is essential to extend the hardware architecture by integrating advanced logic and arithmetic capabilities that are necessary for calculating the above-mentioned models.

8.4 Finalizing the Product

To properly create a service on Amazon Cloud that people can use, follow these steps:

- **Package the Code:** Use Docker containers, AMIs, and CloudFormation templates for packaging the application code.
- **Automate Infrastructure:** Implement Infrastructure-as-Code using CloudFormation or Terraform to automate infrastructure provisioning.
- **Secure the Environment:** Ensure adherence to AWS security best practices, including proper IAM configurations, data encryption, and network security controls.
- **Set up CI/CD:** Automate the build, test, and deployment pipeline with AWS CodePipeline or other CI/CD tools.
- **Monitoring and Logging:** Use AWS CloudWatch and AWS X-Ray to gain operational insights into the system, monitor performance, and track logs.
- **API Exposure:** Leverage AWS API Gateway or an appropriate frontend service to provide external access to your service for customers.
- **Publish to AWS Marketplace:** Package your service according to AWS Marketplace requirements and submit it for review.
- **Cost Management:** Implement cost tracking and optimizations using AWS Cost Explorer and other tools to ensure efficient use of resources.

References

[1] Andrew Ilachinski. *Cellular Automata: a Discrete Universe*. World Scientific, 2001.

[2] Martin Gardner. "Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game "Life"". In: *Scientific American* 223.4 (1970).

[3] Nikolaos Kyparissas and Apostolos Dollas. "An FPGA-Based Architecture to Simulate Cellular Automata with Large Neighborhoods in Real Time". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 95–99. DOI: 10.1109/FPL.2019.00024.

[4] R.Wm. GOSPER. "EXPLOITING REGULARITIES IN LARGE CELLULAR SPACES". In: *Physica 10D* (1984).

[5] John von Neumann and Arthur W. Burks. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[6] Tommaso Toffoli. "CAM: A High-Performance Cellular-Automaton Machine". In: *Physica D: Nonlinear Phenomena* 10.1-2 (1984).

[7] Tommaso Toffoli and Norman H. Margolus. *Cellular Automata Machines - A New Environment for Modeling*. MIT Press, 1987.

[8] Norman H. Margolus. "CAM-8: A Computer Architecture Based on Cellular Automata". In: *Pattern Formation and Lattice-Gas Automata, AMS* (1993).

[9] Monica Dascalu. "Cellular Automata Hardware Implementations". In: *Romanian Journal of Information Science and Technology* (2016).

[10] Rolf Hoffmann, Klaus-Peter Völkmann, and Mark Sobolewski. "The Cellular Processing Machine CEPRA-8L". In: *Mathematical Research* 81 (1994), pp. 179–188.

[11] Christian Hochberger et al. "The CEPRA-1X Cellular Processor". In: *Reconfigurable Architectures: High Performance by Configware, IT Press, Bruchsal* (1997).

[12] Christian Hochberger et al. "The Cellular Processor Architecture CEPRA-1X and its Configuration by CDL". In: *IPDPS 2000. Lecture Notes in Computer Science, vol 1800*. 2000, pp. 898–905.

[13] Paul Shaw, Paul Cockshott, and Peter Barrie. “Implementation of Lattice Gases Using FPGAs”. In: *Physica D: Nonlinear Phenomena* 12.1 (1996), pp. 51–66.

[14] NIKOLAOS KYPARISSA and APOSTOLOS DOLLAS. “Large-scale Cellular Automata on FPGAs: A New Generic Architecture and a Framework”. In: *ACM Journals* (2020).

[15] Tomoyoshi Kobori, Tsutomu Maruyama, and Tsutomu Hoshino. “A Cellular Automata System with FPGA”. In: *9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Rohnert Park, CA, USA, 2001, pp. 120–129.

[16] Shakeeb Murtaza, Alfons G. Hoekstra, and Peter M. A. Sloot. “Cellular Automata Simulations on a FPGA Cluster”. In: *International Journal of High Performance Computing Applications, SAGE* 25.2 (2010), pp. 193–204.

[17] André C. Lima and João Canas Ferreira. “Automatic Generation of Cellular Automata on FPGA”. In: *9th Portuguese Meeting on Reconfigurable Systems*. Coimbra, Portugal, 2013, pp. 51–58.

[18] Emmanouil Mylonakis. “Development of a CAD tool and Hardware Design in Order to Execute Cellular Automata on a Reconfigurable Platform by non-FPGA-Conversant Users”. PhD thesis. Technical University of Crete, 2024.

[19] Nikolaos Kyparissas and Apostolos Dollas. “An FPGA-Based Architecture to Simulate Cellular Automata with Large Neighborhoods in Real Time”. In: 14.1 (2020), pp. 1–32. DOI: 10.1145/3423185.

[20] Nikolaos Kyparissas and Apostolos Dollas. “Field Programmable Gate Array Technology as an Enabling Tool Towards Large-Neighborhood Cellular Automata on Cells with Many States”. In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. 2019, pp. 940–947. DOI: 10.1109/HPCS48598.2019.9188084.