NTNU – Trondheim
Norwegian University of
Science and Technology

The Cellular Automata Research Platform: Revised, Rebuilt and Enhanced

Per Thomas Lundal

Master of Science in Computer Science
Submission date: June 2015
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Sammendrag

Naturen har mange attraktive egenskaper som ingeniører håper på å en dag kunne gjenskape i menneskeskapt teknologi for å revolusjonere databehandling. De inkluderer blant annet reproduksjon, læring, tilpasning og massiv parallellisering. Én bioinspirert struktur er den Cellulær Automaten (CA). Den kan gjenskape flercellede organismers massive parallellitet, distribusjon og lokal samhandling.

På NTNU har tre masteroppgaver gått med til å lage en FPGA-plattform med formål om å muliggjøre forskning på CAer i kombinasjon med kunstig evolusjon og utvikling. Hovedprinsippet er at en genetisk algoritme (GA) brukes til å lage utviklingsregler, som så blir brukt til å konstruere en CA, som til slutt brukes til å beregne en fitnessverdi. Fitnessverdien blir så matet tilbake inn i GAen og prosessen gjentas til en god løsning er funnet.

I forventning om å få ny maskinvare med en større FPGA, gikk den nyligste masteroppgaven ut på å utnytte den økte ressursmengden til å bedre ytelsen og utvide CAen til 3D. Men på grunn av produksjonsproblemer kunne dessverre ikke maskinvaren leveres i tide. Derfor ble ikke en ny kommunikasjonsmodul implementert og kun grunnleggende simuleringstester kunne utføres.

I det innledende spesialiseringsprosjekt til denne masteroppgaven ble en ny kommunikasjonsmodul implementert og integrert i plattformen, som tillot skikkelig maskinvareverifisering. Den viste at designet hadde mange problemer, inkludert feilende instruksjoner, stor bruk av utdaterte elementer og separate versjoner for 2D og 3D. I denne masteroppgaven har derfor hele plattformen blitt revidert og gjenoppbygd.

Den nye plattformen løser alle store problemer med den gamle og legger til nye forbedringer som mer avansert kontrollflyt og et adaptivt programvare API. Flere og mer fininstillbare byggparametre tillater større justering av ytelse og gjør det mulig å få plass til større CAer på FPGAen.

Funksjonaliteten til alle instruksjoner er verifisert i maskinvare og demonstrert med et program som lager repliserende strukturer. Dette viser at plattformen er komplett og klar for bruk til forskning. Et klokkeproblem med kommunikasjonsmodulen fører for tiden til at hele plattformen kjører på halv hastighet, men viss det fikses vil den rå CA ytelsen bli 35% bedre i 2D og 300% bedre i 3D sammenlignet med det forrige designet. For disse ytelseskonfigurasjonene er ressursbruken vesentlig mindre i 2D og omtrent lik i 3D.

Abstract

Nature has many attractive properties that engineers hope to once incorporate into man-made technology to revolutionize computing. Properties include, among other things, reproduction, learning, adaption and massive parallelism. One bio-inspired computational structure is the Cellular Automaton (CA), which can mimic the massively parallel, distributed and locally interactive nature of multi-cellular organisms.

At NTNU, three master theses have gone into creating a Field-Programmable Gate Array (FPGA) platform whose purpose is to allow research on CAs in combination with artificial evolution and development. The main principle is that a Genetic Algorithm (GA) is used to create development rules, which are then used to build a CA, which can finally be used to compute a fitness value. The fitness value is then fed back fed into the GA and the process is repeated until a good solution is found.

In expectation of new hardware with a larger FPGA, the purpose of most recent thesis was to take advantage of increased resources to improve speed and to extend the CA into 3D. However, the hardware failed to be delivered on time due to manufacturing problems. This caused the new communication module to remain unimplemented and only allowed rudimentary simulation testing.

In a specialization project leading up to this thesis, a new communication module was implemented and integrated into the platform, allowing proper hardware verification. It showed that the design had many issues, including failing instructions, extensive use of outdated features and separate versions for 2D and 3D CAs. In this thesis, the entire platform has therefore been revised and rebuilt.

The new platform solves all major issues with the previous and adds further enhancements such as more advanced control flow and an adaptive software API. More and better fine-tunable build parameters allow wider adjustment of performance and make it possible to fit larger CAs within the FPGA.

The functionality of all instructions is verified in hardware and a program that creates replicating structures is demonstrated, proving that the platform is complete and ready to be used for research. A clocking issue with the communication module is currently reducing the entire platform to half speed, but if fixed, the raw CA performance will be 35% higher in 2D and 300% higher in 3D compared to the previous design. For those performance configurations, resource usage is significantly lower in 2D and equivalent in 3D.

Preface

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

The thesis counts for 30 credits and is the continuation of a 15-credit specialization project conducted autumn 2014. It concludes a 5-year master of science study in Computer Science.

I would like to personally thank Gunnar Tufte for his invaluable inspiration and support, and Odd Rune Strømmen Lykkebø who stepped in during his brief absence.

Per Thomas Lundal
2014 June 14th

Contents

Sammendrag **i**

Abstract **iii**

Preface **v**

Contents **vii**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Outline . . . 2

2 Background **5**
2.1 Evolution . . . 6
2.1.1 Genetic Algorithms . . . 6
2.2 Development . . . 7
2.2.1 Lindenmayer Systems . . . 7
2.3 Cellular Automata . . . 8
2.3.1 Neighborhoods . . . 10
2.3.2 Complexity Classes . . . 11
2.3.3 Evolution and Development in CAs . . . 11
2.4 FPGA . . . 12
2.4.1 Sblock . . . 12
2.5 PCI Express . . . 13
2.6 Related Work . . . 15
2.6.1 CAM-8 . . . 15
2.6.2 CAM-Brain Machine . . . 16
2.6.3 BioWall . . . 16

3 Previous Work **19**
3.1 Djupdal . . . 19
3.2 Aamodt . . . 21
3.3 Støvneng . . . 22

3.4 Issues 23

4 Development Platform **25**
4.1 Spartan-6 SP605 Evaluation Platform 25
4.2 Hardware Setup 26
4.3 Software Setup 27

5 Implementation **29**
5.1 General Concepts 31
5.1.1 Parameterization 31
5.1.2 Pipelining 32
5.1.3 Buffers 32
5.2 Communication 33
5.2.1 PCI Express Endpoint Core 34
5.2.2 Reception Engine 34
5.2.3 Transmission Engine 35
5.2.4 Request Handler 35
5.3 Fetch 36
5.3.1 Fetch Communication 36
5.3.2 Fetch Handler 36
5.4 Decode 37
5.5 Control 38
5.5.1 Rule Writer 39
5.5.2 LUT Writer 39
5.5.3 Information Sender 39
5.5.4 Fitness Sender 39
5.5.5 Rule Vector Reader 40
5.5.6 Rule Numbers Reader 40
5.5.7 Cell Writer Reader 40
5.6 Cell Storage 42
5.7 Development 43
5.7.1 Cell Fetcher 43
5.7.2 Rule Fetcher 45
5.7.3 Rule Testers 46
5.7.4 Hit Processors 46
5.8 Cellular Automaton 46
5.8.1 State Machine 47
5.8.2 Sblock Matrix 48
5.8.3 Live Counter 48
5.9 Fitness 48
5.9.1 Live Count 49
5.9.2 DFT 49
5.10 Software API 51
5.10.1 Main API 51
5.10.2 Optional APIs 52

5.10.3 Communication 52
5.10.4 Compilation 53

6 Verification **55**
6.1 Functional Tests 55
6.1.1 DFT 55
6.2 Example Replicator 56

7 Discussion **59**
7.1 Performance 59
7.1.1 Communication 60
7.1.2 Cellular Automaton 61
7.2 Resource Usage 62
7.3 Challenges 63
7.4 Future work 63
7.5 Warnings 64

8 Conclusion **65**

Bibliography **67**

A Test Descriptions **71**

B Attached Files **73**

C Instruction Set Architecture **79**

D Specialization Project Report **125**

List of Figures

2.1 POE model 5
2.2 Genetic algorithm 6
2.3 Development 7
2.4 Dragon curve 8
2.5 Computing principles 9
2.6 von Neumann neighborhood 10
2.7 Complexity classes 11
2.8 CA system with evolution and development 12
2.9 FPGA 13
2.10 Sblock 13
2.11 PCI Express structure 14
2.12 CAM-8 system diagram 15
2.13 BioWall 16

3.1 System design 19
3.2 Djupdal's hardware design 20
3.3 Aamodt's hardware design 21
3.4 Støvneng's hardware design 22

4.1 SP605 26
4.2 Hardware setup 26

5.1 High-level system diagram 29
5.2 Detailed system diagram 30
5.3 Pipeline wave diagram 32
5.4 FIFO buffer wave diagram 33
5.5 Communication module 33
5.6 Reception Engine state machine 34
5.7 Transmission Engine state machine 35
5.8 Fetch module 36
5.9 Fetch Handler state machine 37
5.10 Control modules 38
5.11 Cell Writer Reader 41
5.12 Combiner operation 41
5.13 Cell Writer Reader state machine 42

5.14 Development module 43
5.15 Development module wave diagram 44
5.16 Development module state machine 45
5.17 Cell Fetcher state machine 45
5.18 CA module 47
5.19 CA module state machine 47
5.20 DFT Fitness 50
5.21 DSP Wrapper 50
5.22 DFT state machine 51
5.23 API 51

6.1 Replicator 58

7.1 Example program 61
7.2 Test program 61

List of Tables

3.1 Issues 23

5.1 Parameters. 31
5.2 Special requests 35

6.1 DFT error 56

7.1 Communication performance 60
7.2 Resource usage 63

Chapter 1

Introduction

It is predicted that conventional CPU architectures will be unable to continue to scale in about a decade [8]. Many engineers are therefore investigating entirely different technologies in hope of finding viable alternatives. Some have looked towards nature; at biological organisms whose complexities far outweigh what humans have so far been able to engineer. Additionally, biological systems exhibit a wide range of characteristics that could possibly revolutionize computing, such as reproduction, learning, adaption, massive parallelism, graceful degradation, self-assembly and self-repair.

This has formed the field of bio-inspired computation, where the principles of nature in the form of artificial evolution, development and learning are used in the creation of computer systems. Some focus on mimicking the structure of the human brain, in the form of artificial neural networks, to create robot controllers [5]. Others focus on the emergent behaviors from thousands or millions of individual cells in Cellular Automata (CAs).

Bio-inspired computing has been an area of research at NTNU for more than a decade. In 2002, NTNU invested in dedicated FPGA hardware for the purpose of creating a platform for experimentation with CAs in combination with artificial evolution and development.

The initial work was done by Djupdal, and then extended by Aamodt shortly after. The CA was implemented as a matrix of sblocks, a form of reprogrammable CA cells, connected to a development unit capable of simulating cell growth and change. The hardware platform was connected to and controlled by a computer over a PCI connection.

A general use-case for the platform is to have the computer run a Genetic Algorithm (GA), where the genotype represents the development rules and initial CA state. Development is then used to create a phenotype in the form of a CA structure, which can be used for computation and to produce a fitness value. The fitness value is then fed back into the GA until an acceptably good solution is found.

In expectation of new hardware with a larger FPGA and faster PCI Express connection, Støvneng refurbished the design in 2014. He took advantage of the added resources to greatly improve the performance of the platform, giving a speedup of 4 or more for many operations. Additionally, he extended the CA into 3D and added a Discrete Fourier Transform (DFT). However, since the hardware did not arrive in time, the new design was only tested in simulation and the communication interface was not upgraded.

The task of the specialization project leading up to this thesis was to finish the extended platform by implementing a new PCI Express communication module, and to verify the platform's functionality in hardware. The verification process uncovered many issues, some of which made the platform unusable, and others which made debugging and fixing very difficult. This led to the decision of revising and rebuilding the entire platform from scratch in this thesis.

1.1 Outline

The thesis is organized as follows:

- Chapter 2 – Theoretical background, technology and related work. This chapter gives an overview of the relevant research that this thesis is based on and the technology which is used.
- Chapter 3 – Previous designs and implementations. This chapter states a brief history of the platform that this thesis builds on and the main issues that needs improvement.
- Chapter 4 – Development platform and setup. This chapter describes the hardware and software systems used in this thesis and their setups.
- Chapter 5 – Implementation details. This chapter provides in-depth descriptions of all parts of the rebuilt and enhanced platform.
- Chapter 6 – System verification. This chapter asserts the functionality of the platform through tests and an example program.
- Chapter 7 – Performance, challenges and future work. This chapter analyzes the system's performance, discusses difficulties and compromises during development, and mentions possible improvements.
- Chapter 8 – Concluding remarks. The final chapter concludes this paper by reviewing the new platform, its performance and its potential for future applications.
- Appendix A – Functional tests. This appendix briefly describes the test programs that together provide test coverage of all instructions.

- Appendix B – Attachment index. This appendix lists the attached files comprising the hardware design and software API.
- Appendix C – Instruction Set Architecture. This appendix provides a complete specification of all instructions, the rule format and the LUT format.
- Appendix D – Specialization project. This appendix holds the paper that led up to this thesis, in which a PCI Express-based communication module is implemented and integrated into the previous design.

Chapter 2

Background

The field of bio-inspired computing encompass a wide variety of technologies that take advantage of different natural concepts. In [31], Sipper et al. partition these technologies into a space along three axes (illustrated in Figure 2.1):

- Phylogeny: Temporal adaption to the environment caused by mutation and reproduction. It is named after the evolution of the species.
- Ontogeny: Growth through cell division or equivalent methods. An example is a zygote ("mother cell") replicating to form a larger multi-cellular organism.
- Epigenesis: Adaptation through a lifetime of interactions with the environment. This can loosely be defined as learning.

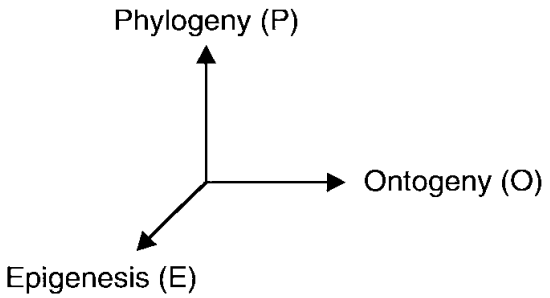


Figure 2.1: The POE model for bio-inspired hardware.

This thesis focuses on the first two axes, phylogeny and ontogeny, in the form of artificial evolution and development. This chapter goes into details about the two concepts, some of the technologies that can take advantage of them and other technologies that are used in this thesis.

2.1 Evolution

Evolution is the natural process that over time advances species by letting the fit survive to reproduce while the less fit perish. Essentially, each generation includes slight variations to gradually create a species that is more adapted to the environment by having the fit traits passed on to subsequent generations. In nature, this process effectively creates more fit solutions to the problem of life, which is to reproduce in a changing environment.

Similarly, artificial evolution can be used by computers to evolve solutions to computational problems instead of manually creating them, by the means of Evolutionary Algorithms (EAs) [20]. EAs can search millions of possible solutions, guided by their fitness scores, and find solutions that humans would never have imagined. In contrast to nature, where years usually pass between each generation, powerful computers can create hundreds of generations every second and find good solutions in relatively short time.

EAs have been successfully applied to many scientific tasks. For example, NASA has had great success with evolving antenna designs [21], and Floreano and Mondada have evolved robot controllers with homing navigation [9]. An interesting feature is that EAs can find ways to exploit hardware in ways that human designers cannot comprehend [34]. This can be due to complex parallel interactions, or usage of properties that are not fully understood [35].

2.1.1 Genetic Algorithms

A very common type of EA is the Genetic Algorithm (GA) presented in [16]. It represents each solution as a genotype, a binary string used as a blueprint to create the solution itself, the phenotype. The genotype is comparable to nature's DNA, and it is this genetic material which is modified in the evolutionary process.

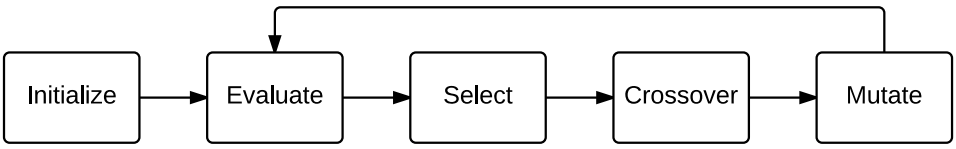
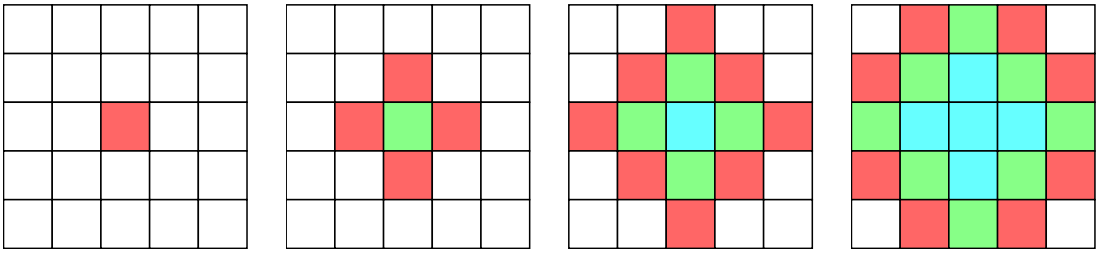


Figure 2.2: A genetic algorithm. The cycle is broken when the fitness is above a given threshold.

The GA process is shown in Figure 2.2 and works as follows: First, a base population with random genotypes is generated. Then, each phenotype is constructed and evaluated using a fitness function. If a solution has a fitness score above a set threshold, the process stops. Otherwise, a new population is created by selecting solutions with high fitness scores, crossing their genotypes, and mutating the results, before repeating the process.

2.2 Development

The process that transforms a genotype into a phenotype is called development, and can be regarded as a form of decompression algorithm [18]. In nature, this process is seen when a single cell transforms into a more complex multicellular organism, as visualized in Figure 2.3. Unlike with a pure decompression algorithm however, biological systems also use information about their environments to tailor themselves to them. This is known as plasticity.



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2 (d) Iteration 3

Figure 2.3: Example of cells replicating and changing to develop a larger and more complex entity.

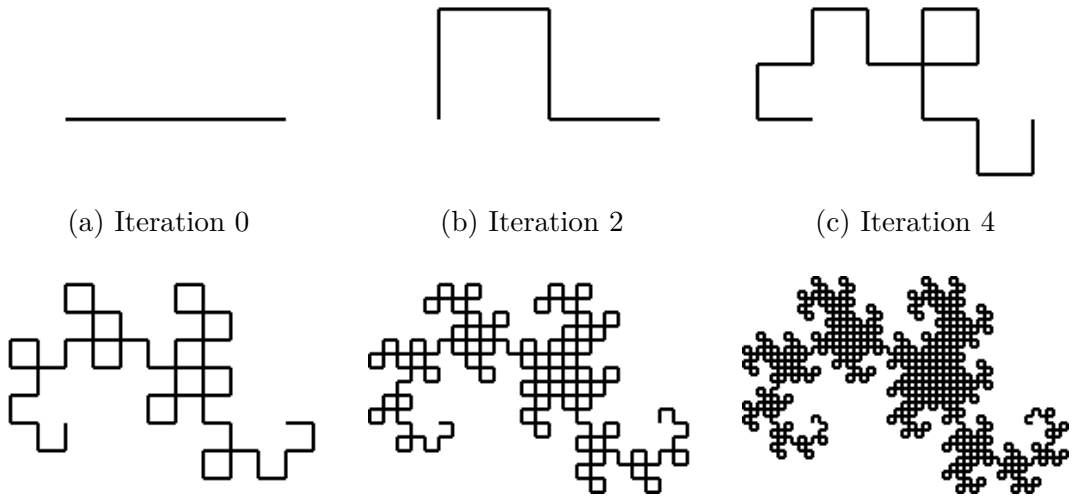
Development is needed because a complete specification of a complex organism requires much more information than practically can be stored; it is several orders of magnitude greater than that of one cell and its development rules. Plasticity and self-repairing abilities are merely bonuses, but have shown to be highly valuable properties. For the same reasons, artificial development is lucrative for building more complex computer systems that are also fault-tolerant and adaptable.

Development is never necessarily "finished"; it can continue during the entire lifespan of the system. Over time, a dynamical system tends to end up in a state of equilibrium, an attractor [19]. The attractor may be a single state, or a series of states that repeat to create a cycle. External interactions, for example suffering damage, will however likely cause the system to break from its current attractor in search for another. If the system ends up providing the same functionality, albeit possibly with a different structure, it is said to have self-repairing abilities, which is a lucrative property in the eyes of hardware engineers.

2.2.1 Lindenmayer Systems

Perhaps the most known and widely used artificial development systems are Lindenmayer Systems (L-Systems). They were introduced by biologist and botanist Lindenmayer in 1968 to describe the growth of plants and fungi [24]. The L-System is a form of parallel generative grammar; starting with an axiom, a string is built by iteratively applying all the grammar rules in parallel. The string is a representa-

tion of the structure of the object; each character symbolizes a branch, twist, turn, stretch or other feature.



(a) Iteration 0 (b) Iteration 2 (c) Iteration 4

(d) Iteration 6 (e) Iteration 8 (f) Iteration 10

Figure 2.4: Heighway dragon curve generated with L-System using [3]. The starting string is FX and the rules are $X \rightarrow XRYFR$ and $Y \rightarrow LFXLY$ where F is forward, R is a right turn and L is a left turn.

In addition to plants, L-Systems are also suitable for generating other structures that grow and branch. Figure 2.4 shows how they can be used to generate fractals, specifically the dragon curve [10]. It is the pattern that emerges when a piece of paper is folded many times and then all folds opened to 90 degree angles. The corresponding L-System uses only two rules, and is a testament to how very simple development rules can create outstandingly complex shapes. L-Systems have also been applied to other tasks, such as music composition [25].

2.3 Cellular Automata

A Cellular Automaton (CA) is a computational structure made up of vast numbers of very simple functional elements called cells that are arranged in a grid. Each cell contains a state and is connected to a handful of nearby cells to form neighborhoods. At given time intervals the cells then update their states based on transfer functions over the states in their neighborhoods.[1]

CAs are attempts to mimic the structures found in biological lifeforms, where complex results emerge from interactions between many simple cells. The key principles are massive parallelism, local interactions and simple computational units. As seen in Figure 2.5, this is the directly opposite paradigm to the general-purpose serial

[1] CAs are specific cases of Random Boolean Networks [14].

architecture that is common in computers today. However, CAs have been shown to be Turing complete [4, 36], and can therefore perform any task.

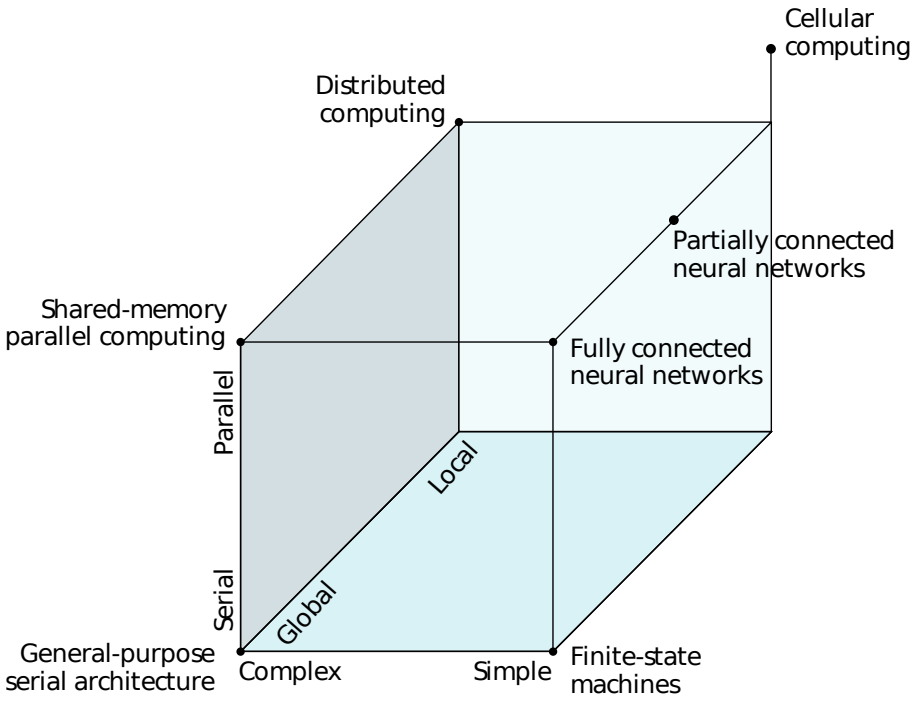


Figure 2.5: Graphical representation of how different computing paradigms relate with regards to computing principles. (Reprinted from [30])

CAs are attractive due to their inherent robustness and scalability when paired with development. In the event of broken cells, signals can simply be rerouted, and to increase the performance, extra cells can be added and the program grown; local communication ensures that there are no bottlenecks. This greatly contrasts modern processors where a broken part normally renders the entire chip unusable, and adding more cores are of limited benefit due to the shared-memory architecture. Increasing the clock rate is no longer an option either, due to the fixed power budget.

A major challenge with CAs is programming. CAs compute by emergence using massive parallelism, while humans mostly solve problems serially [27]. This makes it near impossible for humans to construct programs, unless for very simple problems. Genetic Algorithms are therefore often used.

Other major challenges are the representation of input and parsing of output. Both arise due to the CA's distributed nature. For output, the Discrete Fourier Transform (DFT) over the number of cells with a given state appears promising however [2].

CAs have been also been used in various research. They have been environments for simulating lifeforms and creating replicating machines [36]. A very popular life simulator is Conway's Game of Life [11].

There are many variations of CAs, which are discussed in great detail in [30]. Following is a brief summary: Cell states can have discrete or continuous values. The

transfer function can be represented by exhaustive enumeration or a parameterized expression. The cells may be uniform by having the same transfer function, or non-uniform. Cell updates can be synchronous or asynchronous, and in the latter case the CA can be either deterministic or nondeterministic depending on the order in which cells update. The CA may be fully specified by direct programming or adaptively evolved using a EAs. Finally, there are numerous schemes that can be used to connect cells together to form neighborhoods.

In this thesis, only a specific form of CA is used: Discrete, exhaustively enumerated, non-uniform, synchronous and deterministic. It is possible to use direct programming, but there are systems in place for adaptive evolution.

2.3.1 Neighborhoods

An important property of CAs is how cells are connected to form neighborhoods, as this defines what data will be available for the transfer functions and therefore changes the way data flow through the machine. It is common to connect directly adjacent cells, and sometimes those diagonally adjacent. Cells do not have to be part of their own neighborhoods, but it is common. The von Neumann neighborhood shown in Figure 2.6 is commonly used for 2D CAs, and is also used by the design in this thesis. It includes the directly adjacent cells; north, south, east and west; and the cell itself.

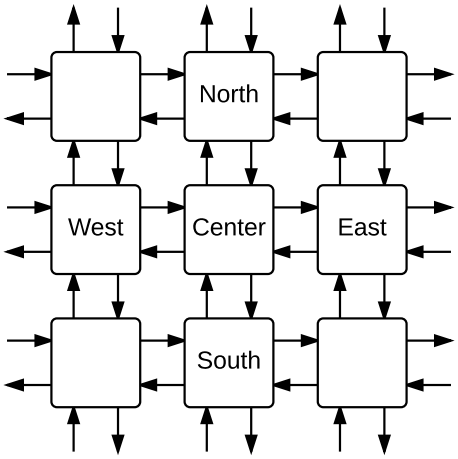


Figure 2.6: The von Neumann neighborhood in a 2D CA. The cell states are shared with all directly adjacent cells.

The von Neumann neighborhood is easily extendable to 3D by adding the cells directly above and below to the neighborhood. The extra dimension allow more complex signal routing since the signals can cross over or under each other, which hopefully allows more complex computation. It should however be possible to achieve the same complexity in fewer dimensions by increasing the neighborhood size, but that would likely require much more advanced transfer functions.

2.3.2 Complexity Classes

Wolfram observed that some CAs developed complex pattern while others decayed into uniformity or chaos. He therefore developed a set of four classes to group them based on their emergent behavior [37]:

- Class 1 – Uniform: The CA quickly results in a homogeneous state, regardless of initial condition. It lacks both the means to store data and perform computation.
- Class 2 – Repetitive: The CA develops both periodic and static data structures. However, after a number of cycles the states begin to repeat. This makes complex computation impossible.
- Class 3 – Chaotic: The CA is not held back by repetition as with class 2, but the patterns are so chaotic that it is unable to support data structures.
- Class 4 – Complex: The CA supports complex data structures and has no discernible repetition. It should be capable of universal computation.

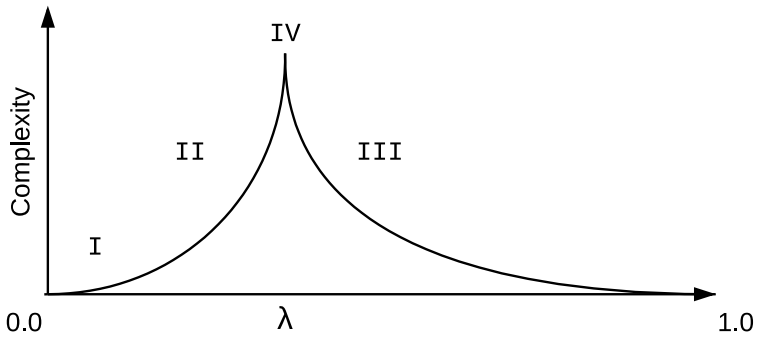


Figure 2.7: The location of Wolfram's complexity classes in λ space. (Adapted reprint from [23])

Langton attempted to connect Wolfram's classes to transfer function complexity [23]. For this he used a measure called λ, which essentially determines the inverse ratio of rules that cause a cell to transition into any given state. This means that all transitions are to a given state if $\lambda = 0.0$, while there are no transitions to any state if $\lambda = 1.0$. His findings, displayed in Figure 2.7, show that class 4 resides in a phase transition between the order of class 2 and the chaos of class 3. This is known as "The Edge of Chaos".

2.3.3 Evolution and Development in CAs

The problem with using EAs to program CAs is that the search spaces are vast, even for the simplest binary CAs ($2^{Neighborhood \cdot Cells}$ configurations[2]). However, by

adding development to the process, the search spaces can be dramatically reduced to manageable sizes while gaining other benefits such as scalability and robustness.

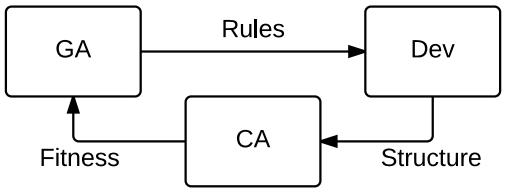


Figure 2.8: CA system with evolution and development.

The essence of the process is illustrated in Figure 2.8: The GA defines development rules. Development is then used to define a CA structure. Finally, the CA is executed to produce a fitness score that is fed back into the GA.

2.4 FPGA

A Field Programmable Gate Array (FPGA) is a type of reconfigurable hardware. It can implement any desired logical operation by configuring and connecting a number of lookup tables (LUTs) and flip-flop registers (FFs). FPGAs can also contain dedicated blocks for addition, multiplication, storage, and other functionality. The resources are grouped into configurable logic blocks (CLBs), which through a network of interconnects can be connected to each other or input/output (I/O) pins. An example of this structure is shown in Figure 2.9. Note that modern FPGAs consists of thousands of CLBs and hundreds of I/O pins [40].

FPGAs have been the subject of bio-inspired research due to their reconfigurability, and several researchers have been successful in using EAs to evolve working electronic circuits [22, 34]. However, the resulting circuits have often ended up using intrinsic properties of the silicon and been very sensitive to environmental changes. A problem with using modern FPGAs is that some configuration bit strings can destroy the FPGA by creating short-circuits [38, 42]. This means that the bit strings can not be used directly as the genotype without complicated tests to discard those that are dangerous.

The regular structure of an FPGA makes it well suited as the basis for implementing CAs however, especially 2D ones.

2.4.1 Sblock

The sblock was introduced as part of a new EA-friendly FPGA architecture in [17]. The architecture is a non-uniform CA with a von Neumann neighborhood, where

[2] The number of possible configurations for a relatively small 4x4x4 CA dwarfs the estimated number of particles in the universe.

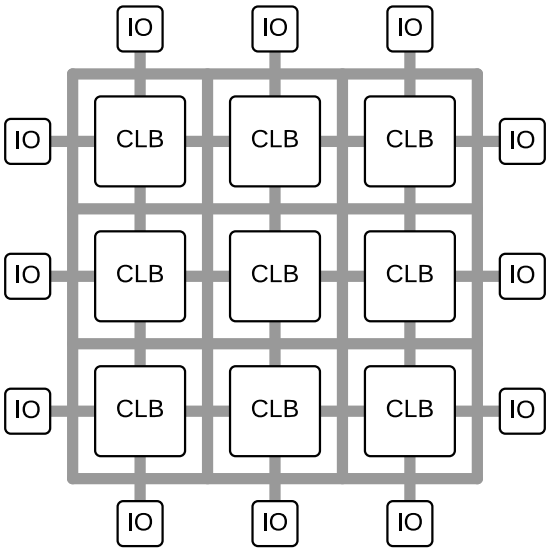


Figure 2.9: High-level block diagram of an FPGA. An array of CLBs and I/O pins are connected by a network of interconnects.

the transfer function of each cell is independently configurable at run-time. The cells, known as sblocks, are very simple structures; they consist of a configurable LUT and a FF, as shown in Figure 2.10.

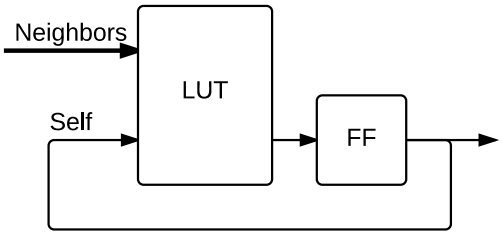


Figure 2.10: Detailed block diagram of an sblock. The LUT can be reconfigured on-the-fly to implement any logical function.

The greatest benefit of using sblocks in research using EAs is that there is no risk of damage or exploitation of intrinsic properties in the silicon. Additionally, the simple structure and hardwired signal routing allows for very efficient area usage. The likelihood of a mass-produced sblock-FPGA arriving on the market in the near future is slim. However, it is possible to implement it virtually within another FPGA, as is done in this thesis.

2.5 PCI Express

The PCI Express interface was designed to tackle the rising troubles with clocked parallel buses like PCI. The problem with such buses is that the clock speed can not be increased beyond a given threshold, as the slightly different lengths of the

wires causes data to arrive at slightly different times. Reducing the clock period to less than the variation in arrival time means the data will become corrupted. This problem is exacerbated with larger bus sizes.

PCI Express is therefore based on serial communication over differential pairs (lanes[3]) without the need for a reference clock [29]. This allows an extremely fast clock speed compared to a parallel bus, and much greater bandwidth in total. PCI Express consists of three layers; the physical layer, the data link layer and the transaction layer, structured as shown in Figure 2.11.

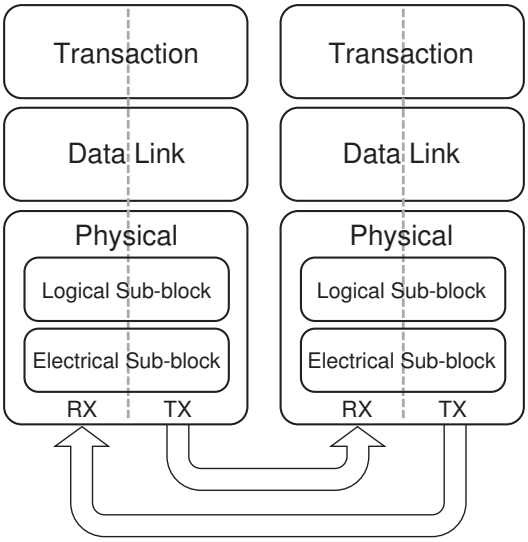


Figure 2.11: High-level diagram showing the layered structure of PCI Express. (Reprinted from [29])

The transaction layer's primary responsibility is the creation and parsing of transaction layer packets (TLPs). TLPs are used to trigger events or start various transactions, most commonly to initiate read and write requests[4]. Most requests entail the return of a completion TLP containing the requested data or other information. TLPs consists of multiple 32-bit double words (DW), where the first is a common header describing the type of packet.

The data link layer ensures integrity by adding error detection codes to outgoing TLPs and performing error detection and correction on incoming TLPs. It is also responsible for retransmission if corruption occurs.

The physical layer is responsible for serialization and deserialization of the data stream. Each byte is padded with two extra bits (8b/10b encoding) to allow clock recovery.

[3] PCI Express operates in full duplex mode, which means that each lane has an independent differential pair in each direction. 1, 2, 4, 8, 16 or 32 lanes are supported, but data is striped and thus still transmitted serially.

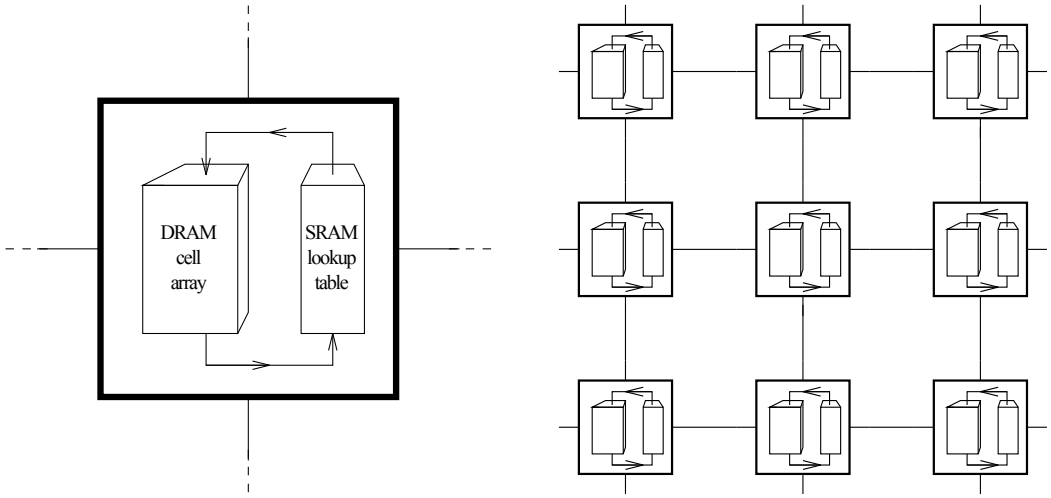
[4] Read and write requests are directed at one of up to six base address registers (BARs). They represent internal memory areas that can be anywhere from a few bytes to several gigabytes in size.

2.6 Related Work

This section describes work performed by others, which is related to that performed in this thesis but not directly applicable to it.

2.6.1 CAM-8

The Information Mechanics Group at MIT Laboratory for Computer Science has had a focus the question "How can computation and computers best be adapted to the constraints and opportunities afforded by microscopic physics?" This has led to more than a decade of study of CAs, as their fine-grained computation with local interconnectivity are particularly good candidates for micro-physical efficiency. To this end, they have created CA Machines (CAMs) that aim to use common computer parts in smart arrangements to provide CA computation performance akin to modern supercomputers. [26] describes their eighth iteration, known as the CAM-8.



(a) A single processing node (b) Spatial array of CAM-8 nodes

Figure 2.12: CAM-8 system diagram. (Reprinted from [26])

The CAM-8 implements a discrete, exhaustively enumerated, uniform, and synchronous 3D CA split over multiple nodes that run in parallel. Any number of nodes can be connected to form a CA of desired size, and an 8-module prototype showed to be on par with regular supercomputer simulations. As illustrated in Figure 2.12, each node contains DRAM that potentially holds millions of cells and an SRAM that holds the LUT for the current program. Within each node, cells are updated in sequence, making it a semi-parallel machine. This is a trade-off that compromises performance for the benefit of massively increased CA size and the ability to use common hardware. For 1-bit cell states, the prototype is capable of $3 \cdot 10^9$ cell updates per second and can fit up to $5 \cdot 10^8$ cells.

2.6.2 CAM-Brain Machine

In [5], de Garis et al. introduces the CAM-Brain Machine (CBM); an FPGA-based platform that implements a CA-based neural network that is evolved using a GA. It is part of de Garis' "Artificial Brain Project", which goal is to build an artificial brain with 1 billion neurons. The CBM is a stepping stone in the right direction, and allows the formation an artificial brain with nearly 75 million neurons in a CA of 843 million cells. It consists of up to 64640 modules, each containing 24x24x24 cells split over 72 FPGAs, of which 1152 cells can be neurons. Modules are evolved separately using the Collect and Distribute based neural network model from [13], and then connected by human design.

Brain building is still mostly in the "proof of concept" phase, so to attract attention to further research they designed a cute life-sized robot kitten that would be controlled by the CBM. The work to design and evolve it's brain architecture was expected to continue well into the 2000s, but was halted when the research institution went bankrupt in 2001 [15].

2.6.3 BioWall

The Logic Systems Laboratory at the Swiss Federal Institute of Technology have been working on bio-inspired hardware systems for many years, mainly focusing on the ontogenetic axis through embryonics. [33] describes a machine whose purpose is to convey the principles of embryonics to the public through visual and tactile interactions. The machine, depicted in Figure 2.13, is named BioWall due to its bio-inspired nature and sheer size at 5.3x0.6x0.5 meters^3.

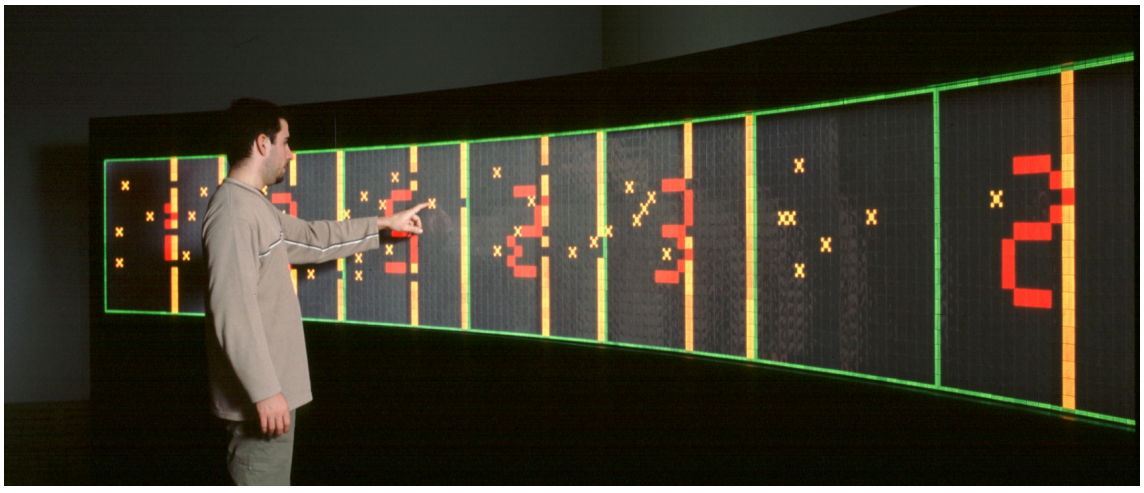


Figure 2.13: The BioWall, running BioWatch. (Reprinted from [28])

The BioWall is made up of 3200 identical units which can be seen an artificial molecules that can be combined to form cells. Each unit consists of an FPGA, 64

LEDs and a touch sensor, allowing users to interact with the surface by touching it and receiving immediate response from the LED display.

The main application for the screen is the BioWatch, an organism capable of counting hours, minutes and seconds. It is used to demonstrate the growth and self-repair capabilities of the system. 20 x 25 molecules/units are arranged into cells, each responsible for one digit. Users can then touch molecules to disable them, forcing the cell to reroute its functionality to a neighboring molecule.

The BioWall's cellular structure is well suited for many other bio-inspired applications as well. Examples are 2D CAs such as the Game of Life [11], self-replicating structures and artificial neural networks.

Chapter 3

Previous Work

This thesis continues to build on the Cellular Automata Research Platform (CARP), which is the result of three previous master theses at NTNU. The original implementation was made by Djupdal in 2003. It was then extended with a range of various output methods by Aamodt in 2005. Finally, it was further extended and optimized in expectation of new hardware by Støvneng in 2014.

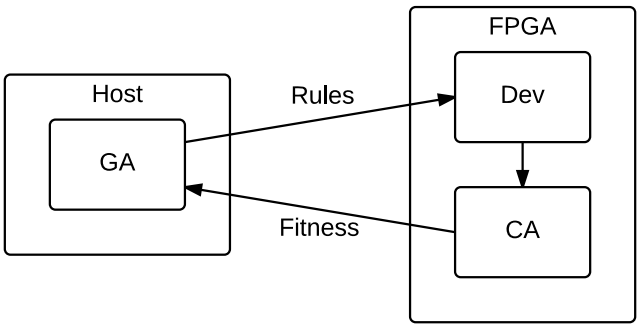


Figure 3.1: General system design.

The platform is more or less a "proof of principle" for how a CA can be combined with development and evolution to create a powerful bio-inspired system. The general system design, which is based on the setup in Section 2.3.3, is depicted in Figure 3.1: An FPGA implements the CA and development process while evolution is performed by a host computer.

3.1 Djupdal

In 2002, NTNU invested in a CompactPCI computer with a NallaTech BenERA FPGA board to be used for research within the field of evolutionary hardware. The task of developing a platform for the system, based on a matrix of sblocks, fell to Djupdal [7].

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

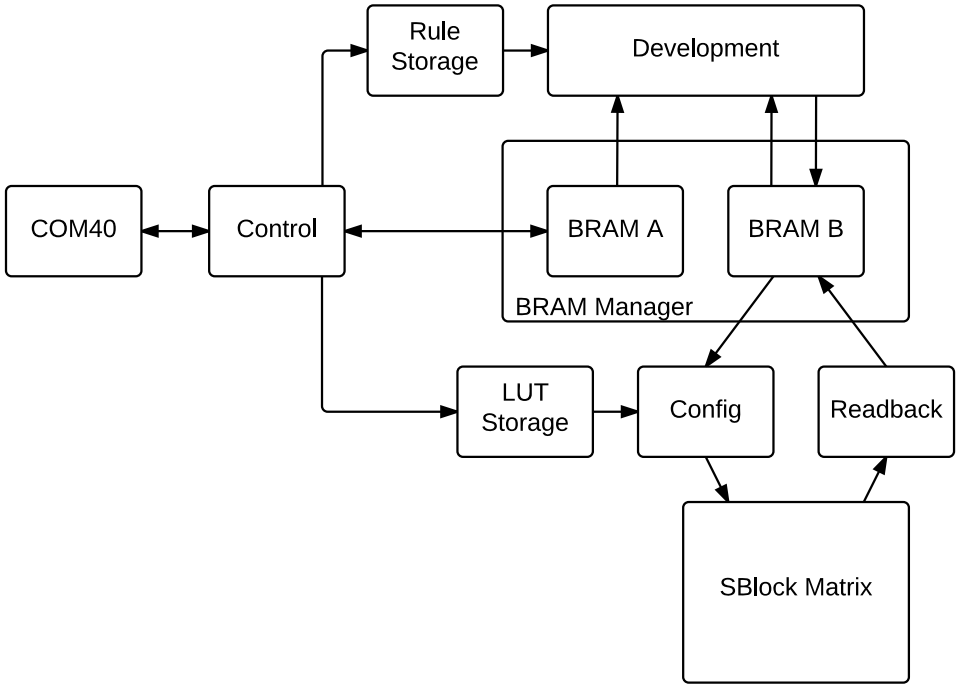


Figure 3.2: High-level block diagram of the hardware platform after Djupdal's original work.

The system is meant to be controlled by a computer running a genetic algorithm. A common flow of operation is to initialize the system with the genotype, develop it into its phenotype, step the SBM, and send the new states back to the computer. The computer then uses the newly received state data to calculate a fitness score.

The system is initialized by writing states and types to BRAM A, in addition to storing development rules and LUT conversion rules. Then development can be performed by reading cell types from BRAM A[1], testing development rules, and writing the (possibly changed) types to BRAM B. The development unit tests 8 rules on 2 cells each cycle in raster order. Optionally, the BRAMs can be logically swapped and further development performed. The SBM can then be configured by translating the types in BRAM B into LUT entries according to the LUT conversion rules, before being stepped for a desired amount of cycles. Afterwards, the new states in the SBM can be read back into BRAM B, swapped into BRAM A, and sent to the computer.

The design is split into two clock domains; the communication unit uses 40 MHz to be able to interface with PCI, while the rest uses 80 MHz for higher performance.

[1] After the first 8 rules have been tested on all cells, center cell types are read from BRAM B instead. This is needed to prevent the result of a rule in an earlier iteration from being deleted if no rules trigger in a later iteration.

3.2 Aamodt

There was one major bottleneck in the original design. To calculate the fitness of an individual, the state of each cell had to be transferred to the computer over the PCI interface. Having a dedicated hardware unit would greatly improve the performance. Additionally, it was desired to have more information about the development process. The task of realizing this fell to Aamodt [1].

An overview of the hardware platform with Aamodt's additions is shown in Figure 3.3. The additions consists of a Run-Step Function (RSF) that calculates the number of live cells, BRAM to store the numbers, a fitness function, and two information outputs from the development unit.

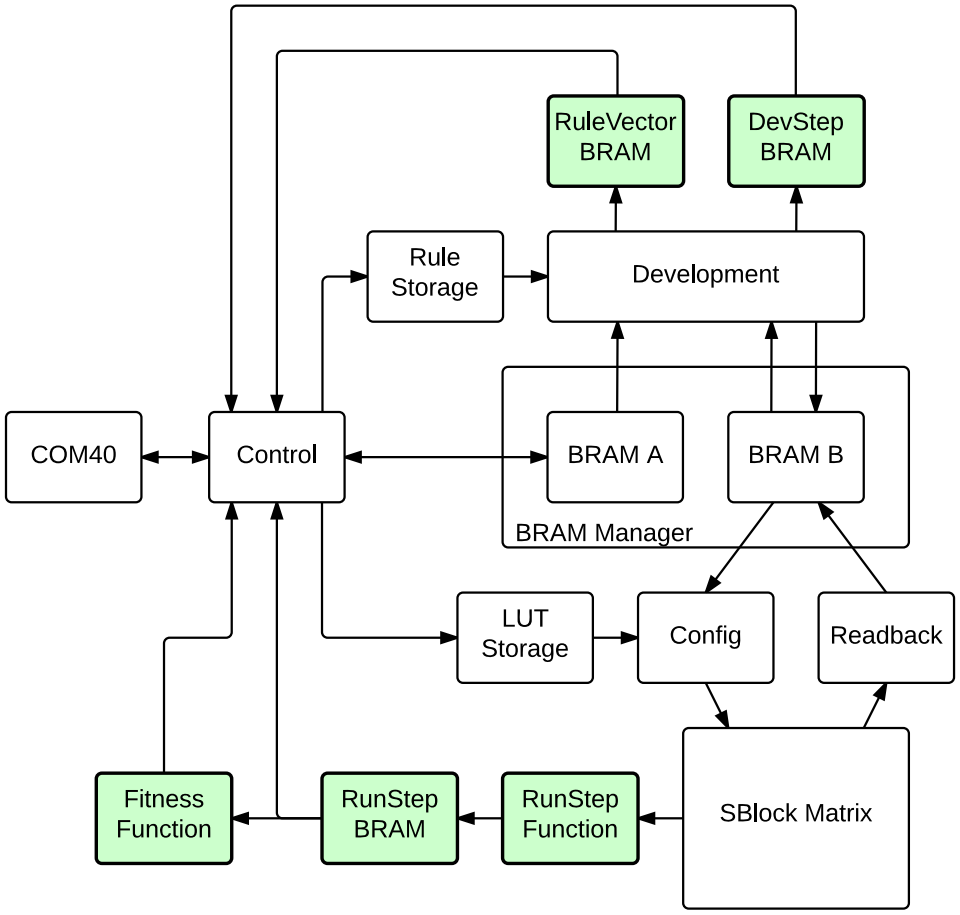


Figure 3.3: High-level block diagram of the hardware platform after Aamodt's work. Additions are highlighted in green.

The rule vector BRAM stores lists of which rules were triggered and not for the last 256 development phases. The lists are implemented as bit vectors where each bit represents the status of a rule for a single development phase. The development step BRAM is more detailed; it stores which rule was triggered for each cell. However, it only has storage space for one development phase.

The RSF calculates the number of live cells after each SBM step by using a large adder tree. The numbers are stored in Run-Step BRAM for later usage by the fitness function, which is replaceable.

3.3 Støvneng

In expectation of receiving new hardware with a larger and faster FPGA, there was a demand to optimize the platform by taking advantage of the increased resource pool. Extending the SMB into the 3D was also a lucrative thought, as it allows more complex signal pathways to form within it. A Discrete Fourier Transform (DFT) was also desired for interpretation of the RSF data; it should give very useful data according to Berg's research [2]. The task of realizing this was taken on by Støvneng [32].

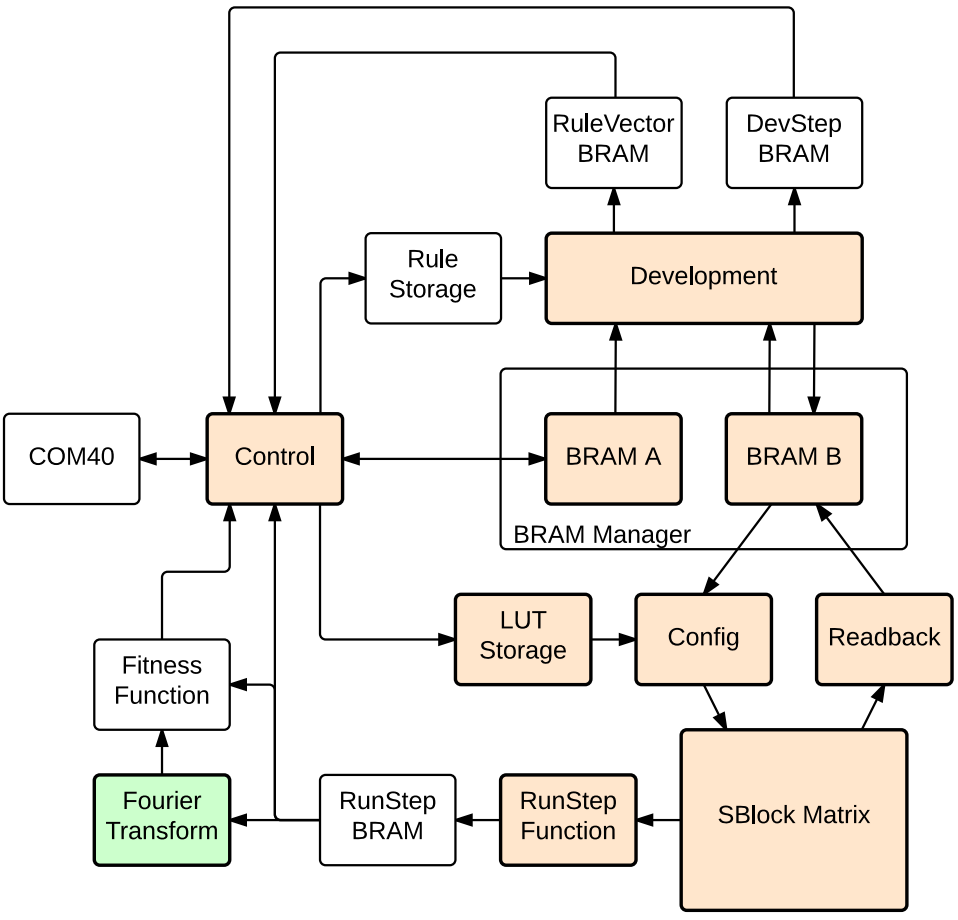


Figure 3.4: High-level block diagram of the hardware platform after Støvneng's work. Additions are highlighted in green, and optimizations and 3D modifications in orange.

An overview of the hardware platform with Støvneng's additions and optimizations

is shown in Figure 3.4. The only addition is the DFT, but nearly all units has been optimized, yielding a speedup of 4 for many operations.

Unfortunately, due to challenges with manufacturing, Støvneng was unable to get hold of the new hardware for the duration of his project. The system was therefore only verified in simulation, and the PCI communication unit was not upgraded for the PCI Express connection on the new board.

3.4 Issues

The original idea was to complete Støvneng's design by implementing the PCI Express module and driver in the specialization project leading up to this thesis, and then proceed to use the platform for research. However, after successful integration, verification of the platform revealed multiple serious issues, which are listed in Table 3.1. The full project report can be read in Appendix D.

Type	**Issue**
Severe	Many instructions fail or do not follow specification
Severe	The code is unnecessarily complex, making debugging nearly impossible
Minor	Outdated features are extensively used (tristate buffers and global resets)
Minor	Two different hardware designs and software APIs (2D and 3D)
Minor	DFT twiddle factors are generated by an external python program
Minor	It is impossible to retrieve the hardware parameters in the software API
Minor	There is almost no control flow available in the hardware design

Table 3.1: Issues with the existing design.

Starting from a blank slate would allow the platform to be made more modular, configurable, and maintainable. Also, it would be possible to unify the 2D and 3D designs and remove external dependencies. This led to the decision of rebuilding the entire platform from scratch in this thesis.

Chapter 4

Development Platform

The new hardware originally intended for Støvneng's thesis was still unavailable at the beginning of the specialization project. Therefore, the Spartan-6 SP605 Evaluation Platform was ordered as replacement hardware. It has an FPGA from the same product line, but substantially smaller in size. This meant that the architecture would be the same, but there would be around 70% less logic resources to work with. However, this should not be a significant problem, as the hardware design can be scaled down by reducing the size of the sblock matrix or the system's performance.

4.1 Spartan-6 SP605 Evaluation Platform

The Spartan-6 SP605 Evaluation Platform is essentially a circuit board with the Spartan-6 LX45T FPGA wired to every useful peripheral imaginable. It has connections for PCI Express[1], Ethernet, DVI, USB, flash card, JTAG, LEDs, switches, and more. However, the only peripherals utilized in this paper are PCI Express and JTAG. An overview of the system is shown in Figure 4.1.

The Spartan-6 is Xilinx' most cost-effective FPGA series. Its CLBs are divided into two independent slices, one of which is connected to a carry-chain. In addition to the standard slices which contain mainly LUTs and FFs, the Spartan-6 also contains a handful of specialized Digital Signal Processing (DSP) slices. These are optimized for pipelined wide multiplications and additions, which are often required in signal processing.

[1] Even though the PCI Express finger has lines for power, they are not connected on the SP605. This means an external power source has to be connected.

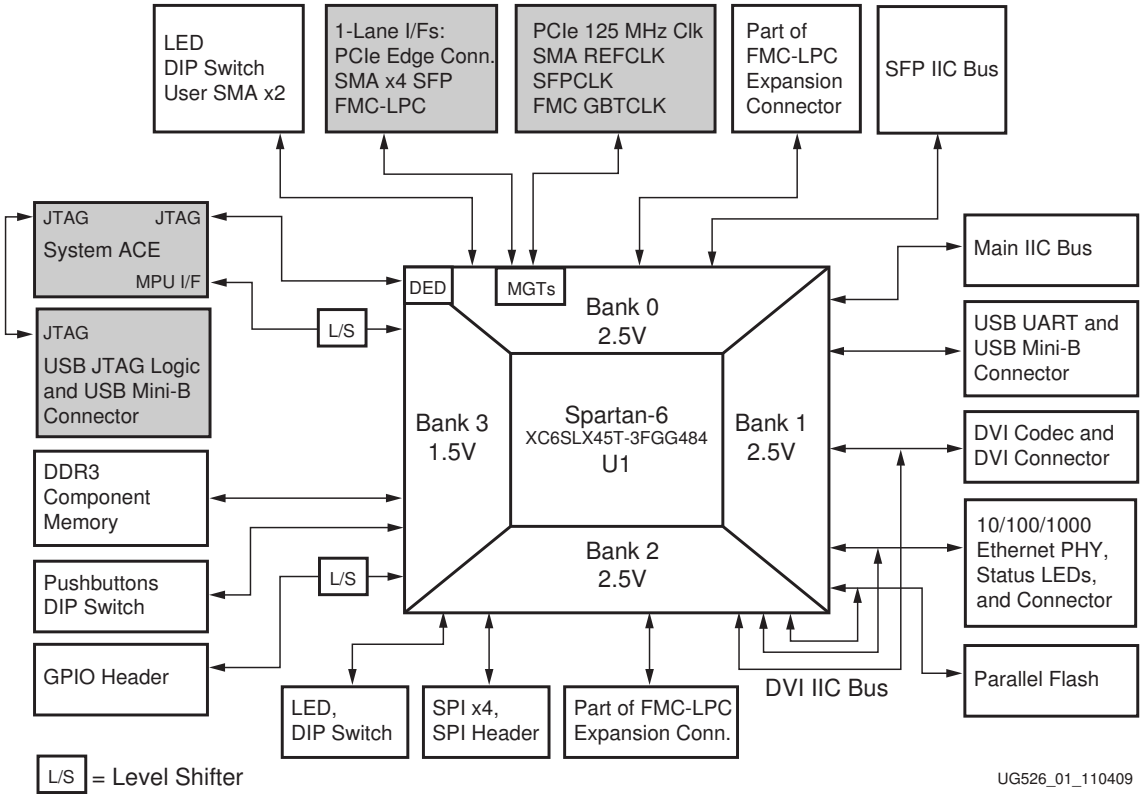


Figure 4.1: High-level block diagram of the SP605 and its peripherals. Peripherals utilized in this paper are highlighted in gray. (Modified reprint from [39])

4.2 Hardware Setup

Due to the experimental nature of building a new hardware design, two computers were used in this project, as shown in Figure 4.2. One is the main development workstation, used for coding and synthesis; it has a JTAG connection to the SP605 over USB, which allows it to upload new designs. The other is the host for the SP605, which is mounted in a PCI Express expansion slot.

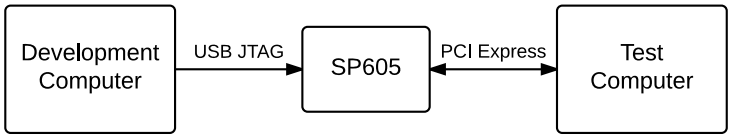


Figure 4.2: High-level block diagram of the hardware setup.

The setup allows a new design to be uploaded and tested on the SP605 without disrupting the workflow of the main workstation due to the power-cycle required to reset the PCI Express connection after a new design has been uploaded.

The switch and jumper configurations of the SP605 are set to factory defaults per [39], with the exception of SW1 which is set to 10 (M0=1 and M1=0).

4.3 Software Setup

The operating systems used on the computers have varied, without complications, between Linux Mint 16, Linux Mint 17 and Manjaro during the lifespan of the project. Linux Mint and its base Ubuntu are currently the two most popular Linux distributions [6]. Manjaro and its base Arch has smaller user bases, but are popular with enthusiasts. The procedures and software used in this thesis should therefore work without trouble on most modern Linux systems.

Xilinx ISE version 13.3 was used for hardware design and synthesis, while ISim was used for simulations. The third-party USB cable driver from [12] was used for JTAG, as explained in Section 7.3. The software API was compiled with both GCC version 4.8.2 and 4.9.2.

Chapter 5

Implementation

The new platform is nearly identical in overall structure to the previous, as shown in Figure 5.1, However, there are a lot of underlying changes to reduce complexity and improve reliability and scaling. A more detailed view of the system in shown in Figure 5.2.

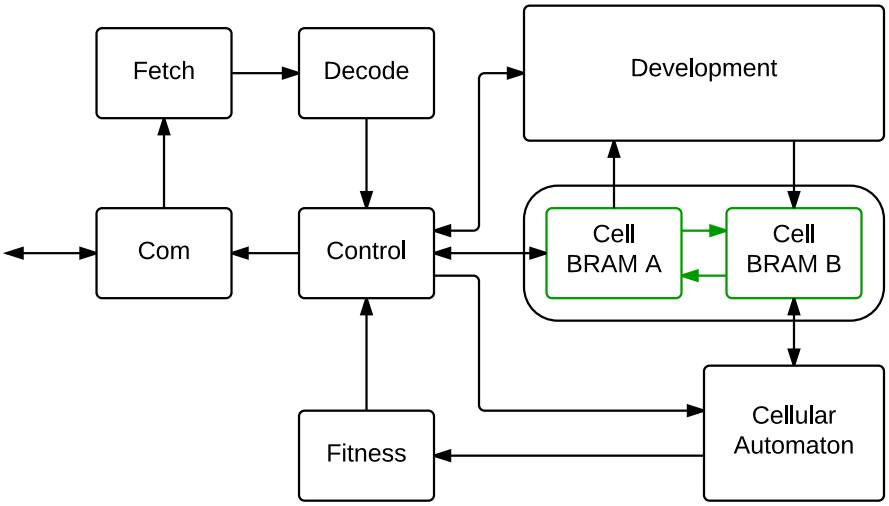


Figure 5.1: High-level block diagram of the new hardware platform.

Conceptually, the platform is a three-stage interlocked pipeline with the stages Fetch, Decode and Execute, where the Execute stage includes the Control, Development and CA modules. Only one module within each stage is activated at a time, and the interlocking allows each module to use multiple cycles and contain sub-pipelines without requiring complex hazard detection and evasion. Fitness is special in that it is not part of the main pipeline since it operates in a dataflow-like fashion.

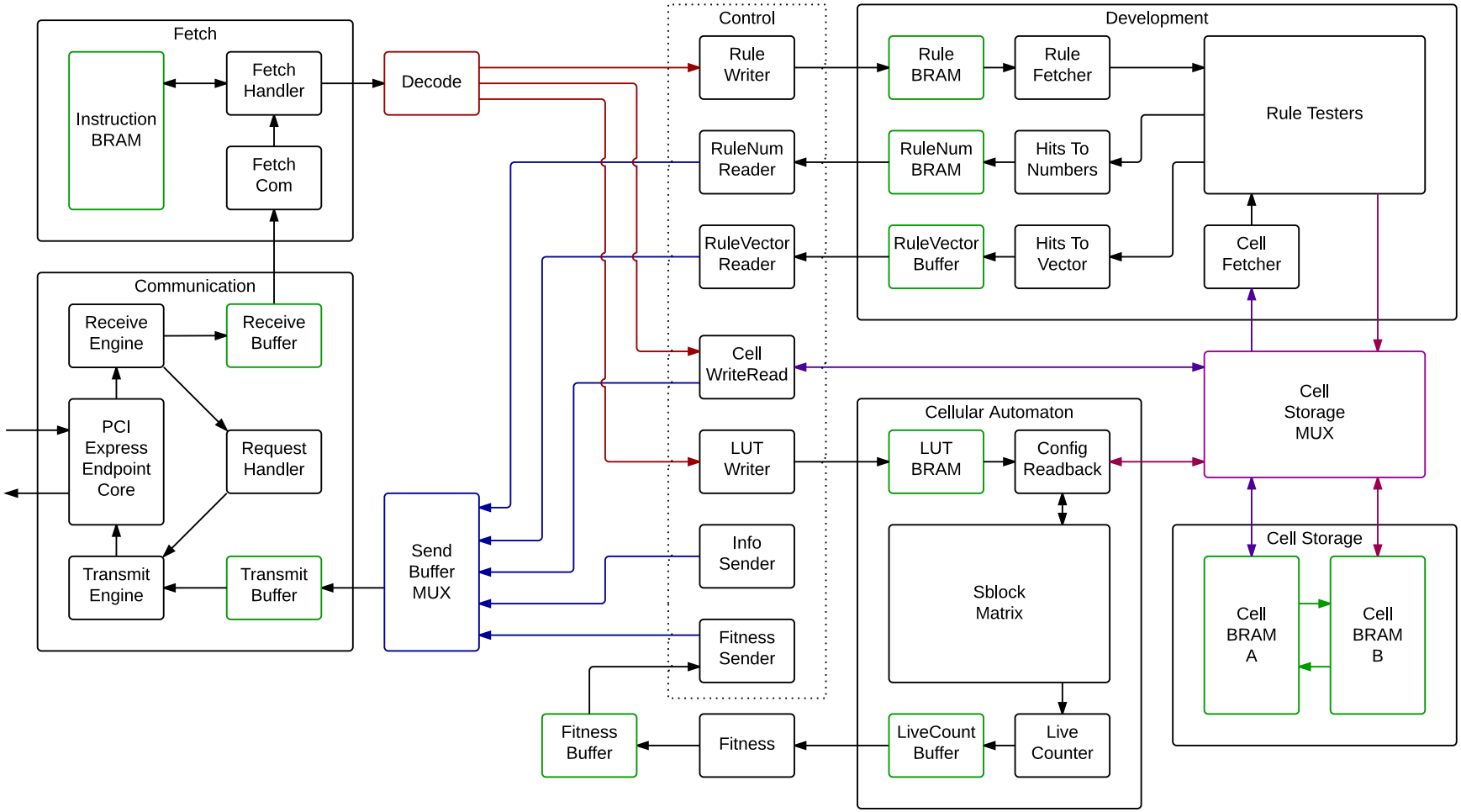


Figure 5.2: Detailed block diagram of the new hardware platform. Control is implemented as a group of modules, marked by a dotted border. Some signals are color-coded for increased readability. Signals from Decode are colored red, while those to the Send Buffer and Cell Storage Multiplexers are colored blue and purple respectively. Note the two different hues of purple for the different Cell BRAMs. Control signals are not shown.

5.1 General Concepts

Some concepts are used repeatedly throughout the design. The main ones are parameterization, pipelining, buffers and states machines. The first three are general and are detailed in the following subsections. The state machines vary from module to module and are therefore detailed where appropriate, but all are of Mealy design with clocked output.

5.1.1 Parameterization

Almost every part of the design is parameterized, usually with little restriction on the range of values. Where restrictions do apply, asserts have been placed in the code of the modules that do restrict them. These alert the user of incorrect values during synthesis, and pinpoints the code that must be changed in the event of an expansion. Keep in mind though that the ISA must likely be changed as well. The list of parameters is shown in Table 5.1.

Parameter	**Values**	**Notes**
Communication Buffer Size Lg	$[1, \infty]$	log_2 of buffer size
Communication Reverse Endian	$True, False$	Required for x86 systems
Program Counter Bits	$[1, 16]$	Restricted by ISA/Decode
Matrix Width	$[2, 256]$	Restricted by ISA/Decode
Matrix Height	$[2, 256]$	Restricted by ISA/Decode
Matrix Depth	$[1, 256]$	Restricted by ISA/Decode
Matrix Wrap	$True, False$	
Type Bits	$[1, 32]$	Restricted by ISA/Decode
State Bits	1	Restricted by Sblocks
Counter Amount	$[1, 256]$	Restricted by ISA/Fetch
Counter Bits	$[1, 32]$	Restricted by ISA/Fetch
LUT Configuration Bits	$1, 2, 4, 8$	Restricted by Sblocks
Rule Amount	$[1, \infty]$	
Rules Tested In Parallel	$[1, \infty]$	
Rule Vector Buffer Size	$[1, \infty]$	
Fitness Buffer Size	$[1, \infty]$	
Fitness Module Name	*Special*	Without “fitness_” prefix

Table 5.1: Parameters with supported values.

5.1.2 Pipelining

The new hardware design makes extensive use of pipelining, and since many stages use a variable amount of cycles for a variety of reasons, interlocking is used in nearly all pipelines. Interlocking is implemented with two signals connected to each stage: Run and Done. When a stage does not require further cycles to finish, it asserts its Done signal and then waits for the Run signal before continuing. The Run signals for all stages are asserted when all Done signals are asserted. Each stage then resets its Done signal and the process repeats. An example is shown in Figure 5.3.

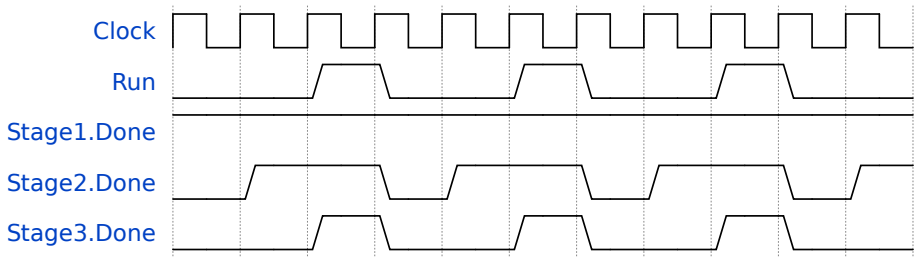


Figure 5.3: Wave diagram showing pipeline interlocking signals for a 3-stage pipeline where the stages complete in one, two and three cycles respectively.

Often, a multi-cycle stage only looks at its input in the first cycle to determine its execution path. This can be taken advantage of to reduce register usage by ruling that the data in the pipeline registers only have to be valid when Run is asserted. The stages can then write directly to their output registers, instead of caching partial output internally in extra registers. If a stage happens to require the output of the previous stage for multiple cycles however, input caching is needed. This causes no register usage reduction in the worst case, while the register usage is halved in the best case. The common case for this design is to only look at the input the first cycle for the most part, which means that is should provide a nice reduction.

5.1.3 Buffers

All buffers are implemented as first-in first-out (FIFO) queues using one BRAM and two counters. The counters determine the addresses that are written to and read from, and are incremented when the write or read signals are asserted. Figure 5.4 shows an example where a FIFO is used to buffer two words.

Notice how the read signal needs to be asserted before the clock tick when data is read to ensure correct consecutive reads. This is due to the operation of the BRAM, which updates its state at clock ticks. To have correct data available for a read in the following cycle, the address must therefore be updated before the clock tick (by asserting the read signal).

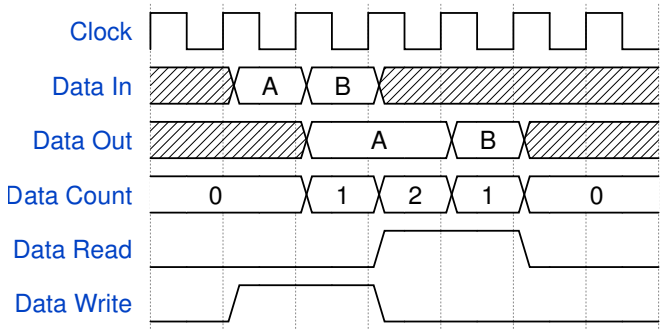


Figure 5.4: Wave diagram for a FIFO buffer, showing two consecutive writes immediately followed by two consecutive reads.

5.2 Communication

The Communication module is based on Xilinx' reference PCI Express programmed input/output design. It consists of the Xilinx PCI Express Endpoint Core, reception and transmission engines, data buffers, and a special request handler, as shown in Figure 5.5.

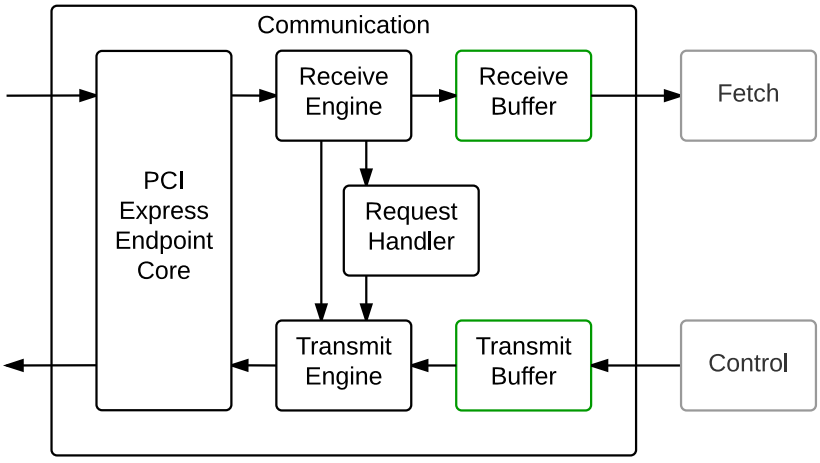


Figure 5.5: Detailed block diagram of the Communication module.

The Endpoint Core completely handles the physical and data link layers, plus all TLPs related to configuration and establishment of the PCI Express connection. The Reception Engine is responsible for parsing TLPs and either writing received data to the Reception Buffer or notifying the Transmission Engine of a read request. The Transmission Engine is responsible for building completer TLPs to respond to read requests, using data from the Transmission Buffer. The Request Handler listens to the read requests provided by the Reception Engine, and can override the Transmission Engine to respond to special requests.

5.2.1 PCI Express Endpoint Core

Several Spartan-6 FPGAs, including the one used in this project, contain a special-purpose hardware block for implementation of PCI Express. The block completely handles the physical and data link layers, with the transaction layer left for the user.

To make use of the block, Xilinx provides the Spartan-6 Integrated PCI Express Endpoint Core; version 2.3 was used in this project. This core additionally takes care of all TLPs related to configuration of the PCI Express connection. Other TLPs, such as read and write requests, are presented on an AXI4-Stream interface [41].

The endpoint core is configured with two memory regions, both 4 kB in size[1]. The first memory region (BAR0) is used for normal communication, while the second (BAR1) is used for special requests. The separation is mostly conceptual as both regions are treated as one data stream. The difference is that the special request handler kicks in for read requests to BAR1.

5.2.2 Reception Engine

The Reception Engine is implemented as a simple state machine, as shown in Figure 5.6.

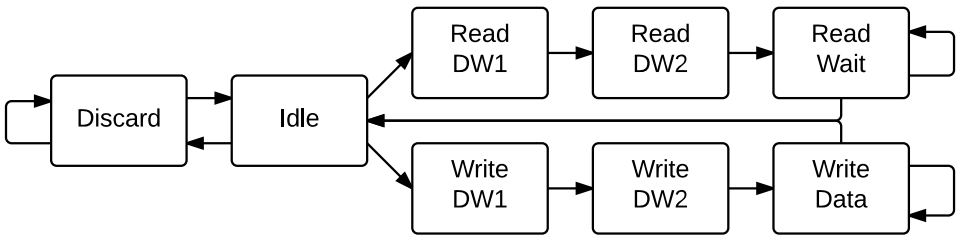


Figure 5.6: State machine for the Reception Engine.

Until the endpoint core presents valid data, the state machine remains in Idle. When it does, the data is stored, and the TLP type is checked. If it is a read or write request, the state machine continues down the corresponding path, otherwise the remaining data is discarded. The remaining portion of the TLP headers are then parsed in the DW1 and DW2 states. For read requests, the state machine waits in ReadWait until the Transmission Engine is ready to accept a new read request, and then proceeds to Idle. For write requests, the state machine stays in WriteData, where one DW of data is written to the Reception Buffer each cycle, for the length of the packet, and then proceeds to Idle.

[1] The smallest memory region that can be memory-mapped is one page. The default page size in Linux is 4 kB.

5.2.3 Transmission Engine

The Transmission Engine is implemented as a simple state machine, as shown in Figure 5.7.

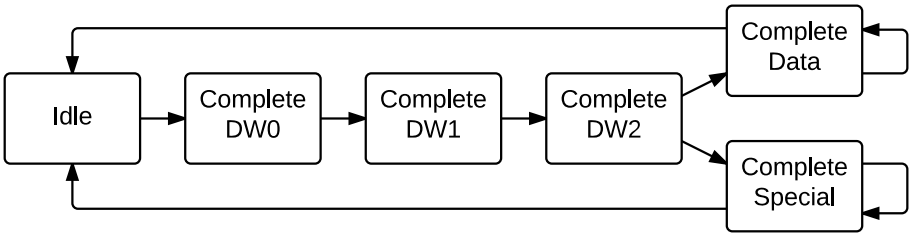


Figure 5.7: State machine for the Transmission Engine.

Until the Reception Engine signals a read request, the state machine remains in Idle. When a read request is signaled, the state machine begins to traverse the DW path. The DW0, DW1 and DW2 states each transmit one DW of the completer TLP header. Then if the special request signal is set, it proceeds to CompleteSpecial, where it transmits data presented by the Request Handler. Otherwise, it proceeds to CompleteData where it transmits one DW of data from the Transmission Buffer each cycle. When the requested number of DWs has been transmitted, it proceeds back to Idle.

5.2.4 Request Handler

The Request Handler continually listens to the read requests presented by the Reception Engine. If the request is targeting the primary memory area (BAR 0), it is a normal read request and the Transmission Engine is allowed to proceed as usual. Otherwise, it is a special request and the Transmission Engine is overridden.

The kind of special request is determined by the address of the read request, and handled thereafter. There are currently four special requests implemented, as shown in Table 5.2.

Address	**Request**
0x00	Get Transmission Buffer data count
0x01	Get Transmission Buffer available space
0x02	Get Reception Buffer data count
0x03	Get Reception Buffer available space

Table 5.2: Special requests.

Note that each of the implemented special requests assumes a read request length of one DW. If the request has a greater length, the returned data is simply repeated to fill the packet.

5.3 Fetch

The Fetch module is responsible for retrieving the next instruction that should be decoded and then executed. It also handles all control flow. It is implemented as a two-stage interlocked pipeline consisting of a Fetch Communication module and a Fetch Handler module connected to an Instruction BRAM. This is shown in Figure 5.8.

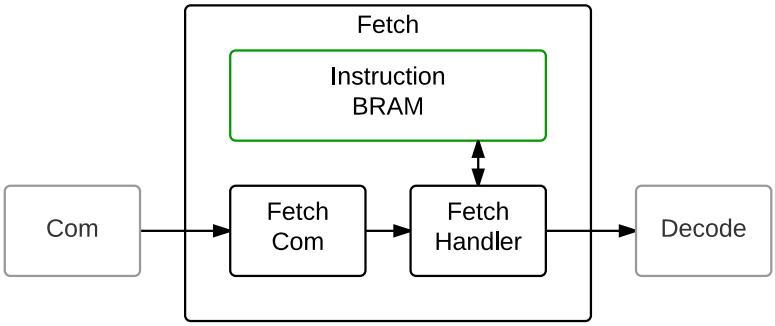


Figure 5.8: Detailed block diagram of the Fetch module.

Fetch Communication is responsible for converting data from Communication into instructions while the Fetch Handler takes care of control flow and the instruction memory. Both stages are implemented as state machines, making interlocking necessary.

5.3.1 Fetch Communication

While instructions are 256-bit, the communication interface is only 32-bit. This means that the host system has to split each instruction into multiple 32-bit pieces. As detailed in Appendix C, many instructions make use of less than 256 bits. In fact, most instructions fit within the first 32 bits. Sending all 256 bits for each instruction is therefore a bit excessive. To optimize communication, the first 32-bit piece of each instruction has a field declaring the amount of following pieces required for reassembly.

The job of the Fetch Communication module is to combine all the pieces back into full 256-bit instructions. It starts by reading the first 32-bit piece and setting all other bits to zero. Then, the 3-bit instruction length field is analysed to determine how many further pieces are part of the same instruction. The remaining pieces are then incorporated into the instruction, before it is passed on to the Fetch Handler.

5.3.2 Fetch Handler

The Fetch Handler has three modes of operation: FetchCom, FetchMem and StoreMem. They are implemented as states in a state machine, as shown in Figure 5.9.

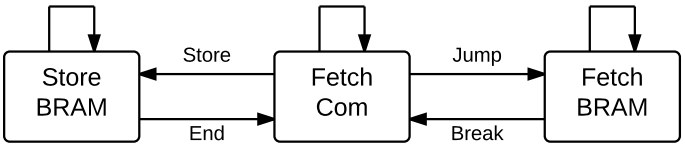


Figure 5.9: State machine for the Fetch Handler.

In FetchCom mode, instructions are fetched from Communication and sent to Decode. Since a variable-length format is used, this may take multiple cycles. To make sure that instructions do not get "stuck" in the pipeline due to no further instructions arriving at the communication interface, NOPs are sent when Fetch Communication is busy. When encountering a Store instruction, it enters StoreMem mode and for a Jump instructions it enters FetchMem mode.

In FetchMem mode, instructions are fetched from Instruction BRAM and sent to Decode. The first Instruction BRAM address is specified by the Jump instruction, and then it is incremented by one after each instruction. When encountering a Break instruction, it enters FetchCom mode. As a safety precaution to prevent some potential lock-ups, FetchCom mode is also entered if the program counter overflows.

In StoreMem mode, instructions are fetched from Communication and stored in Instruction BRAM. The first Instruction BRAM address is specified by the Store instruction, and then it is incremented by one after each instruction. Instructions are stored in full 256-bit format. When encountering an End instruction, it enters FetchCom mode.

Control flow is implemented by having N M-bit general counters and a JumpEqual instruction. The counters can be incremented or reset using special instructions. The JumpEqual instruction is treated as a Jump instruction when the specified counter matches the specified value, but is otherwise discarded.

5.4 Decode

Decode is responsible for parsing instructions, setting up control signals and passing instruction parameters to activated modules. It is a very simple module, being essentially a giant switch statement with a case for each instruction.

Control signals are sent to all top-level modules except Communication and Fetch which operate earlier in the pipeline, and Fitness which operates in a dataflow-like manner. By default, all modules are given a no-operation signal and multiplexers stay unchanged. Then, depending on the instruction's operation code, the control signal for the appropriate module is set, parameters (if any) are extracted and passed on to the module, and multiplexers are changed if needed.

Each instruction will cause exactly one module to be activated in the Execute stage.

This is to keep the design clean and to reduce inter-module dependencies.

5.5 Control

Control is a group of modules, shown in Figure 5.10. Together, the modules control all inputs and outputs for the Cell Storage, CA and Development modules. Each module is designed to do one specific task and be independent of any other modules. This means that modules are mostly very simple and that it requires a low amount of effort to add new modules or to modify existing.

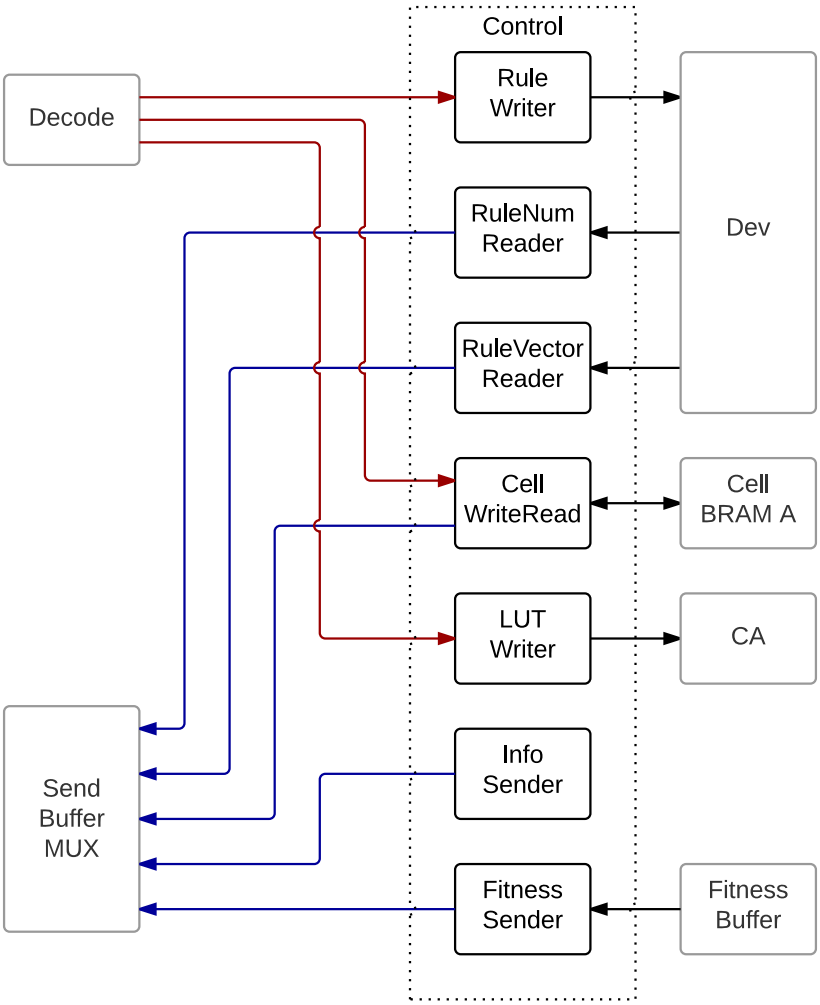


Figure 5.10: Detailed block diagram of the Control modules. Red signals are inputs while blue are outputs. Control signals are not shown.

Following are detailed descriptions of the different modules, in order of increasing complexity.

5.5.1 Rule Writer

The purpose of the Rule Writer is to store new rules to the Rule BRAM within the Development module. It is, along with the LUT Writer, the simplest control module. When activated, it stores one rule to a specified index of the Rule BRAM. The indexes double as priority for the rules, with higher indexes having higher priority.

As explained in Section 5.7, index zero is reserved for representing that no rules have triggered. Writing a rule to it has no effect as the Rule Testers are reset instead of testing the rule during development.

5.5.2 LUT Writer

The purpose of the LUT Writer is to store new LUTs to the LUT BRAM within the CA module. It is, along with the Rule Writer, the simplest control module. When activated, it stores one LUT to a specified index of the LUT BRAM. The index is equivalent to the cell type that should be given that LUT during CA configuration.

5.5.3 Information Sender

Nearly all parts of the system are parameterized. The previous practice of manually ensuring that the parameters of both the design and API were in sync was both tiresome and prone to error. Therefore, the Information Sender provides a means for the API to automatically query these parameters.

When activated, it puts all parameters that might be useful into the Transmission Buffer. This includes information about the CA such as the size, whether wrapping is enabled, and number of bits per state and type; information about counters available for control flow; maximum number of rules; and information about the fitness modules, such as the type and output size.

5.5.4 Fitness Sender

The Fitness Sender is responsible for sending the output of the Fitness module to the host. This is a simple matter of moving data from the Fitness Buffer to the Transmission Buffer when data is available in the first and there is space in the second. The number of words that is transferred per activation is declared by the Fitness module (see Section 5.9). Keep in mind that the machine will go into a deadlock if insufficient data is produced for the Fitness Buffer.

5.5.5 Rule Vector Reader

The Rule Vector Reader is tasked with reading one or more rule vectors created by the Development module and sending them to the host. Since rule vectors can be of any length, they are each split over multiple words, starting with the lowest indexes. The final word of each vector is padded with zeroes. For more information on rule vectors, see Section 5.7. Keep in mind that the machine will go into a deadlock if insufficient data is produced for the Rule Vector Buffer.

5.5.6 Rule Numbers Reader

The Rule Numbers Reader is tasked with reading each cell's most recently activated development rule and then sending them to the host. The rule numbers are stored in the Rule Numbers BRAM of the Development module and are scanned in raster order[2]. Each word is fitted with as many rule numbers as possible without splitting them over multiple words or containing numbers from different rows. Any remaining space is filled with zeroes.

5.5.7 Cell Writer Reader

The purpose of the Cell Writer Reader is to perform read and write operations against Cell BRAM A, causing it to be the system's main input/output channel. It is possible to write states and types to either a single cell, a row of cells or all cells, while it it is possible to read from a single cell or all cells.

It is easily the most complex control module, due to the intricate operations required to change only selected values in BRAM rows. Additionally, it must convert variable-width types and states into fixed-width words in an efficient manner when reading.

The module consists of Combiners which are used to combine new data with existing data from the Cell BRAM, Repeaters to simplify the process of filling the entire Cell BRAM with a given state and type, Shifters used to select output, and a state machine which controls everything. Figure 5.11 shows how the components are connected.

Combiner

The Combiner is a combinatorial unit that combines two signals of different lengths by replacing one part of the long signal with the short signal. This is implemented using a shifter and a mask that is the size of the short signal. First, the short input and mask is shifted into the desired position. Then, the long signal is AND-ed with

[2] In raster scanning, two-dimensional data is read line-by-line, least significant to most significant. This is extendable to 3D; increment X first, then Y, then Z.

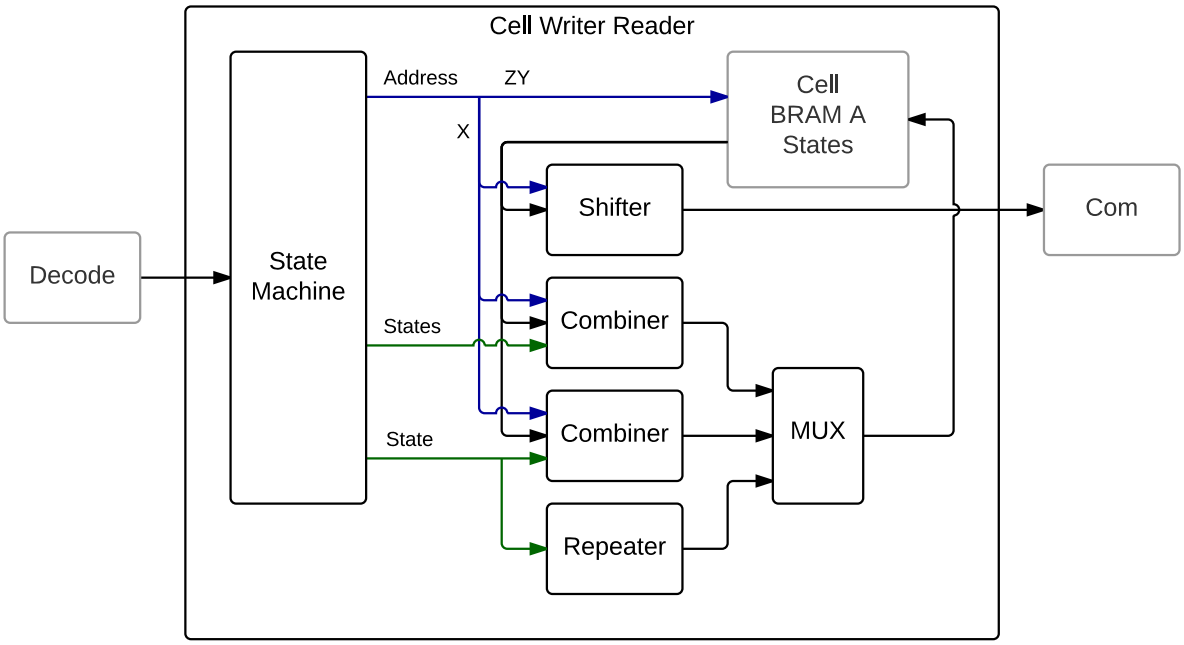


Figure 5.11: Detailed block diagram of the Cell Writer Reader. Only the state part is shown to reduce complexity; the type part is identical. Cell BRAM A is drawn inside the module for completeness. Control signals are not shown.

the inverted mask and OR-ed with the short signal, producing the combined signal. The process is illustrated in Figure 5.12.

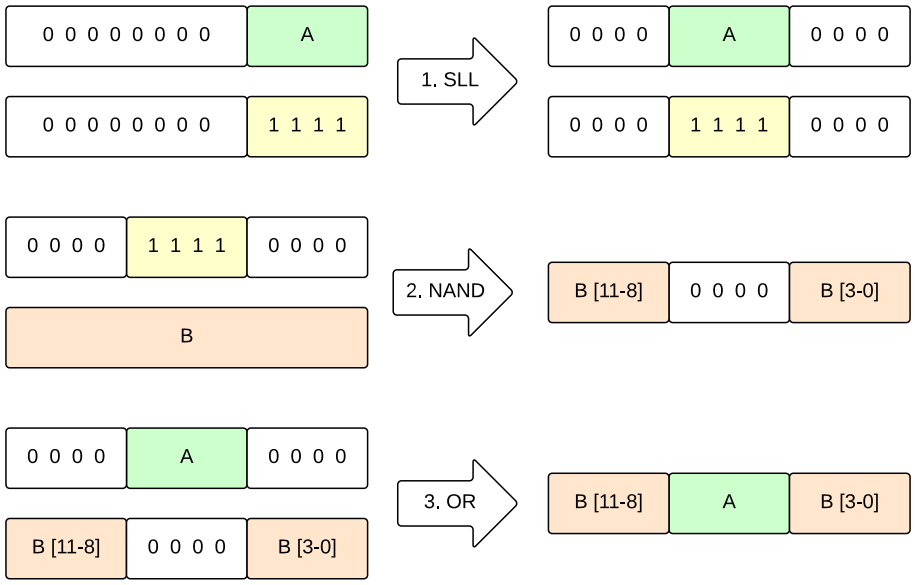


Figure 5.12: The three operations that power the combiner. A is the short input, highlighted in green. B is the long input, highlighted in orange. The mask is highlighted in yellow.

State Machine

The state machine consists of seven states, shown in Figure 5.13. When an operation is received, the BRAM address is set and it transitions from idle to the corresponding state. Some states are dual-purpose due to the similarity of the operations, while others are not. Coincidentally, the dual-purpose states complete in one cycle while the others require multiple[3].

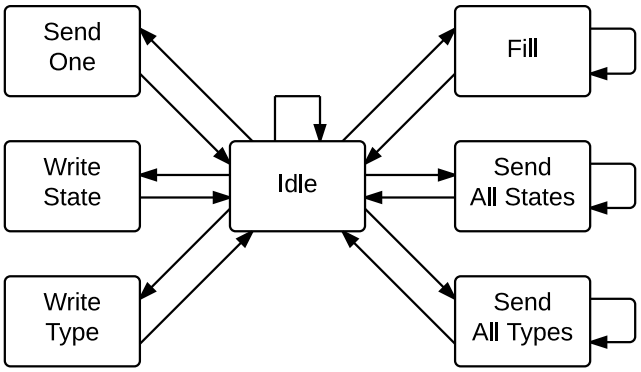


Figure 5.13: Cell Writer Reader state machine

The dual-purpose states are as follows: Send One can send either a state or a type, Write State can write either one or a row of states, and Write Type can write either one or a row of types. The remaining states are as follows: Fill writes the same state and type to all cells, Send All States reads all states in raster order, and Send All Types reads all types in raster order. The output formatting of the Send All states are equal to that of the Rule Numbers Reader, detailed in Section 5.5.6.

5.6 Cell Storage

The Cell Storage serves as the location for exchange of cell data between the CA, Development and host. It contains two separate storage areas, the contents of which can be swapped. Each storage area can host a full matrix of cell states and types, and allows one row of both to be read each cycle. The main reason two storage areas are needed is the Development module. It requires a place to store its output without affecting its input during the development process. More on this in Section 5.7.

The module is implemented as two dual-port BRAMs, one for states and one for types, each sized to twice the size of the matrix. To create two separate storage areas (A and B) with both states and types, the address of the first port is prefixed with 0 and the second with 1. The contents of the storage areas can then be made to appear swapped by simply inverting the prefix bits.

[3] Technically, the Send One state does not necessarily complete in one cycle since it will wait until there is available space in the Transmission Buffer. However, one cycle is the common case.

To service all required components, the Cell Storage is connected via a multiplexer. It has two modes; normal and development. In normal mode, storage A is connected to the Cell Writer Reader and storage B to the CA. In development mode, both storage areas are connected to the Development module.

5.7 Development

The Development module is responsible for providing the ontogenetic aspect of the system by allowing cells to be changed based on user-supplied development rules that are described in Appendix C. It uses Cell BRAM A as input and outputs the modified cells to Cell BRAM B.

The module is implemented as a two-stage interlocked pipeline controlled by a state machine. Stage one contains the Cell Fetcher, which retrieves cell neighborhoods from Cell BRAM A, and stage two contains a four-stage pipeline that tests development rules against the cell neighborhoods. This is illustrated in Figure 5.14.

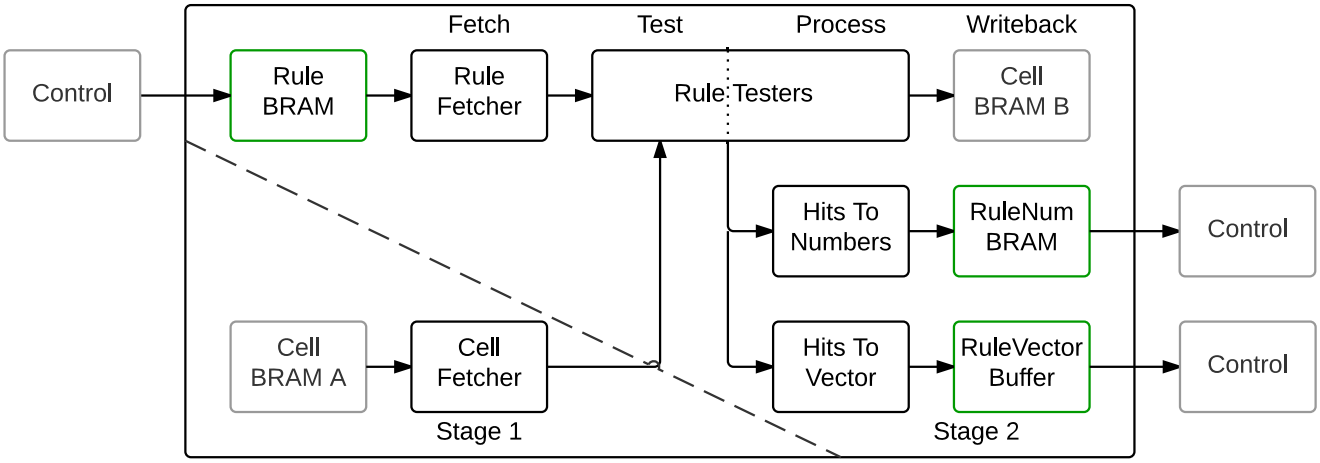


Figure 5.14: Detailed block diagram of the Development module. The two main pipeline stages are separated by a dashed line, while the sub-stages of the pipeline within the second main stage are marked at the top. The cell BRAMs are drawn inside the module for pipeline completeness. Control signals are not shown.

The state machine, shown in Figure 5.16, ensures proper timing of the complex pipeline. It is responsible for setting input and output addresses, activating pipeline stages and setting write signals. A complete timing diagram can be seen in Figure 5.15.

5.7.1 Cell Fetcher

The Cell Fetcher reads the cell neighborhoods for one row of cells from Cell BRAM A per run. It is implemented as two state machines, one which sets the BRAM

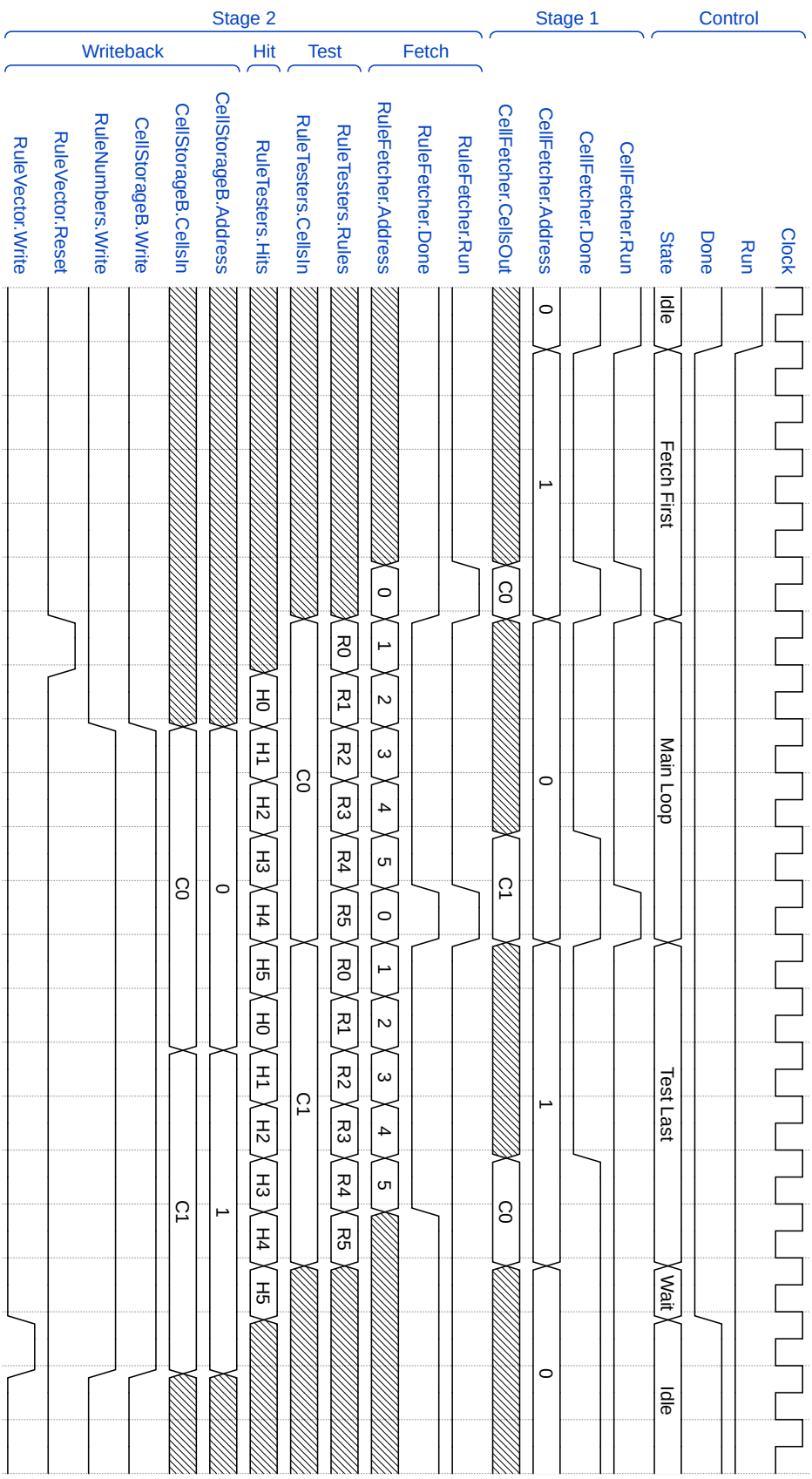


Figure 5.15: Wave diagram for the Development module, showing the development process for an Nx2 matrix with five active rules.

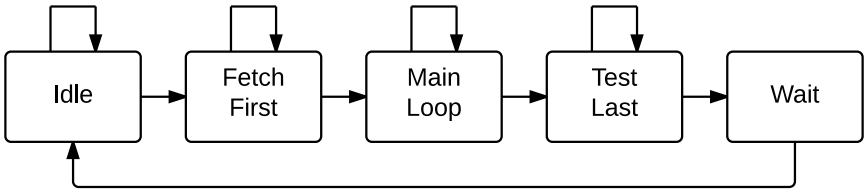


Figure 5.16: State machine controlling the Development module.

address and one which retrieves the output. The first can be seen in Figure 5.17. The other is equivalent to the first except for being delayed two states, thus having Wait 1 as initial state instead of Fetch Center.

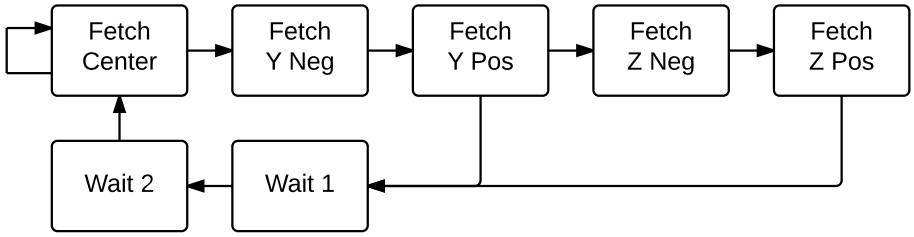


Figure 5.17: State machine for the Cell Fetcher.

By reading an entire row at a time, the neighbors along the X axis are fetched "for free", while the other axes require two extra reads each. This lands the run time on 5 cycles for 2D matrices and 7 cycles for 3D when including BRAM latency. The Cell Fetcher is therefore only a limiting factor when there are very few active development rules.

By default, neighbors which would be outside the matrix are treated as having zero for both state and type. However, it is possible to enable matrix wrapping to instead use the neighbors on the opposite side of the matrix.

5.7.2 Rule Fetcher

The Rule Fetcher is responsible for fetching development rules from the 1-in N-out Rule BRAM (one write port and N read ports) and passing them to the Rule Testers. It takes into account the specified number of active rules to speed up the development process by only fetching those. It is possible to further improve performance by setting that multiple rules should be fetched and tested at the same time. In that case, those with indexes higher than the number of active rules within the last rule batch are replaced by rules with no effect.

5.7.3 Rule Testers

The Rule Testers are responsible for testing development rules received from the Rule Fetcher against cell neighborhoods received from the Cell Fetcher. They are unique in that they produce output after both one and two cycles. Information about rule activations ("hits") are passed on to the Hit Processors after one cycle, and the result of the rule application is passed on to Cell BRAM B after two cycles. Rule hits undo any effects of previous hits so that it is impossible for two rules to have simultaneous partial effects, in cases where one modifies only the state and the other only the type.

There is a special case for rule zero. It is used as an internal reset by forcing a hit and setting the output cell to the input cell. This implementation was chosen because of its simplicity and the ability to use zero to mean "unchanged" in the Hit Processors. It also makes it possible to have zero active rules.

5.7.4 Hit Processors

There are two modules dedicated to providing information about the development procedure, Hits To Vector and Hits To Numbers. The input for both are hits passed on from the Rule Testers.

Hits To Vector stores a vector, where each bit signifies whether the rule of that index was triggered or not, to the Rule Vector Buffer after each development phase. Rules that have triggered but is later overridden by a rule with higher priority is still marked as having been triggered in the vector. If the buffer is full, it is reset instead of waiting, to allow programs to disregard the data without causing deadlocks.

Hits To Numbers stores the index of the last triggered rule for each cell to the Rule Numbers BRAM. Since rule zero is always marked as a hit, the BRAM is reset to zeroes at the beginning of each development phase. Therefore, only hits from the most recent phase are stored.

5.8 Cellular Automaton

The CA module is the centerpiece of the system. It is what contains the sblock matrix and the control logic to service it. The module is responsible for configuring the sblock matrix with data from Cell BRAM B, step the sblocks, store the number of live cells after each step, and write new states back to Cell BRAM B.

The implementation can be seen in Figure 5.18. It consists of a state machine, the Sblock Matrix and a Live Counter, in addition to storage for LUTs and a buffer for the Live Counter.

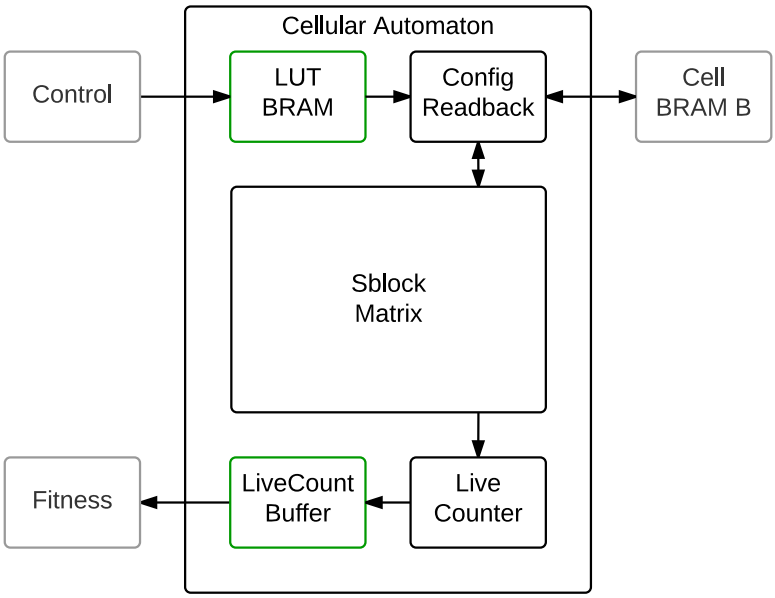


Figure 5.18: Detailed block diagram of the CA module. Config Readback is a symbolic module for the majority of the state machine.

5.8.1 State Machine

The state machine in Figure 5.19 is what powers this module. It is what controls both configuration, readback and stepping of the Sblock Matrix. Additionally, it sets write signals for the Live Count buffer.

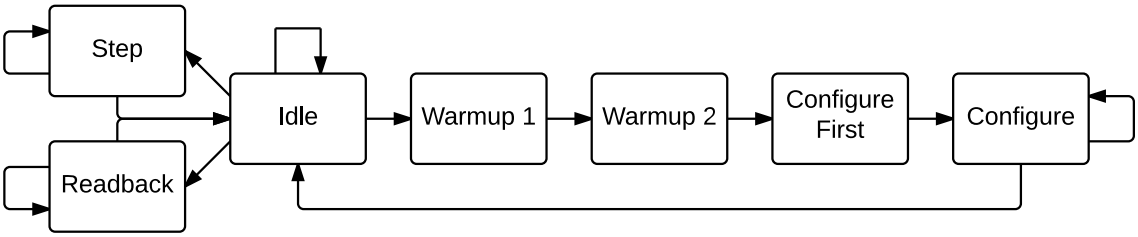


Figure 5.19: State machine controlling the CA module.

Configuration is the process of taking cells from Cell BRAM B, converting them into LUTs and programming the sblocks. It operates on one row of cells/sblocks at a time. First, the cells are read from Cell BRAM B and the types of used as addresses for the 1-in N-out LUT BRAM to find the corresponding LUTs. Then, the LUTs are loaded into shift registers and transferred to the sblocks a few bits at a time, the speed of which is configurable. Finally, the states/FFs of the sblocks are set to the states of the cells.

Readback is the process of extracting the cell states from the Sblock Matrix and storing them back in Cell BRAM B. This operation also works on one row at a time. However, it is much simpler and faster than configuration since it simply directs the output of the sblocks to the Cell BRAM and sets the correct row.

Stepping is the process of telling the Sblock Matrix to update the state of each of its sblocks. The new state is determined by the using the neighbor states as the input to the LUT, according to the format detailed in Appendix C. Since it is common to step hundreds of times in sequence, the step instruction has a parameter for the number of steps, up to a maximum of 65535 (a 16-bit number).

5.8.2 Sblock Matrix

The Sblock Matrix essentially contains enormous amounts of the sblock that are described in Section 2.4.1. The only difference is that the sblocks used here have support for configuring multiple bits of the LUT each cycle. However, to use the dedicated shift registers as configurable LUTs, there are some restrictions on the number of bits. Firstly, it can only be powers of two; secondly, it can maximally be 2 for 2D matrices and 8 for 3D. Keep in mind that each bit adds one extra signal for each sblock, which can accumulate to a significant amount of required routing resources.

By default, neighbors which would be outside the matrix are treated as having zero for both state and type. However, it is possible to enable matrix wrapping to instead use the neighbors on the opposite side of the matrix. This option lets the user decide if programs should be able to exploit the matrix size when for example creating oscillators.

5.8.3 Live Counter

The Live Counter is essentially a giant adder tree that is connected to the output of each sblock. It calculates the total number of live cells after each CA step and stores them in the Live Count Buffer. Due to the massive amount of sblocks, the calculation pipelined over many cycles, the exact number of which is dependant on the number of sblocks. However, the throughput remains at one total per cycle.

If the Live Count Buffer happens to be full, it is reset instead of waiting, to allow programs to disregard the data without causing deadlocks. This will likely corrupt all fitness evaluation until all buffers are properly reset however.

5.9 Fitness

The Fitness module is responsible for evaluating the output of the CA for use with EAs. Since fitness evaluation vary widely between applications, the interface of the Fitness module is designed to be simple and generic, such that the module is easy to replace.

It is connected in a dataflow-like manner between the Live Count Buffer and the Fitness Buffer. Whenever there is enough data available in the Live Count Buffer, it should fetch that data, processes it, and store the result in the Fitness Buffer. The host can then later retrieve it by activating the Fitness Sender.

To comply with the adaptive interface, there are a few things that the Fitness module needs to tell other parts of the system. First is the number of words per result, required by the Fitness Sender. Second is a unique identifier that is reported to the host by the Information Sender. To allow further versatility, the module can also report custom synthesis parameters to the host, as long as they all fit within 16-bits.

There are currently two implemented fitness modules: Live Count and DFT.

5.9.1 Live Count

The Live Count Fitness module is used to transfer the live counts to the host for software-side fitness evaluation. The implementation is as simple as it gets: The output of the Live Count Buffer is fed directly into the Fitness buffer, and the write and read signals are activated when the Live Count Buffer has data and the Fitness Buffer has space.

5.9.2 DFT

The DFT Fitness module uses a DFT to convert the live count data into frequency spectrums. This has the advantage of being able to pick up certain information that might be near-impossible to detect using conventional means. As shown in [2], using a DFT to interpret CA output holds potential.

A DFT of transform size N takes N complex numbers as input and produces N complex numbers as output. For transforms where all input numbers are real, the second half of the output mirrors the first half and can be safely ignored to save computation effort. Each output value of the DFT is a linear combination of all input values, the constants of which are known as twiddle factors. The twiddle factors rely only on the transform size and can therefore be computed ahead of time to reduce the complexity of the calculations.

This module is essentially a revised version of Støvneng's design in [32]. It has been refactored, streamlined and adapted to fit into the new design. It is also more customizable, supporting parameters other than powers of two, and can generate twiddle factors by itself during synthesis instead of relying on an external program. The twiddle factors are stored in the order they are needed, encoded in a fixed-point format since they range from -1 to 1.

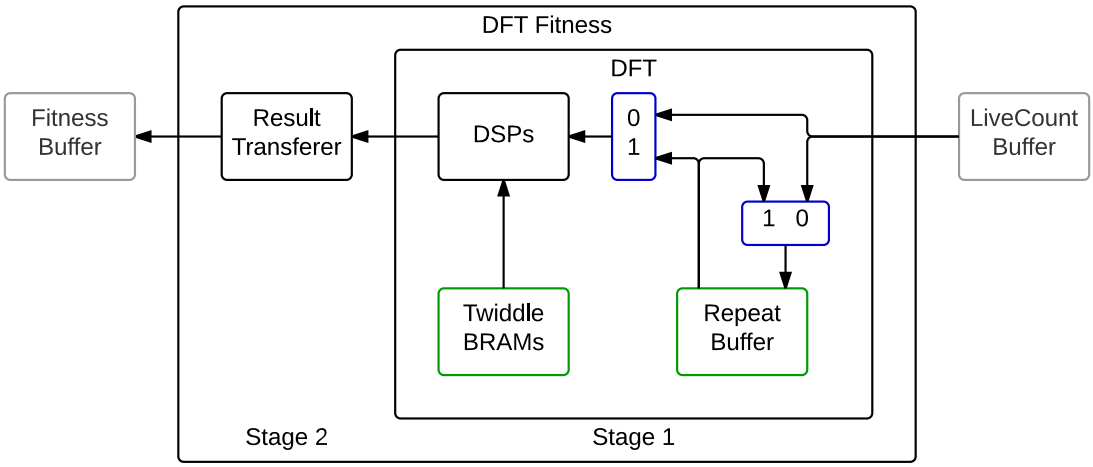


Figure 5.20: Detailed block diagram of the DFT Fitness module. The two pipeline stages are marked at the bottom. Blue boxes are multiplexers. Control signals are not shown.

To allow the DFT to continue processing while results are transferred from the DFT to the Fitness Buffer, the module is divided into two pipeline stages. This can be seen in Figure 5.20. The first stage contains the DFT while the second contains the logic for transferring the result. The pipeline is interlocked, since both stages can run for hundreds of cycles.

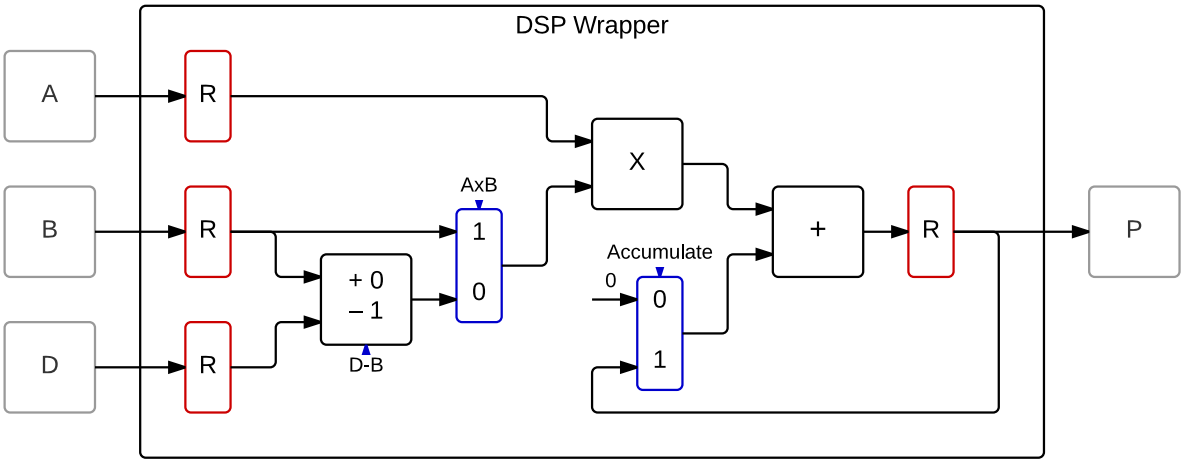


Figure 5.21: Detailed block diagram of the DSP Wrapper. Red boxes are registers and blue boxes are multiplexers.

The DFT is calculated using the DSP slice wrappers seen in Figure 5.21 in multiply-accumulate mode. Afterwards, the real and imaginary parts of each output value are combined into a single positive number by adding their absolutes. The most optimal configuration would be two DSP slices per output value; one for each real and imaginary part. However, FPGAs have a limited number of DSP slices. For larger transform sizes, the DSPs therefore have to calculate multiple output values in sequence. Since the Live Count Buffer, as all FIFO buffers, is delete-on-read, an

internal buffer is used to repeat the values. This Repeat Buffer is flushed and filled during the first calculation phase and then repeats the values to the DSPs during subsequent phases while also feeding the values back into itself. The state machine controlling the DFT is shown in Figure 5.22.

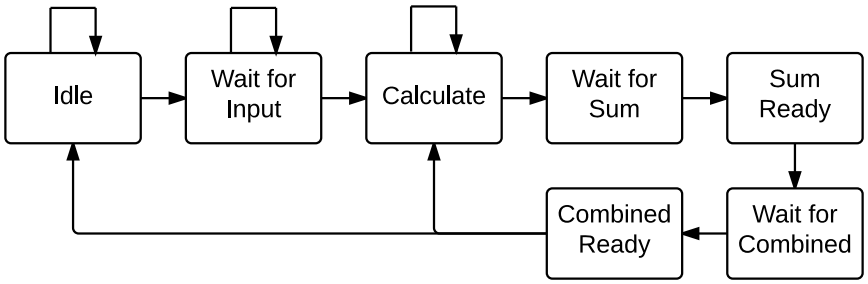


Figure 5.22: State machine controlling the DFT module.

5.10 Software API

In accord with the hardware design, the software API has also been given a complete overhaul. Everything has been rewritten from scratch to improve clarity, functionality and ease of use. Figure 5.23 illustrates the API structure: A main API is used to connect to the hardware platform, send instructions and receive data; and two optional APIs allow conversion of the received data into human-readable forms.

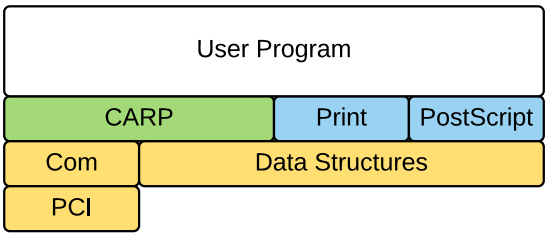


Figure 5.23: API structure. The main API is colored green, optional APIs blue and dependencies yellow.

5.10.1 Main API

The main API is used to connect, disconnect, and reset the hardware platform, as well as execute instructions and parse received data into data structures.

On connect, the API automatically clears any remaining data in the Transmission Buffer and queries the synthesis parameters with the Send Information instruction. The buffer is cleared once again on disconnect, and the synthesis information is nulled. The reset function can be used to set the system into freshly booted state

by clearing all counters, sblocks, buffers and BRAMs, except for the instruction memory.

Each instruction is given its own function with one-to-one correlation of parameters. Data structures are used to represent the parameters where possible, such as for LUTs and development rules. The instructions are queued in a buffer until the program tries to receive data or manually flushes it, to allow for debugging and creating test benches from the instruction stream.

For each instruction that should return data there is a corresponding getter function used to receive that data and parse it into a data structure. The getter functions are needed because of the hardware platform's separate instruction memory and control flow, which causes the amount of instructions issued by the API to not necessarily correspond to the number of times the instruction is executed.

5.10.2 Optional APIs

The two currently implemented optional APIs are Print and PostScript. Both are used to convert data structures into human-readable formats.

The Print API allows terminal printouts of all data structures, which includes system information, rule vectors and matrices (types, states or rule numbers).

The PostScript API allows programs to generate PostScript figures from one layer of a matrix data structure (types, states or rule numbers). The caller specifies the coloring scheme that is used by providing a function that takes in a cell value and returns a hexadecimal RGB value.

5.10.3 Communication

The communication part of the software API is split into two parts:

The first is a general interface for connecting to PCI and PCI Express devices without using a custom driver. It takes advantage of Linux' automatic population of **/sys/devices/pci*** with files representing the memory regions of all PCI and PCI Express devices. The directory is searched by vendor and device id, and the corresponding memory regions are memory-mapped into the program. Due to this direct interaction with device files, each program must be run with superuser rights.

The second is an interface specifically for the Communication module. It provides open, close, read and write functions similar to the old BenERA interface, in addition to implementing all special request functions in Table 5.2. When a read or write operation is initiated, buffers are checked for available data or space. If there is not enough present, the program waits for 1 microsecond and then rechecks.

5.10.4 Compilation

The compilation system has been streamlined to allow users to simply add new files in **libcarp/** or **programs/**, and have them automatically be integrated into the API or compiled with the API respectively by calling **make**.

First, all files in **libcarp/** is compiled to the statically linked library **libcarp.a**. Then each file in **programs/** is compiled with references to all the header files in **libcarp/** and the compiled library. Programs must include **carp.h** to use the main API and optionally **print.h** and **postscript.h** for the output APIs.

To facilitate unit testing, all programs whose name start with **test_** are executed in sequence when **make test** is called. **testframework.cinclude** is provided as a common assertion framework.

Flags

There are three compilation flags available: Debug, low-latency and testbench.

Debug mode causes various information to be printed at certain times during execution, such as during reset and when the API must wait for buffer space or data. To prevent excessive prints from the buffer checks, the delay between each check is increased to 100 ms.

Low-latency mode causes the buffer check delay to be skipped entirely, creating a busy-wait loop. It can be used when low latency is crucial, but should be unneeded in most circumstances.

Testbench mode can be used to create test benches when the hardware platform is unavailable. Instead of connecting to the board, synthesis parameters are mocked at compile time and the program prints the contents of the instruction buffer as a test bench and exits when the first buffer flush is triggered.

Chapter 6

Verification

A system would be useless if it did not operate according to specification. Therefore, the platform has been thoroughly verified through functional tests and an example program.

6.1 Functional Tests

The functionality of the platform has been verified under normal use conditions with the tests described in Appendix A. Each test has a short description and a list of the instructions that it verifies, given that it passes. Together, the 11 tests cover all system functionality except for fitness, which is designed to be application specific. The tests are implemented as separate programs, but use a shared test framework. A full system reset is performed before each test.

All tests are passed for all configurations of the platform that have been implemented. This includes 5x5, 16x16, 32x32, 4x4x4, 8x8x4, 8x8x8, 8x16x4 and 10x10x8 matrix sizes; 5, 6 and 8 type bits; 2, 4, 6, 7 and 8 rules tested in parallel; and 1, 2, 4 and 8 LUT configuration bits. The platform is therefore evaluated to be functional and scalable.

6.1.1 DFT

Although the fitness modules have no test programs, the DFT module and the DSP wrapper has their own test benches to ensure that refactoring has not changed functionality.

Testing shows that the DFT produces slightly different output than before. This is likely due differences is rounding or expression structures between the VHDL and python implementations for generating the twiddle factors. However, as shown in

Table 6.1, the new module is more precise for the default number of twiddle decimals and can be scaled to be even more precise.

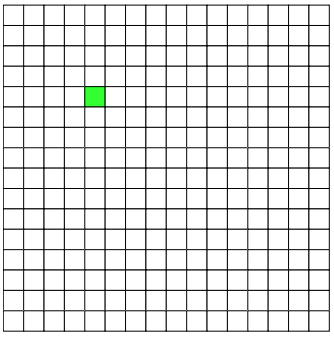
DFT Version	Twiddle Decimals	Absolute Error	
		Maximum	Average
Old	6	7	0.86
New	6	2	0.59
New	12	1	0.11

Table 6.1: Absolute error of new and old DFTs compared to numpy's rfft.

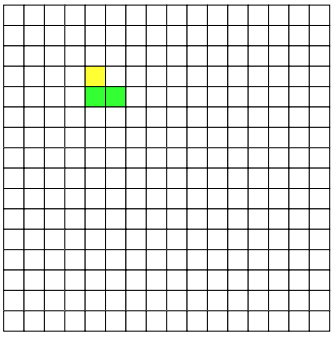
6.2 Example Replicator

As a proof of concept, the example in Figure 6.1 was created to showcase how development can be used to create simple self-replicating structures in a 2D CA that can grow to any desirable size given enough time. It was manually designed using 17 development rules and 13 cell types, and the source code is available in **programs/demo_replicator.c**.

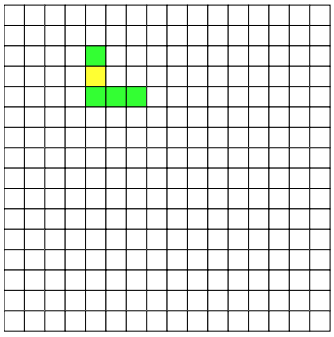
In essence, green cells creates copies of themselves going around in clockwise loops, while yellow cells shoot off from the corners to begin the creation of new loops. When loops are completed, the center cells turns red. After the entire structure has formed, the next step would be to use it for computation by specifying LUTs for each cell type. However, that is outside the scope of this simple example.



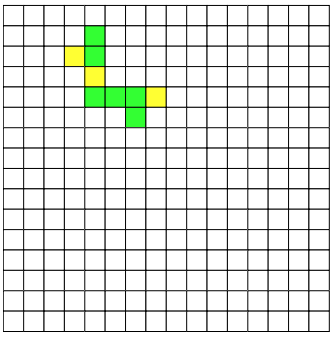
(a) Step 0



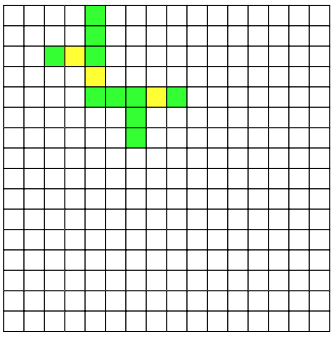
(b) Step 1



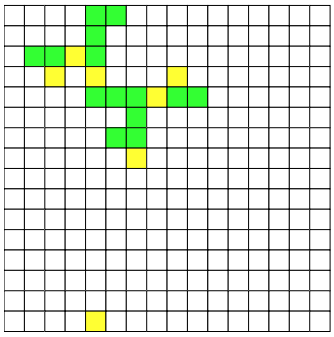
(c) Step 2



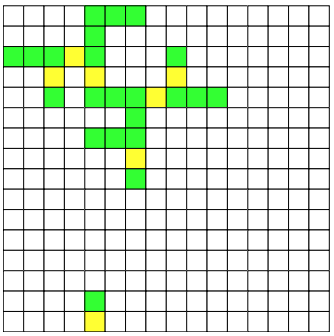
(d) Step 3



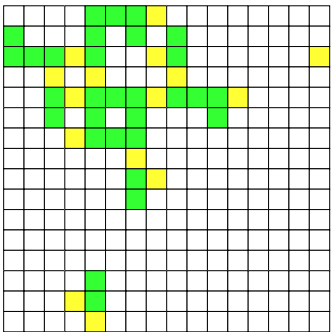
(e) Step 4



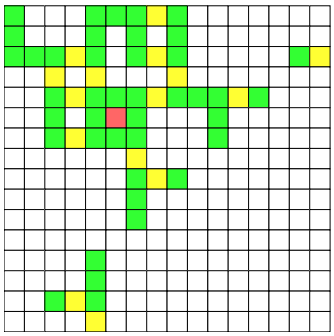
(f) Step 5



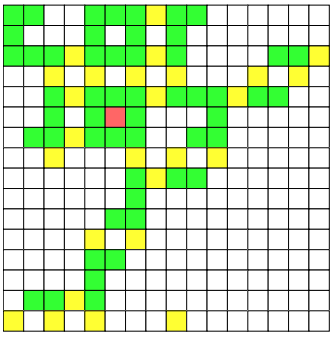
(g) Step 6



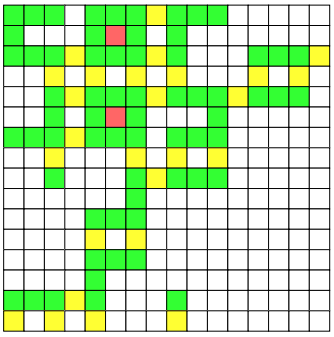
(h) Step 7



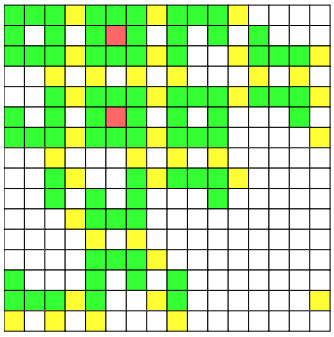
(i) Step 8



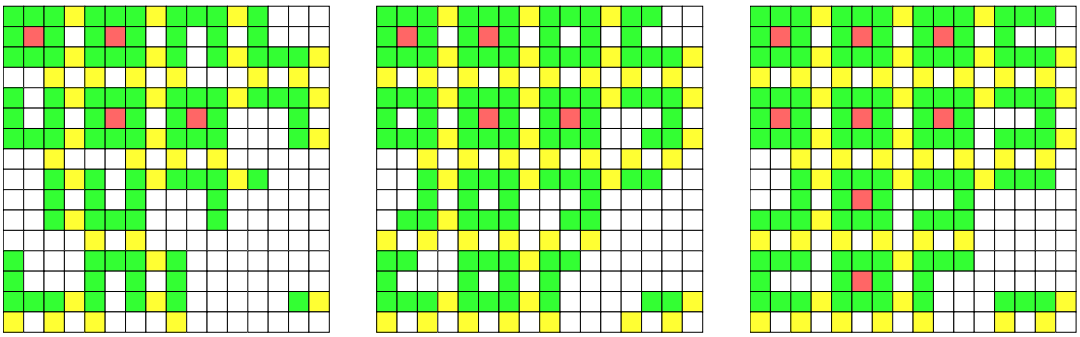
(j) Step 9



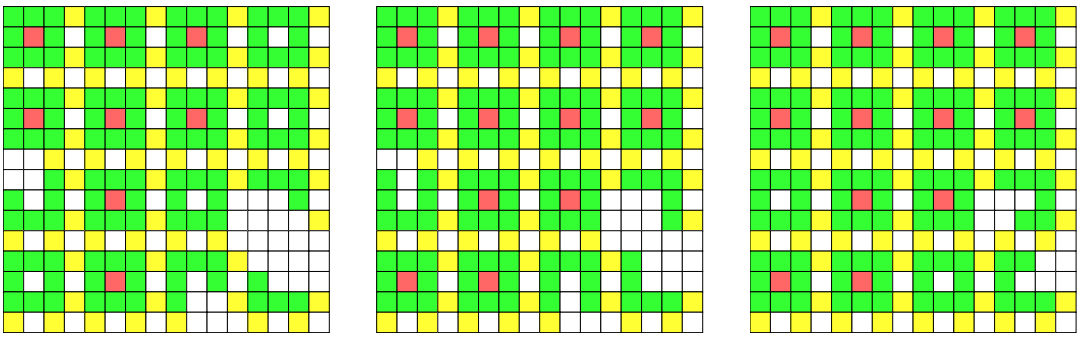
(k) Step 10



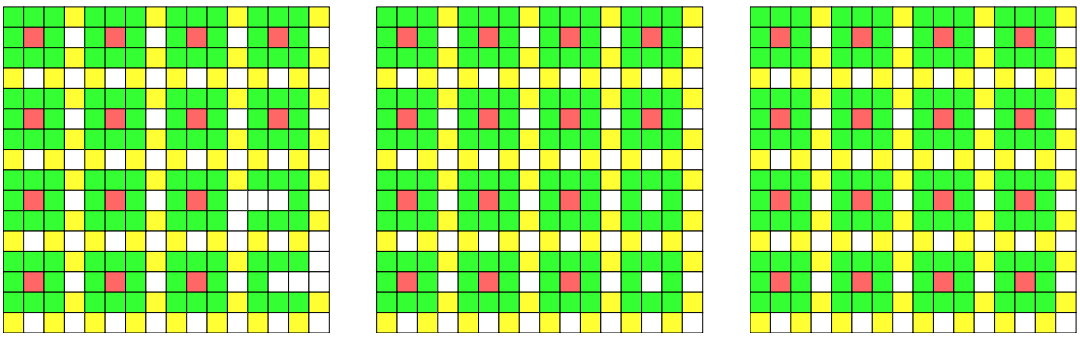
(l) Step 11



(m) Step 12 (n) Step 13 (o) Step 14



(p) Step 15 (q) Step 16 (r) Step 17



(s) Step 18 (t) Step 19 (u) Step 20

Figure 6.1: Simple 2D replicator. The images are created using the PostScript API.

Chapter 7

Discussion

FPGAs appear to be very suitable hardware for implementing CAs. The distributed and locally connected nature allow a large number of cells while keeping performance up at the same time. In the following sections, the platform's performance and resource usage is analysed, followed by a brief discussion of the challenges during development and potential future improvements.

7.1 Performance

As with the previous 3D design, the performance of certain modules scale with the matrix width, as they operate on one row of cells at a time. The main ones are are development, configuration and readback.

In addition, synthesis parameters allow the performance of some components to be scaled up or down in trade for resource usage. [Rules Tested In Parallel] can be increased to greatly improve the performance of development. The speed is equivalent to the previous design when set to 2, but designs have been successfully implemented with values of up to 16, depending on matrix size and configuration. [LUT Configuration Bits] controls the speed of CA configuration, which at maximum value is currently a bit slower than the previous design. Lower values ease routing however, which allow implementation of designs with larger matrices.

The speed of the DFT can be adjusted by setting the number of DSP slices and the transform size. However, as the fitness module is exchangeable, parameters have to be specified at the top of the respective VHDL file.

In circumstances where the matrices are 2D and relatively small, it is possible to synthesize a 3D matrix and stack multiple 2D matrices within it as individual layers. By using only 2D LUTs and development rules, the multiple matrices can then be developed and stepped in parallel, potentially reducing runtime by a large amount.

It is however incompatible with the current fitness system, since the Live Counter totals all cells regardless of how they are used.

7.1.1 Communication

The platform's new PCI Express based communication interface has been profiled by determining the latency and throughput of the link in both normal and low-latency mode.

The latency is measured by averaging the times for 100000 pings. Each ping is the execution a read_type instruction followed by retrieval of the type. Since all retrieval functions check the amount of buffer data first, this totals to 2.5 PCI Express round trips[1]plus one instruction execution per ping.

The throughput is measured by 1000 executions of the read_types instruction followed by retrieval of their data. The calculated number of words transferred are then compared to the time. The reads are interleaved so that an empty buffer is never encountered.

Mode	**Latency**	**Throughput**
Normal	60.3 µs	2.1 MB/s
Low-latency	7.3 µs	2.1 MB/s

Table 7.1: Performance of the PCI Express communication unit.

The results are presented in Table 7.1. The latency appears to be decent in low-latency mode, which is almost equivalent to succeeding the buffer check on the first try. Normal mode is around 50 µs slower as it fails the first check and therefore sleeps. The delay between checks is only set to 1 µs, but it is likely increased due to operating system scheduling.

The throughput is not exceptional at around 1% of the 256MB/s that the PCI Express endpoint block is theoretically capable of [41]. This is likely due to the simplistic PIO scheme that requires that all transfers are processed by the CPU. However, this is not as bad as one would first assume. Following is a short analysis of the desired throughput for the example program in Figure 7.1.

Assume the following synthesis parameters: [LUT Configuration Bits] maximized to 8, [Rules Tested In Parallel] set to 8, [Rule Amount] set to 256, [Fitness] set to DFT with transform size of 128 and 16-bit output values, and the matrix sized to 10x10x8.

That brings development speed to 3.2 cycles/cell and configuration speed to 1.6 cycles/cell (plus overhead)[2]. The DFT work in parallel with the rest of the design,

[1] Sending to the board requires no confirmation, and can thus be seen as half of a round trip.

```
initialize()
while counter[0] != 128:
    develop()
    config()
    step(128)
    send_fitness()
    swap_cell_storage()
    counter[0]++
```

Figure 7.1: Example program.

using less time, therefore adding no further delay. It produces 32 words of data after every 128 CA steps. With 800 cells, the time for each loop iteration then becomes a little over $3.2 \cdot 800 + 1.6 \cdot 800 + 128 = 3968$ cycles. 32 words per 3968 cycles at 125 Mhz constitutes around 4 MB/s. The communication unit can therefore supply 50% of the desired throughput, which should be acceptable for normal operation. A faster communication module is of course desirable, but it requires much more advanced logic that communicates through DMA with a custom driver.

7.1.2 Cellular Automaton

Previous hardware designs have been profiled by the test program in Figure 7.2 when synthesized with an 8x8 matrix and 6 development rules. It is mainly a stress-test of the CA stepping speed, and the fastest speed was 6.3 seconds in both [7] and [32].

```
initialize()
while counter[0] != 10000:
    config()
    step(50000)
    readback()
    swap_cell_storage()
    read_types()
    read_states()
    develop()
    counter[0]++
```

Figure 7.2: Updated test program from [7].

On the new platform, the program completes in 8.2 seconds. This is exactly twice of what it should take given 125 MHz speed. Further study show that the PCI Express Endpoint Core is to blame as it actually halves the frequency to 62.5 MHz at the user side. This fact is barely mentioned in [41] and is never referred to by Xilinx'

[2] The execution time of each instruction is detailed in Appendix C.

core generator or example design. It has also gone unnoticed in simulations since the communication module has been replaced with a special simulation version.

During synthesis and implementation, the constraint of 125 MHz have been applied to all signals though, and the critical paths require about 7.9 ns. This means that it is possible to (again) separate the communication module into a slower clock domain to return the remaining design to full speed. The new design would then be 35% faster instead of 30% slower for this test.

Unfortunately, the late discovery of the problem leaves insufficient time to fix it, but it should be fairly straightforward. For the purpose of other speed and resource comparisons, the platform is therefore assumed to run at 125 MHz.

7.2 Resource Usage

With a complete rewrite, it is interesting to see the differences in resource usage between equivalent setups. Due to architectural differences, the performance of the new design can not be perfectly matched with that of the previous, but it should be close enough to determine the general trend. The following configuration assume that the design is running at the intended 125 MHz.

The most closely matching configuration is: [LUT Configuration Bits] maximized to 2 in 2D and 8 in 3D, [Rules Tested In Parallel] set to 2, [Rule Amount] set to 256, [Type Bits] set to 5, [State Bits] set to 1, [Fitness] set to Live Count and buffer sizes set to 256. Other parameters do not significantly influence resource usage, functionality or performance.

In 3D, the live counter is four times faster, configuration is half as fast, and readback is an eight as fast. In 2D, configuration is a quarter as fast for 32x32 matrices. Due to different scaling in the old 2D design, no other matrix sizes have equivalent performance and have therefore been left out of the comparison. The results are presented in Table 7.2.

The new design appears to be slightly more efficient in 3D. It uses about the same amount of LUTs, but substantially fewer registers and slightly fewer BRAMs. This is a bit surprising considering the four times larger adder tree in the Live Counter. In 2D, both LUT and register usage are drastically reduced while BRAM usage has gone up.

The size of the matrix is limited by the number of available 16-bit shift registers (SRL16s), as each sblock uses 2 in 2D and 8 in 3D. Since the new design is more finely tunable, larger matrices can fit onto the chip. A 10x10x8 matrix design uses 99.9% of the 6408 shift registers on the Spartan-6 LX45T, and has been successfully implemented and tested with the above configuration. It will even implement with [Rules Tested In Parallel] increased to 6 and [Fitness] set to a DFT using 32 out of the 58 available DSP slices.

Matrix	SRL16		LUTs		Registers		BRAMs	
(XxYxZ)	Total	%	Old	New	Old	New	Old	New
32x32	2048	32.0	14858	11277	16259	7043	38	53
8x8x4	2048	32.0	6529	6265	6011	4495	55	47
8x8x8	4096	63.9	7668	8374	5726	4913	50	47
8x16x4	4096	63.9	8234	8252	6531	4957	50	47
10x10x8	6400	99.9	–	11313	–	5832	–	52

Table 7.2: Resource usage without DFT compared to the old design with most equivalently configured setup and performance. The old numbers are from [32].

At this point, routing becomes the main problem, as there are still many logic resources left but nearly all paths are of critical length. Both the shift registers and DSP slices are spread across the entire FPGA. This means that values from the sblocks must be routed from the entire FPGA into one location when counting and then spread out to the entire FPGA again when computing the DFT.

It 2D, the design clogs up before all shift registers can be used. The largest matrix that has therefore been successfully implemented is 50x50. This is with the above configuration, except for [LUT Configuration Bits] and [Rules Tested In Parallel] reduced to 1, but with [Fitness] set to DFT.

7.3 Challenges

ISE lacks support for VHDL-2008, which is required to use custom types with generics. All array signals must therefore be converted to and from std_logic_vectors before exiting and after entering modules. Hopefully, this is implemented in a way that is both organized and understandable.

It appears that the USB cable driver for JTAG provided by Xilinx has some problems with newer Linux kernels; it certainly does not detect the board in the current hardware setup. Thankfully, a third-party driver was found at [12] which proved to be compatible and solved the problem.

7.4 Future work

The most important issue that needs remediation is the downclocking caused by the PCI Express Endpoint Core, which halves the performance of the entire design. The relatively straightforward fix is to reintroduce a second clock domain, but clock domain traversals might be tricky.

The second most important issue, and by far the most challenging, is the relatively low throughput of the communication module. Improvement requires the use of DMA in combination with a custom device driver. It has the added benefit of allowing programs to run without superuser rights, although the driver will need them instead.

It is possible to further parameterize some parts of the design. First is the number of rows that can be read from Cell Storage, which will allow for higher CA configuration and readback speeds. Another is the number of cycles used by the Live Counter, which will allow the user to trade speed for less resource usage and easier routing. Finally, the Live Counter could be made exchangeable by an interface similar to fitness, to allow more complex evaluation of the output. It might even be baked into fitness.

The current design only executes one instruction at a time to keep everything simple and organized. However, there are many circumstances where the next instruction do not impact the currently executing one and can safely be run in parallel to gain a small speedup. It should be possible to implement this by letting a form of hazard detection module handle the interlocking signals for the main pipeline, such that the run signal is asserted as soon as all modules that the next instruction depends on are done.

Since some instructions have the ability to create deadlocks, some sort of watchdog-timer might be useful to allow recovery of the system. The only alternative is cycling the power, but that will break the PCI Express connection which necessitates a reboot of the host as well.

7.5 Warnings

During synthesis, the design produces a substantial amount of warnings, although significantly less than the previous. Most arise from unused signals due to the selected synthesis parameters and are completely benign. In addition, a good deal stem from the provided PCI Express Endpoint Core, which can also be safely ignored.

In an effort to reduce clutter and direct focus at more important information, most of the benign warnings are blocked by a filter. The remaining could not be removed without possibly causing trouble later, due to a somewhat limited filter system. For a standard 3D design, this reduces the amount of warnings from around 500 to about 50 [3].

[3] Synthesizing from within ISE produces around 730 warnings, opposed to the 500 when using XST directly. However, both are filtered down to about 50.

Chapter 8

Conclusion

In this paper, NTNU's CA research platform has been completely re-engineered. The overall structure is equivalent, but the hardware design has been made more modular, configurable and structured, and the software API more adaptable, complete and user-friendly. The 2D and 3D designs have been unified and any external dependencies removed. It should also be easily extendable to larger FPGAs.

The platform has been thoroughly tested in hardware and all issues with the previous design that are listed in Table 3.1 have been resolved. It can fit CAs of up to 50x50 cells in 2D and 10x10x8 cells in 3D on a Spartan-6 LX45T. There are however some performance setbacks with the communication module, which is slightly ironic as its implementation was the original task of the project. However, when the design is restored to 125 MHz, the raw CA performance should be about 35% higher in 2D and 300% higher in 3D at lower or equivalent resource usage.

In essence, this thesis provides a complete tool for CA research and experimentation, and allows study of self-organization, adaptation, replication and other applicable biological processes. It can be integrated into any computer system with a recent version of Linux and an available PCI Express socket.

Bibliography

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

[2] Sivert Berg. Evolution of Cellular Automata using Lindenmayer Systems and Fourier Transforms. 2013.

[3] Nolan Carroll. Lindenmayer System Generator. `http://nolandc.com/sandbox/fractals/`. [Accessed: 2015-05-26].

[4] Edgar F Codd. *Cellular Automata*. Academic Press, 1968.

[5] Hugo de Garis, Michael Korkin, and Gary Fehr. The CAM-Brain Machine (CBM): An FPGA Based Tool for Evolving a 75 Million Neuron Artificial Brain to Control a Lifesized Kitten Robot. *Autonomous Robots*, 10(3):235–249, 2001.

[6] DistroWatch. DistroWatch Page Hit Ranking. `http://distrowatch.com/dwres.php?resource=popularity`. [Accessed: 2015-05-18].

[7] Asbjørn Djupdal. Konstruksjon av maskinvare for kjøring av sblokkbaserte eksperimenter. 2003.

[8] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[9] Dario Floreano and Francesco Mondada. Evolution of Homing Navigation in a Real Mobile Robot. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(3):396–407, 1996.

[10] Martin Gardner. Mathematical Games. *Scientific American*, 216, 1967.

[11] Martin Gardner. Mathematical Games – The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[12] Michael Gernoth. XILINX JTAG tools on Linux without proprietary kernel modules. `http://rmdir.de/~michael/xilinx/`. [Accessed: 2014-12-16].

[13] Felix Gers, Hugo de Garis, and Michael Korkin. CoDi-1Bit: A Simplified Cellular Automata Based Neuron Model. In *Artificial Evolution*, pages 315–333. Springer, 1998.

[14] Carlos Gershenson. Introduction to Random Boolean Networks. *arXiv preprint nlin/0408006*, 2004.

[15] Jim Giles. Utopian dream in tatters as starlab crashes to earth. *Nature*, 412(6842):6–6, 2001.

[16] David E Golberg. Genetic Algorithms in Search, Optimization, and Machine Learning. *Addion wesley*, 1989, 1989.

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

[18] Simon Harding and Wolfgang Banzhaf. Artificial Development. In *Organic Computing*, pages 201–219. Springer, 2008.

[19] Francis Heylighen et al. The science of self-organization and adaptivity. *The encyclopedia of life support systems*, 5(3):253–280, 2001.

[20] John Henry Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT press, 1992.

[21] Gregory S. Hornby, Al Globus, Derek S. Linden, and Jason D. Lohn. Automated Antenna Design with Evolutionary Algorithms. In *AIAA Space*, pages 19–21, 2006.

[22] Lorentz Huelsbergen, Edward Rietman, and Robert Slous. Evolution of Astable Multivibrators in Silico. In *Evolvable Systems: From Biology to Hardware*, pages 66–77. Springer, 1998.

[23] Chris G Langton. Computation at the Edge of Chaos: Phase Transitions and Emergent Computation. *Physica D: Nonlinear Phenomena*, 42(1):12–37, 1990.

[24] Aristid Lindenmayer. Mathematical Models for Cellular Interactions in Development, Parts I and II. *Journal of Theoretical Biology*, 18(3):280–315, 1968.

[25] Stelios Manousakis. Musical L-systems. *Koninklijk Conservatorium, The Hague (master thesis)*, 2006.

[26] Norman Margolus. CAM-8: A Computer Architecture Based on Cellular Automata. *Pattern Formation and Lattice-Gas Automata*, pages 167–187, 1996.

[27] Allen Newell, Herbert Alexander Simon, et al. *Human Problem Solving*, volume 104. Prentice-Hall Englewood Cliffs, NJ, 1972.

[28] Logic Systems Laboratory of the Swiss Federal Institute of Technology in Lausanne. Overview of the BioWall. `http://lslwww.epfl.ch/biowall/VersionE/DescriptionE/DescriptionE.html`. [Accessed: 2015-06-04].

[29] PCI-SIG. PCI Express Base Specifications Revision 3.0. *PCI-SIG*, 2010.

[30] Moshe Sipper. The Emergence of Cellular Computing. *Computer*, 32(7):18–26, 1999.

[31] Moshe Sipper, Eduardo Sanchez, Daniel Mange, Marco Tomassini, Andrés Pérez-Uribe, and André Stauffer. A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. *Evolutionary Computation, IEEE Transactions on*, 1(1):83–97, 1997.

[32] Ola Martin Tiseth Støvneng. Extending an sblock platform for a new target hardware. 2014.

[33] Gianluca Tempesti, Daniel Mange, André Stauffer, and Christof Teuscher. The BioWall: An Electronic Tissue for Prototyping Bio-Inspired Systems. In *Evolvable Hardware, 2002. Proceedings. NASA/DoD Conference on*, pages 221–230. IEEE, 2002.

[34] Adrian Thompson. An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics. In *Evolvable Systems: From Biology to Hardware*, pages 390–405. Springer, 1997.

[35] Adrian Thompson and Paul Layzell. Analysis of Unconventional Evolved Electronics. *Communications of the ACM*, 42(4):71–79, 1999.

[36] John von Neumann and Arthur W Burks. Theory of Self-Reproducing Automata. 1966.

[37] Stephen Wolfram. Universality and Complexity in Cellular Automata. *Physica D: Nonlinear Phenomena*, 10(1):1–35, 1984.

[38] Xilinx. *Virtex Series Configuration Architecture User Guide*, 2004. XAPP151 (v1.7).

[39] Xilinx. *SP605 Hardware User Guide*, 2011. UG526 (v1.6).

[40] Xilinx. *Spartan-6 Family Overview*, 2011. DS160 (v2.0).

[41] Xilinx. *Spartan-6 FPGA Integrated Endpoint Block for PCI Express*, 2011. UG672 (v1.1).

[42] Xilinx. *Spartan-6 FPGA Configuration User Guide*, 2014. UG380 (v2.7).

Appendix A

Test Descriptions

Test	Description	Verifies
0	Write types one by one, read them, check that types match	Write One Type, Read One Type
1	Write types row by row, read all, check that types match	Write Row Types, Read All Types
2	Write states one by one, read them, check that states match	Write One State, Read One State
3	Write states row by row, read all, check that states match	Write Row States, Read All States
4	Fill cells, check that all cells have correct type and state	Fill Cells
5	Write states and types, swap, check that data is gone, swap again, check that data is back	Swap Cell Storage
6	Write rules and types, develop, check rules have triggered and types have updated	Develop, Write Rule, Read Rule Vectors, Read Rule Numbers
7	Write states, configure the sblock matrix, read it back, check that states are unchanged	Config, Readback
8	Write states, types and LUTs, configure sblock matrix, check states have changed	Step, Write LUT
9	Store program that sends 1 and then breaks, jump to program address three times, check for three 1s	Store, End, Jump, Break
10	Execute program that sends 1, increments counter and jumps to itself unless the counter is equal to three, check for three 1s	Jump Equal, Counter Reset, Counter Increment

Appendix B

Attached Files

An archive containing the code for the hardware design and the software API should be distributed with this thesis. If not, all resources related to the project can be found online at `https://github.com/lundal/carp`.

Nearly all files from previous designs have been completely discarded, but parts of old code have been reused in a few places.

Directory Structure

```
attachment.zip
├── api
│   ├── libcarp
│   └── programs
└── vhdl
    ├── ipcores
    ├── modules
    │   ├── cellular_automata
    │   ├── communication
    │   ├── development
    │   ├── fetch
    │   ├── fitness
    │   └── utility
    ├── packages
    ├── sp605
    └── testbenches
```

API Files

api
- makefile ..... Build script

api/libcarp
- bitvector.[c|h] ..... Utility bit string builder
- carp.[c|h] ..... Main API
- communication.[c|h] ..... Communication module interface
- instructions.h ..... Instruction code definitions
- lut.h ..... LUT data structure
- matrix.[c|h] ..... Matrix data structure
- pci.[c|h] ..... Generic PCI/PCI Express interface
- postscript.[c|h] ..... Optional PostScript API
- print.[c|h] ..... Optional print API
- rule.h ..... Development rule data structure
- utility.[c|h] ..... Various utility functions

api/programs
- demo_development.c ..... Simple development example
- demo_replicator.c ..... Simple replicator example
- demo_stacked.c ..... Simple stacked-mode example
- profile_communication.c ..... Profiles latency and throughput
- test_config_readback.c ..... Test 7
- test_counters.c ..... Test 10
- test_development.c ..... Test 6
- test_fill_cells.c ..... Test 4
- test_instructions_storage.c ..... Test 9
- test_sblockmatrix.c ..... Test 8
- test_swap_cell_storage.c ..... Test 5
- test_write_read_state.c ..... Test 2
- test_write_read_states.c ..... Test 3
- test_write_read_type.c ..... Test 0
- test_write_read_types.c ..... Test 1
- testframework.cinclude ..... Framework used by all tests

VHDL Files

vhdl
- carp.xise ..... ISE project file
- filter.filter ..... Warning filter rules
- makefile ..... Build script
- parameters.conf ..... Synthesis parameters

vhdl/ipcores
- sp605_pcie.xco ..... PCI Express IP core configuration

vhdl/modules
- cell_storage.vhd ..... Cell Storage
- cell_storage_mux.vhd ..... Cell Storage Multiplexer
- cell_writer_reader.vhd ..... Cell Writer Reader
- decode.vhd ..... Decode
- fitness_sender.vhd ..... Fitness Sender
- information_sender.vhd ..... Information Sender
- lut_writer.vhd ..... LUT Writer
- resetter.vhd ..... Buffer Resetter
- rule_numbers_reader.vhd ..... Rule Numbers Reader
- rule_vector_reader.vhd ..... Rule Vector Reader
- rule_writer.vhd ..... Rule Writer
- send_buffer_mux.vhd ..... Send Buffer Multiplexer
- toplevel.vhd.in ..... Toplevel; preprocessed to toplevel.vhd

vhdl/modules/cellular_automata
- cellular_automata.vhd ..... Cellular Automaton
- live_counter.vhd ..... Live Counter
- lut_configurable.vhd ..... Configurable LUT
- sblock.vhd ..... Single sblock
- sblock_matrix.vhd ..... Sblock Matrix

vhdl/modules/communication
- communication.vhd ..... PCI Express communication module
- communication_sim.vhd ..... Simulation version which exposes buffers
- rq_special.vhd ..... Special Request Handler
- rx_engine.vhd ..... Reception Engine: Parses TLPs
- tx_engine.vhd ..... Transmission Engine: Builds TLPs

vhdl/modules/development
- cell_fetcher.vhd ..... Cell Fetcher
- development.vhd ..... Development
- hits_to_numbers.vhd ..... Hits To Numbers processor
- hits_to_vector.vhd ..... Hits To Vector processor
- rule_fetcher.vhd ..... Rule Fetcher
- rule_tester.vhd ..... Single-rule Rule Tester
- rule_tester_multi.vhd ..... Multi-rule Rule Tester
- rule_testers_multi.vhd ..... Multiple multi-rule Rule Testers

vhdl/modules/fetch
- fetch.vhd ..... Fetch
- fetch_communication.vhd ..... Fetch Communication
- fetch_handler.vhd ..... Fetch Handler

vhdl/modules/fitness
- dft.vhd ..... DFT
- dsp_wrapper.vhd ..... DSP Wrapper
- fitness_dft.vhd ..... DFT Fitness
- fitness_live_counter.vhd ..... Live Count Fitness
- twiddles.vhd ..... DFT twiddle factors generator

vhdl/modules/utility
- bit_counter_32.vhd ..... Counts number of ones in 32 bits
- bit_counter_N.vhd ..... Counts number of ones in N bits
- bram_1toN.vhd ..... 1-in N-out BRAM
- bram_tdp.vhd ..... Dual-port BRAM
- combiner.vhd ..... Combiner
- fifo.vhd ..... FIFO
- selector.vhd ..... Selector
- shifter.vhd ..... Static shifter
- shifter_dynamic.vhd ..... Dynamic shifter
- shift_register.vhd ..... Shift register

vhdl/packages
- functions.vhd ..... Utility functions
- instructions.vhd ..... Instruction codes
- types.vhd ..... Custom type declarations

vhdl/sp605
- constraints.ucf.in ..... Constraints; preprocessed to constraints.ucf
- pcie_wrapper.vhd ..... PCI Express Endpoint Core wrapper

```
vhdl/testbenches
├── test_template.vhd ..... Testbench template
├── test_dft.vhd ..... DFT testbench
└── test_dsp_wrapper.vhd ..... DSP Wrapper testbench
```

Appendix C

Instruction Set Architecture

Cellular Automata Research Platform Instruction Set Architecture

Revision 1.0

2014-06-14

Contents

1 Introduction **1**
Instructions . . . 2
Rules . . . 3
LUTs . . . 4

2 General Instructions **5**
No Operation . . . 6
Read Information . . . 7
Read Fitness . . . 8
Swap Cell Storage . . . 9
Reset Buffers . . . 10

3 Development Instructions **11**
Read Rule Vectors . . . 12
Read Rule Numbers . . . 13
Write Rule . . . 14
Set Active Rules . . . 15
Develop . . . 16

4 Cell Storage Instructions **17**
Read One State . . . 18
Read All States . . . 19
Read One Type . . . 20
Read All Types . . . 21
Write One State . . . 22
Write Row of States . . . 23
Write One Type . . . 24
Write Row of Types . . . 25
Fill Cells . . . 26

5 Cellular Automaton Instructions **27**
Write LUT . . . 28
Configure . . . 29
Readback . . . 30
Step . . . 31

6 Control Flow Instructions **33**
Break . . . 34
Store . . . 35
End . . . 36
Jump . . . 37
Jump Equal . . . 38
Increment Counter . . . 39
Reset Counter . . . 40

1 Introduction

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

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

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

Instructions

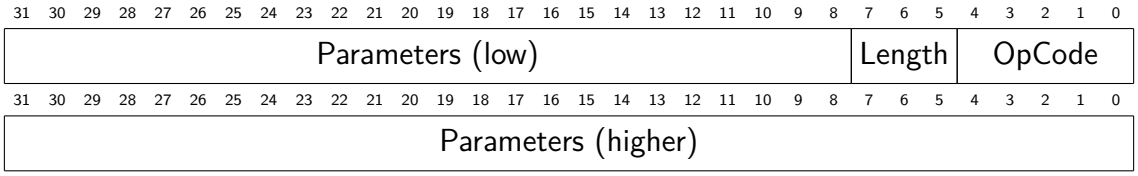
Each instruction is 256 bits and consists of a 5-bit operation code, a 3-bit length field and up to 248 bits of parameters.

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

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

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

Instruction Format



...

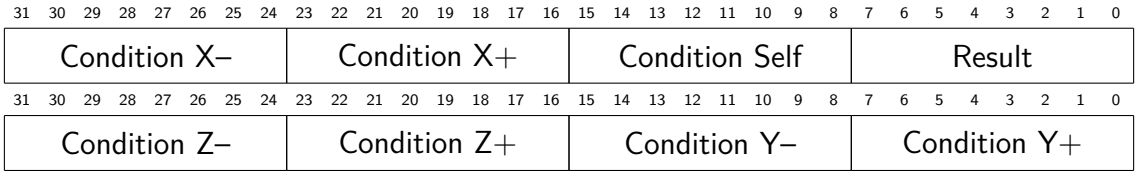
Rules

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

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

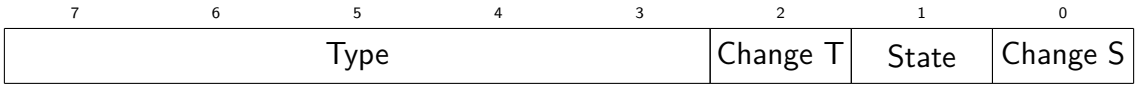
In the formats below, [type bits] is assumed to be 5 and [states bits] 1 for the purpose of having everything nicely align to bytes.

Rule Format



Condition Format

Result Format



Notes

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

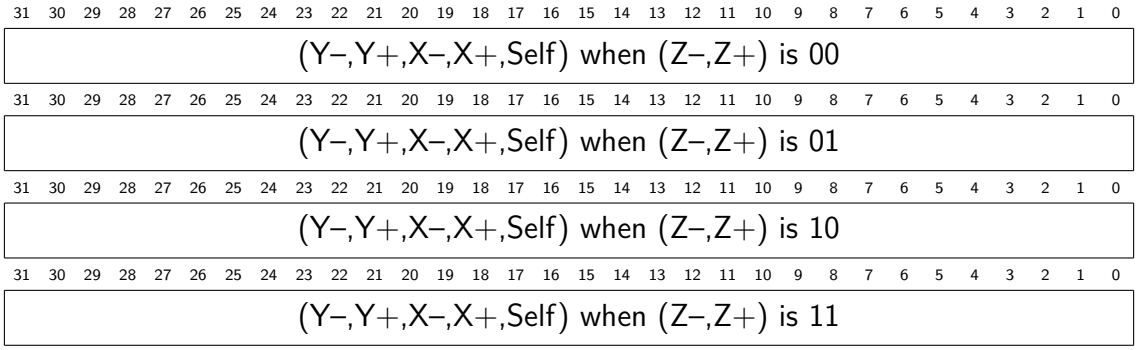
Conditions for Z are ignored when [matrix depth] is 1.

LUTs

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

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

LUT Format



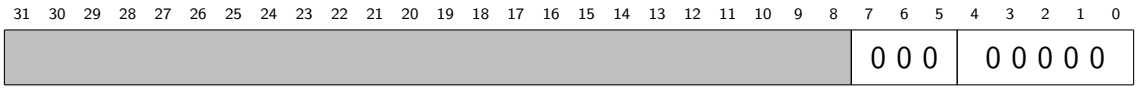
Notes

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

2 General Instructions

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

No Operation



Format

nop()

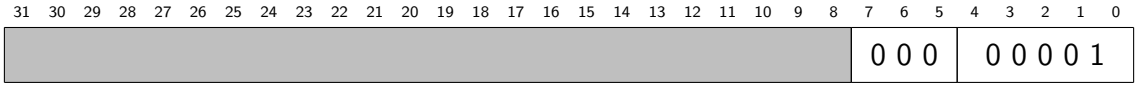
Purpose

To do nothing for one cycle.

Description

Nothing is done for one cycle.

Read Information



Format

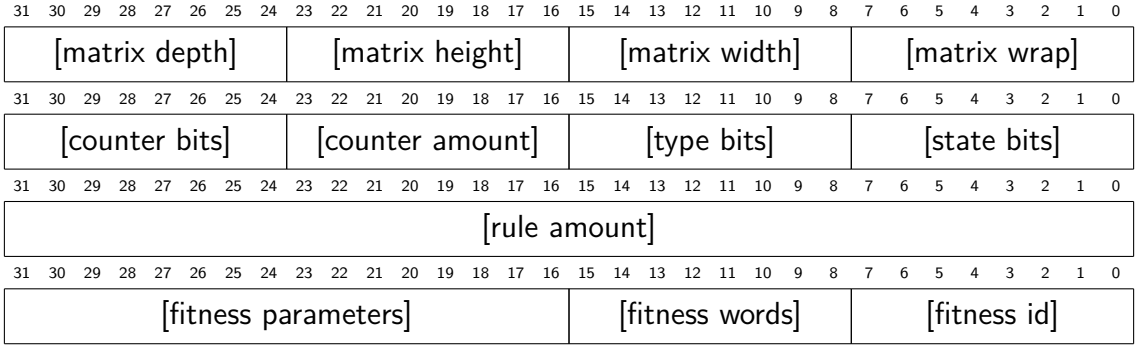
read_information()

Purpose

To retrieve information about the system.

Description

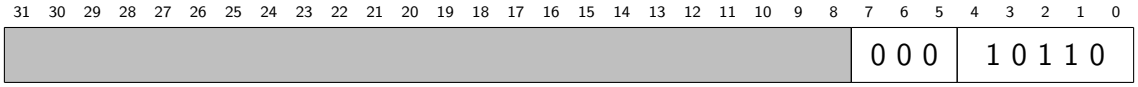
The following words are put into the Send Buffer.



Notes

This instruction takes 4 cycles.

Read Fitness



Format

read_fitness()

Purpose

To retrieve a fitness value.

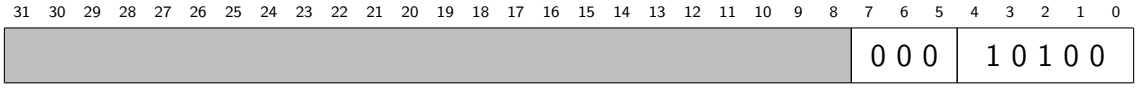
Description

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

Notes

This instruction takes [fitness words] cycles.

Swap Cell Storage



Format

swap_cell_storage()

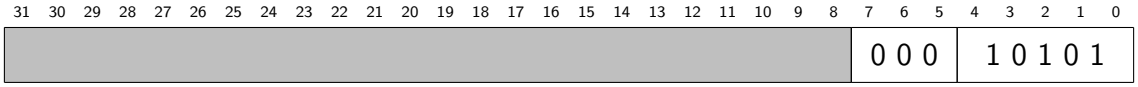
Purpose

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

Description

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

Reset Buffers



Format

reset_buffers()

Purpose

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

Description

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

Notes

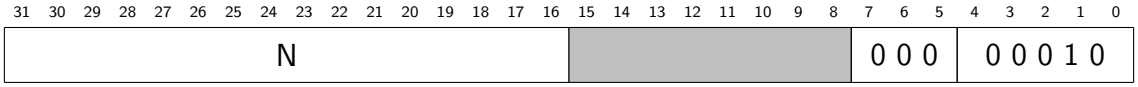
If the Fitness module is processing data, the contents of the Live Count and Fitness Buffers may become undefined.

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

3 Development Instructions

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

Read Rule Vectors



Format

read_rule_vectors(N)

Purpose

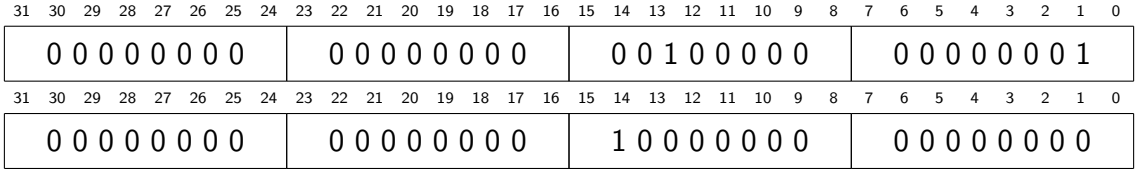
To retrieve N rule vectors.

Description

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

Example

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

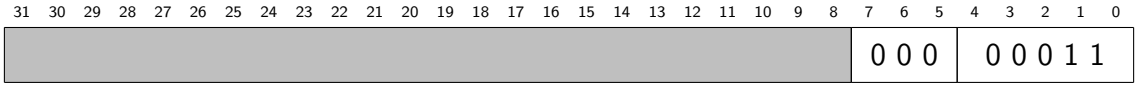


Notes

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

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

Read Rule Numbers



Format

read_rule_numbers()

Purpose

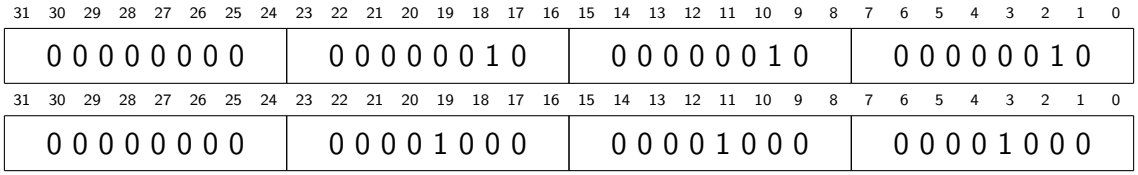
To retrieve the last rule that triggered for each cell during the previous development step.

Description

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

Example

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

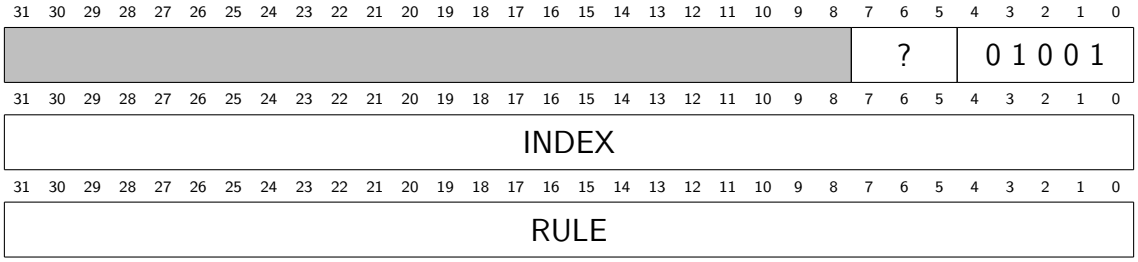


Notes

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

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

Write Rule



Format

write_rule(RULE, INDEX)

Purpose

To write a development rule.

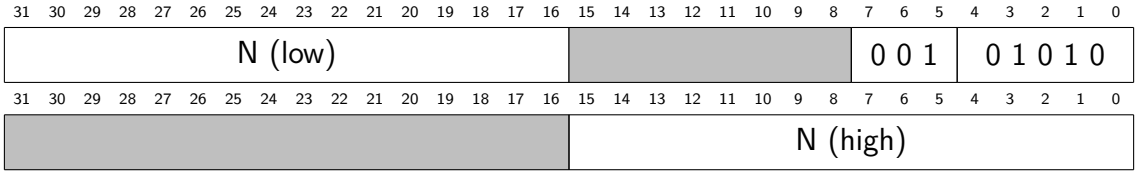
Description

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

Notes

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

Set Active Rules



Format

set_rules_active(N)

Purpose

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

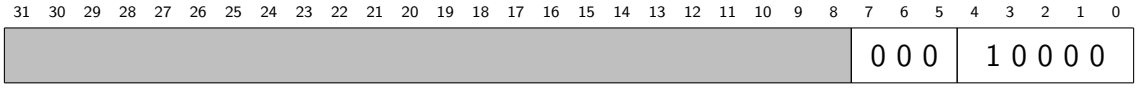
Description

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

Notes

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

Develop



Format

develop()

Purpose

To perform development on all cells.

Description

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

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

Notes

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

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

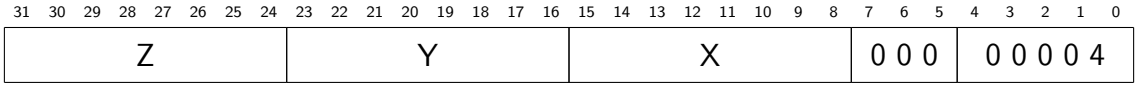
$$T_{3D} = M_Z M_Y \max\left(\frac{R_A + 1}{R_{TIP}}, 7\right) + 6$$

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

4 Cell Storage Instructions

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

Read One State



Format

read_state(Z, Y, X)

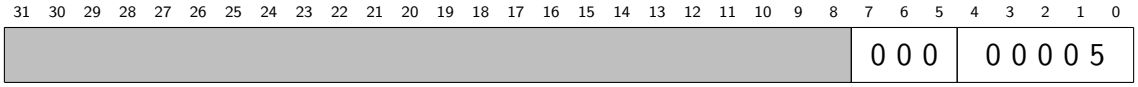
Purpose

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

Description

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

Read All States



Format

read_states()

Purpose

To retrieve the state of all cells.

Description

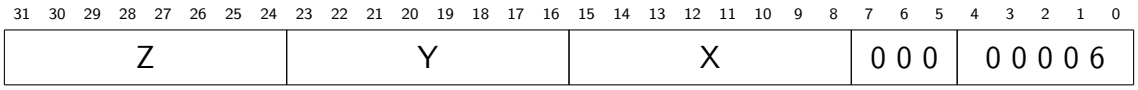
The states of all cells are put into the Send Buffer in raster order (first X, then Y, then Z). The Send Buffer is word-aligned after each row by padding with 0. If a state would be split across two words, it is instead aligned to the next word.

Notes

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

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

Read One Type



Format

read_type(Z, Y, X)

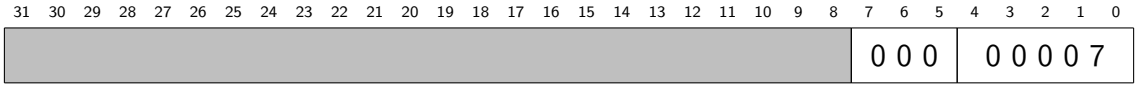
Purpose

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

Description

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

Read All Types



Format

read_types()

Purpose

To retrieve the types of all cells.

Description

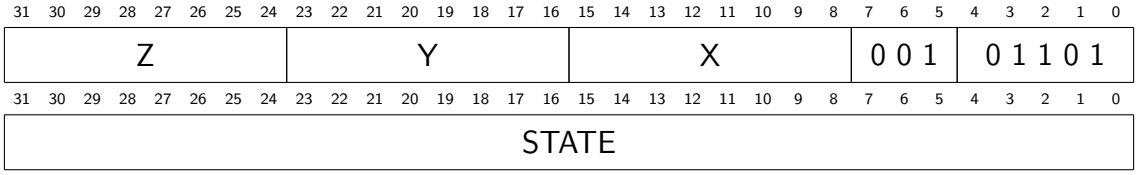
The types of all cells are put into the Send Buffer in raster order (first X, then Y, then Z). The Send Buffer is word-aligned after each row by padding with 0. If a type would be split across two words, it is instead aligned to the next word.

Notes

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

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

Write One State



Format

write_state(Z, Y, X, STATE)

Purpose

To write one state.

Description

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

Notes

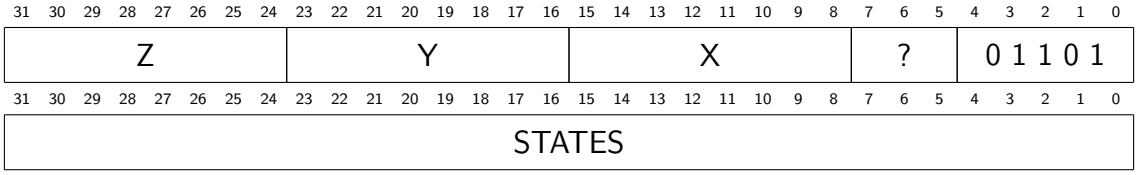
Each coordinate is cropped to the bits in its matrix dimension.

STATE is cropped to [state bits].

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

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

Write Row of States



Format

write_states(Z, Y, X, STATES)

Purpose

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

Description

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

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

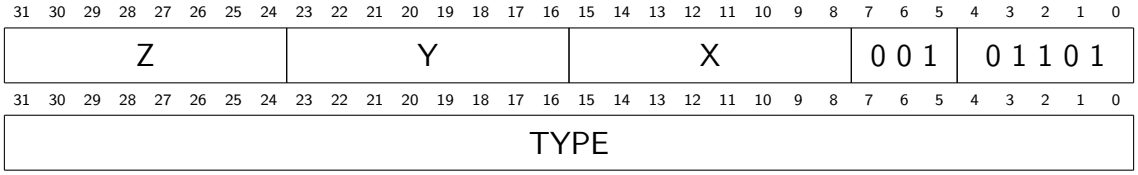
The length of STATES varies depending on [matrix width] and [state bits]. It is sent as one continuous piece spanning multiple words. The instruction length field is adjusted accordingly.

Notes

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

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

Write One Type



Format

write_types(Z, Y, X, TYPE)

Purpose

To write one state.

Description

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

Notes

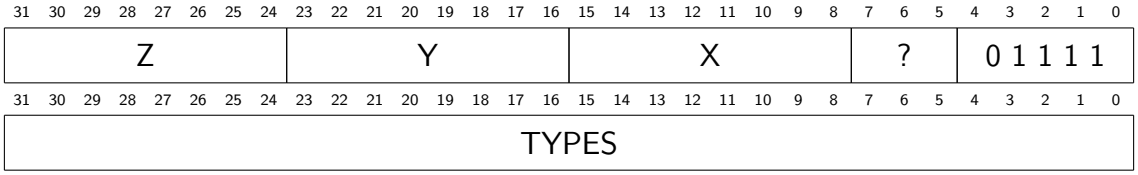
Each coordinate is cropped to the bits in its matrix dimension.

TYPE is cropped to [type bits].

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

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

Write Row of Types



Format

write_types(Z, Y, X, TYPES)

Purpose

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

Description

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

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

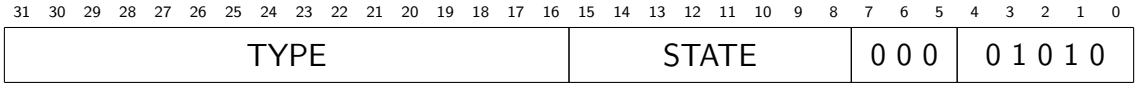
The length of TYPES varies depending on [matrix width] and [type bits]. It is sent as one continuous piece spanning multiple words. The instruction length field is adjusted accordingly.

Notes

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

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

Fill Cells



Format

fill_cells(STATE, TYPE)

Purpose

To set the state and type of all cells.

Description

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

Notes

STATE is cropped to [state bits].

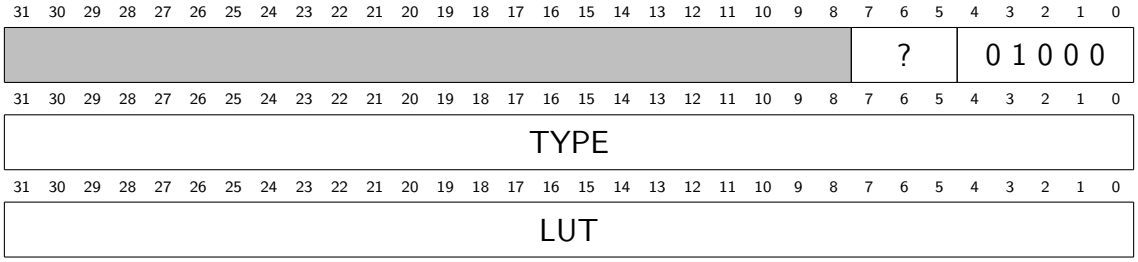
TYPE is cropped to [type bits].

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

5 Cellular Automaton Instructions

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

Write LUT



Format

write_lut(LUT, TYPE)

Purpose

To write a type to lookup table conversion entry.

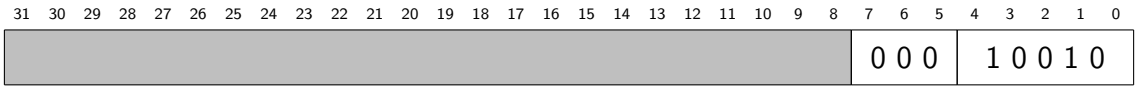
Description

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

Notes

TYPE is cropped to [type bits].

Configure



Format

config()

Purpose

To configure the sblock matrix.

Description

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

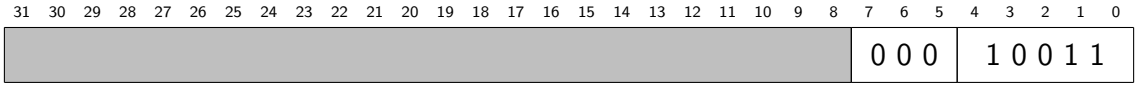
Notes

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

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

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

Readback



Format

readback()

Purpose

To read back cell states from the sblock matrix.

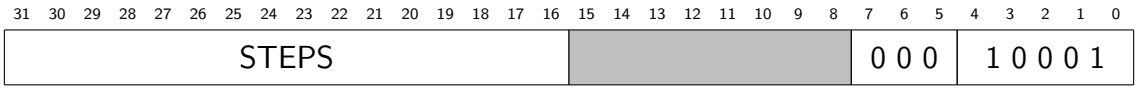
Description

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

Notes

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

Step



Format

step(STEPS)

Purpose

To perform updates of the sblock matrix.

Description

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

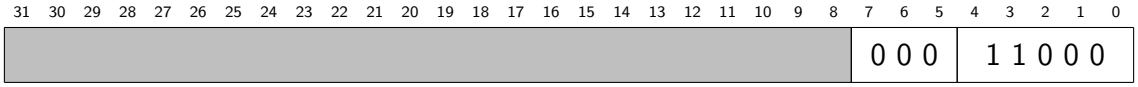
Notes

This instruction takes STEPS + 1 cycles.

6 Control Flow Instructions

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

Break



Format

break_out()

Purpose

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

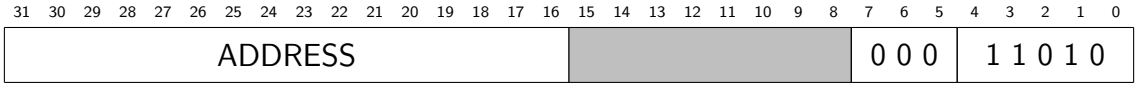
Description

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

Notes

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

Store



Format

store(ADDRESS)

Purpose

To begin storage of a program to internal memory.

Description

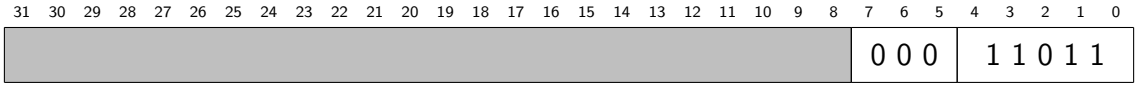
The Fetch module exits [read from communication] mode and enters [save to memory] mode. The next instruction will be saved at address ADDRESS, and then each address thereafter.

Notes

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

ADDRESS is cropped to [program counter bits].

End



Format

end()

Purpose

To end storage of a program to internal memory.

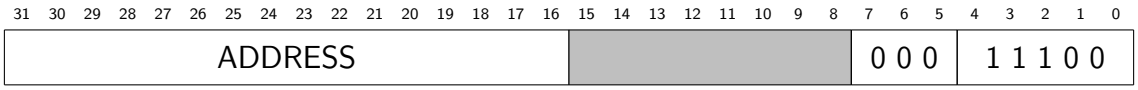
Description

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

Notes

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

Jump



Format

jump(ADDRESS)

Purpose

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

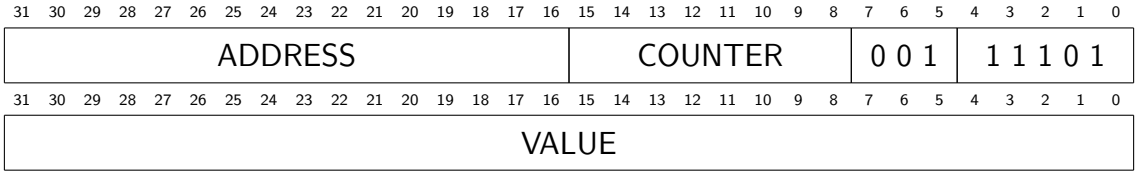
Description

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

Notes

ADDRESS is cropped to [program counter bits].

Jump Equal



Format

jump_equal(ADDRESS, COUNTER, VALUE)

Purpose

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

Description

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

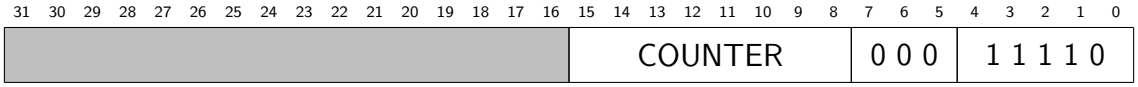
Notes

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

ADDRESS is cropped to [program counter bits].

VALUE is cropped to [counter bits].

Increment Counter



Format

counter_increment(COUNTER)

Purpose

To increment a counter.

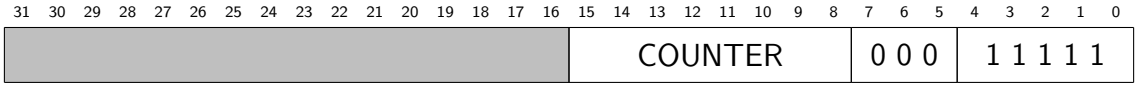
Description

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

Notes

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

Reset Counter



Format

counter_reset(COUNTER)

Purpose

To reset a counter.

Description

Counter COUNTER is set to 0.

Notes

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

Appendix D

Specialization Project Report

The Cellular Automata Research Platform: Communication and Verification

Per Thomas Lundal – Supervised by Gunnar Tufte

***Abstract*—Research into evolvable hardware has long been performed at NTNU. In 2003, a cellular automata research platform was created with the use of an FPGA.**

In expectation of new hardware with a larger FPGA, the platform was updated and extended in 2014. It was optimized to provide a speedup of 4 for most operations by taking advantage of increased resources. However, the new PCI Express interface was not implemented and the design was only verified in simulation.

The goal of this project was to finalize the platform, such that the new features and improved speed could be taken advantage of. The first objective towards that end was to replace the old communication interface. The second was to verify that the new design worked correctly when implemented in hardware.

This paper details the implementation of the new communication interface, both in hardware and software. Then, the previously upgraded design is implemented and tested in the new hardware.

Implementation of the new communication interface is a success. However, testing shows that there are a lot of issues with the upgraded design. Some issues were possible to fix within the alotted time, but several critical remain.

I. Introduction

Evolvable hardware (EHW) is a field of science where evolutionary algorithms (EAs) are used in the creation of specialized hardware. EAs simulate mechanisms found in biological life, such as selection, reproduction and mutation. The goal is to optimize fitness, the ability to survive in the competition for being the best solution. EAs can quickly find approximate solutions to hard problems, and then gradually refine them into good solutions.

An interesting feature is that EAs can find ways to exploit hardware in ways that human designers cannot comprehend [1]. This can be due to complex parallel interactions, or usage of properties that are not fully understood [2].

Technologies related to EHW, including a common EA and hardware platforms are presented in Section II.

Evolvable hardware has been an area of research at NTNU for more than a decade. In 2002, NTNU invested in dedicated FPGA hardware with the intent of building an EHW platform. The purpose was to create a platform for experimentation with evolution and development within a cellular automata (CA).

The initial work was done by Djupdal, before being extended by Aamodt. The CA was implemented as a matrix of sblocks, reprogrammable CA cells designed specifically for usage with evolution. It is connected to a development unit which can simulate growth and change in the sblock matrix. The hardware platform is connected to and controlled by a computer over a PCI connection.

A general use-case for the platform is to have the computer run a genetic algorithm, where the genotype represents the initial CA state and development rules. Then, the CA is initialized and developed to produce a phenotype, which is executed to produce a fitness value.

In expectation of new hardware with a larger FPGA and faster PCI Express connection, Støvneng refurbished the design in 2014. He took advantage of the added resources to greatly improve the performance of the platform, giving a speedup of 4 or more for most operations. Additionally, he extended the CA into 3D and added a DFT. However, since the hardware did not arrive in time, the new design was only tested in simulation and the communication interface was not upgraded.

An in-depth explanation of the platform's features, functionality and iterations is presented in Section III.

The task of this project is to finish the new platform by implementing a new PCI Express communication unit, and to verify that everything is functional in hardware. This will allow the new platform, which is both faster and more feature-rich, to be taken into use. The motives are further detailed in Section IV.

The implementation of the new communication unit is detailed in Section VI, while the verification process and results are detailed in Section VII.

Challenges related to setting up the new hardware platform and testing is detailed in in Section VIII, along with proposed design changes and other future work.

Section IX concludes this paper.

II. Technology

The field of EHW is comprised of many technologies, but only a few are relevant for this paper. Those are genetic algorithms, development, cellular automata, FPGAs and sblocks. Additionally, PCI Express is relevant for the new communication unit.

A. Genetic algorithms

A genetic algorithm (GA) is a very common type of EA. It represents each solution as a genotype, a binary string used as a blueprint to create the solution itself, called a phenotype. The genotype is comparable to nature's DNA, and it is this genetic material which is modified in the evolutionary process.

The GA process is shown in Fig. 1. First, a base population with random genotypes is generated. Then, each

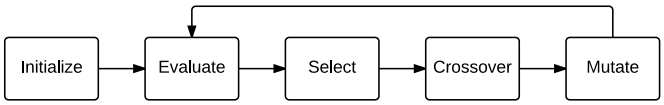


Fig. 1. A genetic algorithm. The cycle is broken when the fitness is above a given threshold.

phenotype is constructed and evaluated using a fitness function. If a solution has a fitness score above a set threshold, the process stops. Otherwise, a new population is created by selecting solutions with high fitness scores, crossing their genotypes, and mutating the results, before repeating the process.

B. Development

The process that transforms the genotype into a phenotype is called development; it can be regarded as a form of decompression algorithm [3]. In nature, this process is seen when a fertilized egg transforms into a multicellular organism. A conseptual example is shown in Fig. 2.

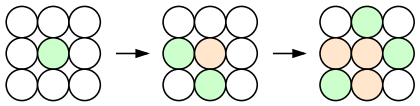


Fig. 2. An example of one cell developing into six. Additionally, some cells change type.

Development also allows individuals to adapt to their environments, making them more robust and scalable [4].

C. Cellular automata

A cellular automaton (CA) is a structure made up of vast numbers of very simple computational units called cells. The cells are arranged in a grid, with communication only permitted between nearby cells according to a neighborhood. The von Neumann neighborhood is common for two-dimensional CAs; It includes the cells to the north, south, east and west, along with itself (center). An example of this is shown in Fig. 3.

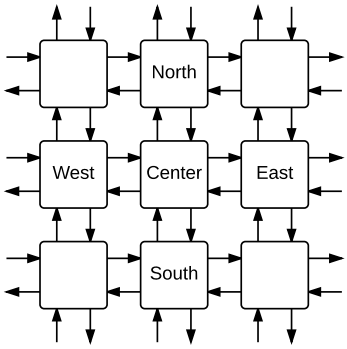


Fig. 3. An excerpt from a 2D CA using a von Neumann neighborhood. All outputs from a given cell carries the same value.

Each cell has a state, which is often a binary number where 1 represents alive and 0 represents dead. At each discrete time step, each cell updates its state based on the states in its neighborhood. The update function is often specified as a look-up-table (LUT), where the next state is defined for all possible neighborhood states[1]. If all cells implement the same update function then the CA is uniform, otherwise it is non-uniform.

The update functions determine how complex patterns and structures emerge within the CA. The emergent behavior can be categorized into one of four classes [5].

1) Homogeneous state.
2) Simple periodic structures.
3) Chaotic patterns.
4) Complex patterns and structures.

The latter is the most interesting, providing both the complexity required for computation and structures required for storage. It exists in the phase change between class 2 and 3, known as the edge of chaos [6].

CAs have been shown to be Turing complete [7] [8], which means they are able to perform any kind of computation.

D. Field Programmable Gate Array

A Field Programmable Gate Array (FPGA) is a type of reconfigurable hardware. It can implement any desired logical operation by configuring and connecting a number of look-up tables (LUTs) and flip-flops (FFs). FPGAs can also contain dedicated blocks for addition, multiplication, memory, and other functions. These elements are grouped into configurable logic blocks (CLBs), which through a network of interconnects can be connected to each other or input/output pins. An example of this structure is shown in Fig. 4. Note that modern FPGAs consists of thousands of CLBs and hundreds of I/O pins [9].

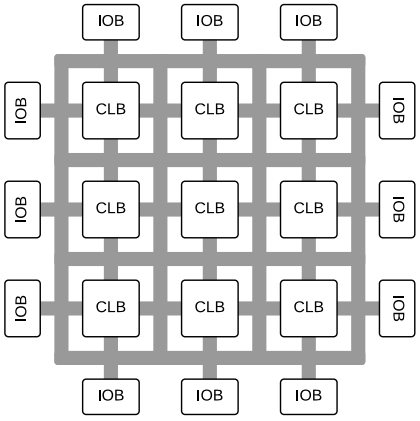


Fig. 4. High-level block diagram of an FPGA. An array of configurable logic blocks (CLBs) and input/output blocks (IOBs) are connected by a network of interconnects.

FPGAs have been the subject of EHW research due to their reconfigurability, and several researchers have been successful in evolving working electronic circuits [10] [1].

[1] The length of the LUT increases exponentially with the size of the neighborhood. $L = S^N$ where L is the LUT length, S is the number of possible states and N is the number of cells in the neighborhood.

However, the resulting circuits have often ended up using intrinsic properties of the silicon and been very sensitive to environmental changes.

The trouble with using modern FPGAs for EHW research is that some configuration bitstrings can destroy the FPGA [11] [12]. This means that the bitstring can not be used directly as the genotype without complicated tests to discard the dangerous bitstrings.

E. Sblock

The sblock was introduced as part of a new EHW-friendly FPGA architecture in [13]. The architecture is a non-uniform CA with a von Neumann neighborhood, where the update function of each cell is independently configurable at run-time. The cells, known as sblocks, are very simple structures; they consist of a configurable look-up-table (LUT) and a flip-flop (FF), as shown in Fig. 5.

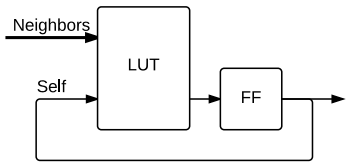


Fig. 5. Detailed block diagram of an sblock. The LUT can be reconfigured on-the-fly to implement any logical function.

The greatest benefit of using sblocks for EHW research is that there is no risk of damage or exploitation of intrinsic properties in the silicon. Additionaly, the simple structure and hardwired signal routing allows for very efficient area usage. The likelihood of a mass-produced sblock-FPGA arriving on the market in the near future is slim. However, it is possible to implement it virtually within an other FPGA.

F. PCI Express

The PCI Express interface was designed to tackle the arising trouble with clocked parallel buses like PCI. The problem with such buses is that the clock speed can not be increased beyond a given threshold, as the slightly different lengths of the wires causes data to arrive at slightly different times. Reducing the clock period to less than the variation in arrival time means the data will become corrupted. This problem is exacerbated with increasing bus size.

PCI Express is therefore based on serial communication over differential pairs (lanes[2]) without the need for a reference clock [14]. This allows an extremely fast clock speed compared to a parallel bus, and much greater bandwidth in total. PCI Express consists of three layers; the physical layer, the data link layer and the transaction layer, structured as shown in Fig. 6.

[2] PCI Express operates in full duplex mode, which means that each lane has an independent differential pair in each direction. 1, 2, 4, 8, 16 or 32 lanes are supported, but data is striped and thus still transmitted serially.

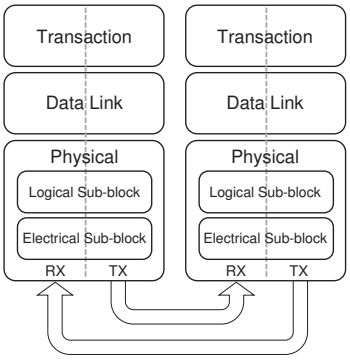


Fig. 6. High-level diagram showing the layered structure of PCI Express. (Reprinted from [14])

The transaction layer's primary responsibility is the creation and parsing of transaction layer packets (TLPs). TLPs are used to trigger events or start various transactions, most commonly to initiate read and write requests[3]. Most requests entail the return of a completion TLP containing the requested data or other information. TLPs consists of multiple 32-bit double words (DW), where the first is a common header describing the type of packet.

The data link layer ensures integrity by adding error detection codes to outgoing TLPs and performing error detection and correction on incoming TLPs. It is also responsible for retransmission if corruption occurs.

The physical layer is responsible for serialization and deserialization of the data stream. Each byte is padded with two extra bits (8b/10b encoding) to allow clock recovery.

III. Previous Work

The Cellular Automata Research Platform (CARP) has been the subject of three previous master theses at NTNU. The original implementation was made by Djupdal in 2003. It was then extended with a range of various output methods by Aamodt in 2005. Finally, it was further extended and optimized in expectation of new hardware by Støvneng in 2014.

A. Conception

In 2002, NTNU invested in a CompactPCI computer with a NallaTech BenERA FPGA board to be used for research within the field of evolutionary hardware. The task of developing a platform for the system, based on a matrix of sblocks, fell to Djupdal [15].

An overview of the resulting hardware platform is shown in Fig. 7. It consists of the mentioned sblock matrix, block RAM (BRAM) for storing the state and type of each cell, a development unit, control logic, and a PCI communication unit.

The system is meant to be controlled by a computer running a genetic algoritm. A common flow of operation

[3] Read and write requests are directed at one of up to six base address registers (BARs). They represent internal memory areas that can be anywhere from a few bytes to several gigabytes in size.

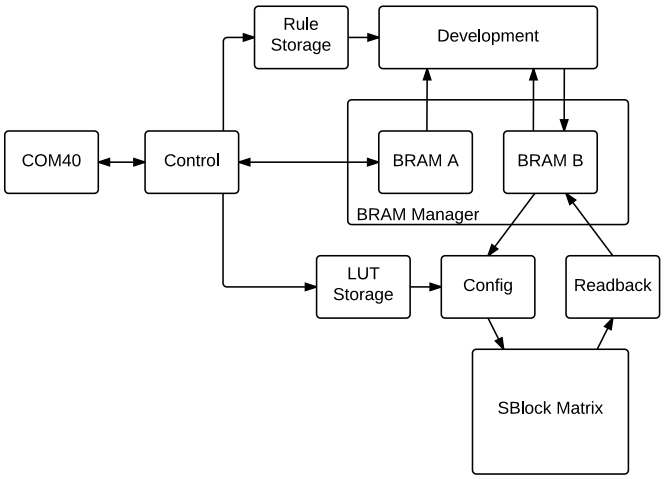


Fig. 7. High-level block diagram of the hardware platform after Djupdal's original work.

is to initialize the system with the genotype, develop it into its phenotype, run the SBM, and send the new states back to the computer. The computer then uses the newly received state data to calculate a fitness score.

The system is initialized by writing states and types to BRAM A, in addition to storing development rules and LUT conversion rules. Then a development step can be performed by reading cell types from BRAM A[4], testing development rules, and writing the (possibly changed) types back to BRAM B. The development unit tests 8 rules on 2 cells each cycle in raster order. Optionally, the BRAMs can be logically swapped and further development steps performed. The SBM can then be configured by translating the types in BRAM B into LUT entries according to the LUT conversion rules, before being run for a desired amount of cycles. Afterwards, the new states in the SBM can be read back into BRAM B, swapped into BRAM A, and sent to the computer.

The design is split into two clock domains; the communication unit uses 40 MHz to be able to interface with PCI, while the rest uses 80 MHz for higher performance.

B. Extension

There was one major bottleneck in the original design. To calculate the fitness of an individual, the state of each cell had to be transferred to the computer over the PCI interface. Having a dedicated hardware unit would greatly improve the performance. Additionaly, it was desired to have more information about the development process. The task of realizing this fell to Aamodt [16].

An overview of the hardware platform with Aamodt's additions is shown in Fig. 8. The additions consists of a run-step function that calculates the number of live cells, BRAM to store the numbers, a fitness function, and two information outputs from the development unit.

[4] After the first 8 rules have been tested on all cells, center cell types are read from BRAM B instead. This is needed to prevent the result of a rule in an earlier iteration from being deleted if no rules trigger in a later iteration.

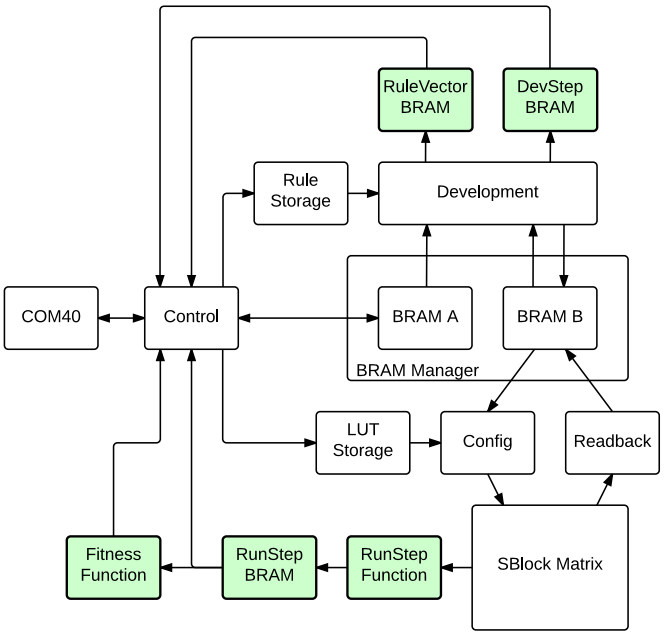


Fig. 8. High-level block diagram of the hardware platform after Aamodt's work. Additions are highlighted in green.

The rule vector BRAM stores lists of which rules were triggered and not for the last 256 development steps. The lists are implemented as bit-vectors where each bit represents the status of a rule for a single development step. The development step BRAM is more detailed; it stores which rule was triggered for each cell. However, it only has storage space for one development step.

The run-step function calculates the number of live cells after each SBM update by using a large adder tree. The numbers are stored in run-step BRAM for later usage by the fitness function, which is replacable.

C. Renovation

In expectation of receiving new hardware with a larger and faster FPGA, there was a demand to optimize the platform by taking advantage of the increased resource pool. Extending the platform into the third dimension was also a lucrative thought, as doing so allows more complex signal pathways to form within the cellular automata. It was also desired to have a discrete fourier transform (DFT) for interpretation of the RSF data; it should give very useful data according to Berg's research [17]. The task of realizing this was taken on by Støvneng [18].

An overview of the hardware platform with Støvnengs additions and optimizations is shown in Fig. 9. The only addition is the DFT, but nearly all units has been optimized, yielding a speedup of 4 for most operations.

Unfortunately, due to some challenges with manufacturing, Støvneng was unable to get hold of the new hardware for the duration of his project. The system was therefore only verified in simulation, and the PCI communication unit was not upgraded for the PCI Express connection on the new board.

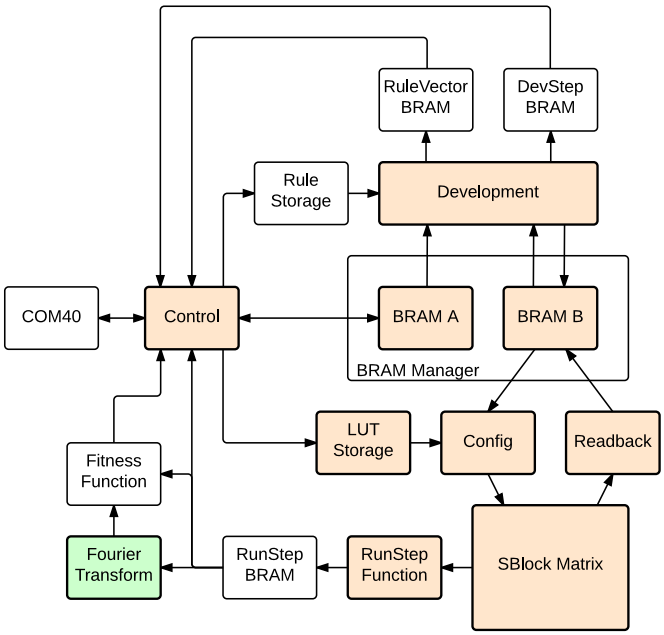


Fig. 9. High-level block diagram of the hardware platform after Støvneng's work. Additions are highlighted in green, and optimizations and 3D modifications in orange.

IV. Motivation

Currently, CARP is only usable in simulations, which are extremely slow and therefore practically unusable. Implementing a PCI Express communication unit should make the new platform operational, allowing it to be used for research.

The new platform is about 4 times faster than the old and support larger sblock matrices. It also has the new and exciting DFT, which is shown to produce very useful data. 3D is also exiting as it allows much more complex communication pathways to form within the CA.

Verification of previous modifications are necessary since it was only tested in simulation. The path from a behavioral description to an implementation using LUTs and FFs is long and complex; what functions in simulation does not necessarily function when implemented.

V. Development Platform

Multiple weeks into this project, several months after the end of Støvnengs project, there were still no signs of the new hardware. To prevent the project from halting dead in its tracks, a decision was taken to order slightly different hardware. The significant difference to the original system is reduced size of the FPGA, a Spartan-6 LX45T instead of a Spartan-6 LX150T, which entails around 70% less available resources. Luckily, the hardware design can be scaled down to fit the smaller chip by reducing the size of the sblock matrix, allowing for implementation of PCI Express and verification of the complete system in hardware.

A. Spartan-6 SP605 Evaluation Platform

The Spartan-6 SP605 Evaluation Platform is essentially a board with the Spartan-6 LX45T FPGA wired to every useful peripheral imaginable. It has connections for PCI Express[5], ethernet, DVI, USB, flash card, JTAG, LEDs, switches, and more. However, the only peripherals utilized in this paper are PCI Express, and JTAG. An overview of the system is shown in Fig. 10.

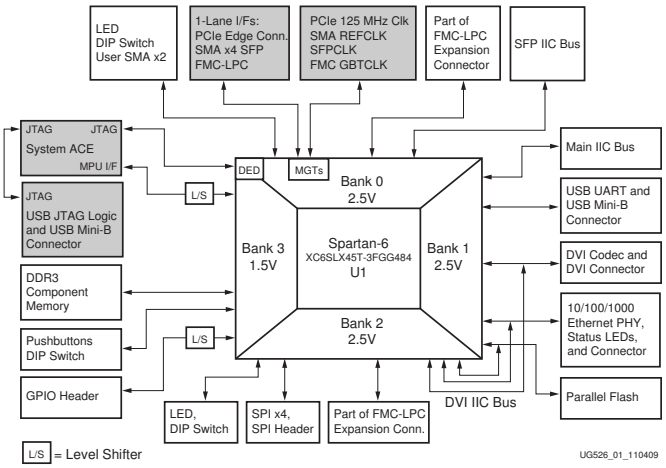


Fig. 10. High-level block diagram of the SP605 and its peripherals. Peripherals utilized in this paper are highlighted in gray. (Modified reprint from [19])

The switch and jumper configurations of the SP605 are set to factory defaults, with the exception of SW1 which is set to 10.

B. Hardware setup

Due to the experimental nature of testing a new hardware platform, two computers were used in this project, as shown in Fig. 11. One is the main development workstation, used for coding and synthesis; it has a JTAG connection to the SP605 over USB, which allows it to upload new designs. The other is the host for the SP605, which is mounted in a PCI Express expansion slot.

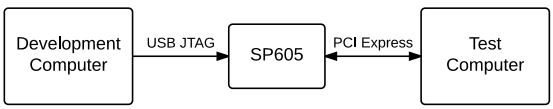


Fig. 11. High-level block diagram of the hardware setup.

The setup allows a new design to be uploaded and tested on the SP605 without disrupting the workflow of the main workstation, due to the power-cycle required to reset the PCI Express connection.

C. Software setup

The operating system on both computers is Linux Mint; version 16 on the development computer and 17 on the test

[5] Even though the PCI Express finger has lines for power, they are not connected on the SP605. This means an external power source has to be connected.

computer. Linux Mint is currently one of the most popular linux distributions, along with Ubuntu, which it is based upon [20]. This means that procedures and software used and created in this paper should work on most systems.

Xilinx ISE version 13.3 was used for hardware design and synthesis, while ISim was used for simulations. The third-party USB cable driver from [21] was used for JTAG, as explained in Section VIII-A. The software API was compiled with GCC version 4.8.2.

VI. Implementation

Fig. 12 shows the changes to the hardware platform. The old COM40 unit has been replaced by a new communication unit and a compatibility layer. It communicates with a new software api which uses Linux' built-in drivers for PCI Express.

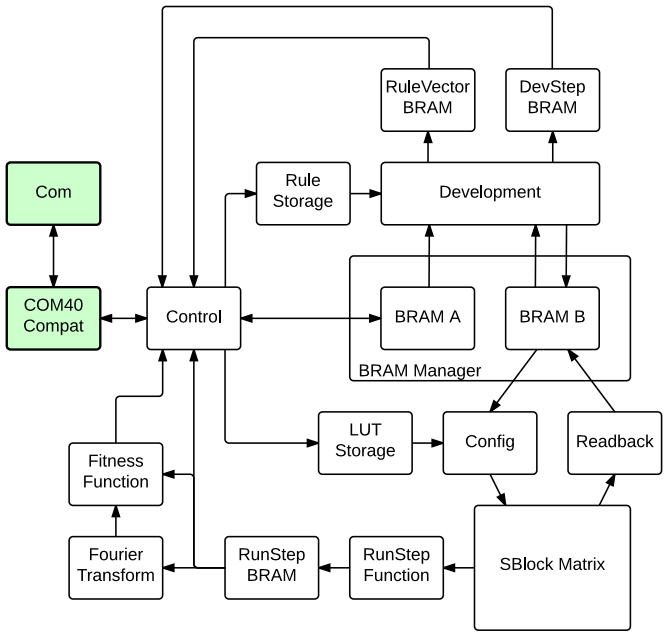


Fig. 12. High-level block diagram of the current hardware platform. Additions are highlighted in green.

A. Detailed overview

The new communication unit is based on Xilinx' reference PCI Express programmed input/output design. It consists of the Xilinx PCI Express endpoint core, reception and transmission engines, data buffers, and a special request handler, as shown in Fig. 13.

The endpoint core completely handles the physical and data link layers, and all TLPs related to configuration and establishment of the PCI Express connection. Other TLPs, such as read and write requests, are presented on an AXI4-Stream interface [22]. The reception engine is responsible for parsing TLPs and either writing received data to the reception buffer or notifying the transmission engine about a read request. The transmission engine is responsible for building completer TLPs to respond to read requests, using data from the transmission buffer.

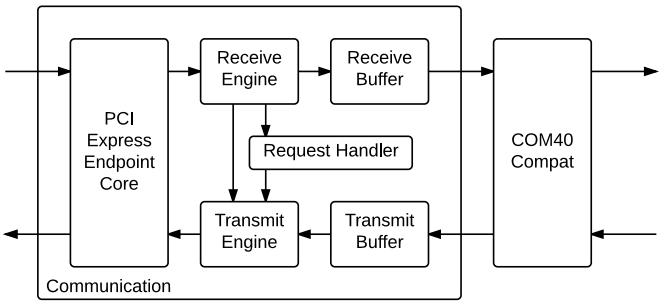


Fig. 13. Detailed block-diagram of the PCI Express communication module.

The request handler listens to the read requests provided by the reception engine, and can override the transmission engine to respond to special requests.

B. PCI Express Endpoint Core

Several Spartan-6 FPGAs, including the one used in this project, contain a special-purpose hardware block for implementation of PCI Express. The block completely handles the physical and data link layers, with the transaction layer left for the user.

To make use of the block, Xilinx provides the Spartan-6 Integrated PCI Express Endpoint Core; version 2.3 was used in this project. This core additionally takes care of all TLPs related to configuration of the PCI Express connection. Other TLPs, such as read and write requests, are presented on an AXI4-Stream interface [22].

The endpoint core is configured with two memory regions, both 4 kB in size[6]. The first memory region (BAR0) is used for normal communication, while the second (BAR1) is used for special requests. The separation is mostly conseptual as both regions are treated as one data stream. The difference is that the special request handler kicks in for read requests to BAR1.

C. Reception engine

The reception engine is implemented as a simple state machine, as shown in Fig. 14.

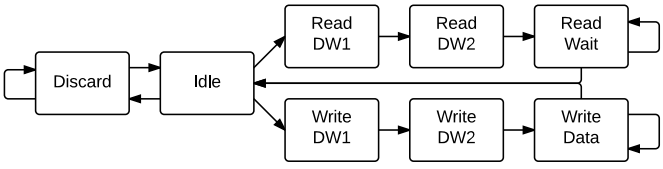


Fig. 14. State machine for the reception engine.

Until the endpoint core presents valid data, the state machine remains in Idle. When it does, the data is stored, and the TLP type is checked. If it is a read or write request, the state machine continues down the corresponding path, otherwise the remaining data is discarded. The remaining

[6] The smallest memory region that can be memory-mapped is one page. The default page size in Linux is 4 kB.

portion of the TLP headers are then parsed in the DW1 and DW2 states. For read requests, the state machine waits in ReadWait until the transmission engine is ready to accept a new read request, and then proceeds to Idle. For write requests, the state machine stays in WriteData, where one DW of data is written to the reception buffer each cycle, for the length of the packet, and then proceeds to Idle.

D. Transmission engine

The transmission engine is implemented as a simple state machine, as shown in Fig. 15.

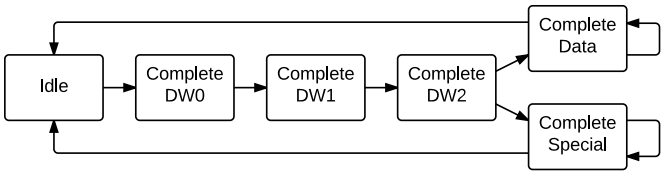


Fig. 15. State machine for the transmission engine.

Until the reception engine signals a read request, the state machine remains in Idle. When a read request is signaled by the reception engine, the state machine begins to traverse the DW path. The DW0, DW1 and DW2 states each transmit one DW of the completer TLP header. Then if the special request signal is set, it procceds to CompleteSpecial, where it transmits data presented by the request handler. Otherwise, it proceeds to CompleteData where it transmits one DW of data from the transmission buffer each cycle. When the requested number of DWs has been transmitted it proceeds back to Idle.

E. Request handler

The request handler continually listens to the read requests presented by the reception engine. If the request is targeting the primary memory area (BAR 0), it is a normal read request and the transmission engine is allowed to proceed as usual. Otherwise, it is a special request and the transmission engine is overridden.

The kind of special request is determined by the address of the read request, and handled thereafter. There are currently four special requests implemented, as shown in Table I.

TABLE I
SPECIAL REQUESTS

Address	Request
0x00	Get transmission buffer data count
0x01	Get transmission buffer available space
0x02	Get reception buffer data count
0x03	Get reception buffer available space

Note that each of the implemented special requests assumes a read request length of one DW. If the request has a greater length, the returned data is simply repeated to fill the packet.

F. Buffers

The buffers are implemented as first-in first-out (FIFO) queues using block RAM (BRAM) and two counters. The counters determine the addresses that are written to and read from, and are incremented when the write or read signals are asserted. Fig. 16 shows how the FIFO is used to buffer two words.

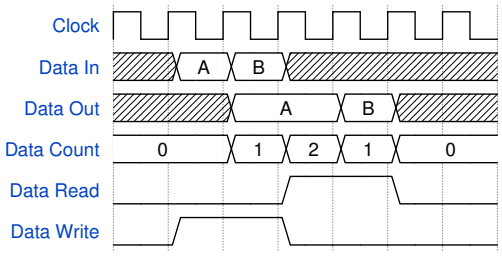


Fig. 16. Wave diagram for the FIFO buffer, showing two consecutive writes followed by two consecutive reads.

Notice how the read signal needs to be asserted before the clock tick when data is read to ensure correct consecutive reads. This is due to the BRAM used in the FIFO, which updates at clock ticks. To have correct data available for a read in the following cycle, the address therefore has to be updated before the clock tick (by asserting the read signal).

G. Compatibility layer

Due to the difference in wordsize between the new and old communication units (32 vs 64 bits), a compatibility layer is added between the communicaton unit and the control unit. This allows the control unit and the rest of the design to remain unchanged, making the replacement transparent.

The compatibility layer contains two very simple state machines. One combines two 32-bit words from the communication unit into 64-bit words for the control unit. The other splits 64-bit words from the control unit into two 32-bit words for the communication unit.

H. Software API

The communication part of the new software API is split into two parts.

The first is a general interface for connecting to PCI and PCI Express devices without using a custom driver. It takes advantage of Linux' automatic population of /sys/devices/pci* with files representing the memory regions of all PCI and PCI Express devices. The directory is searched by vendor and device id, and the corresponding memory regions is memory-mapped into the program.

The second is an interface specifically for the communication unit. It provides open, close, read and write functions similar to the old BenERA interface, in addition to implementing all special request functions in Table I. When a read or write operation is initiated, buffers are checked for available data or space. If there is not enough present, the program waits and then rechecks.

VII. Verification

Due to lack of hardware, Støvneng only verified his changes in simulation. With available hardware and an updated communication unit, the design can finally be properly verified.

A. Methodology

The communication unit was tested in hardware by connecting the output of the reception buffer to the input of the transmission buffer. Then sample data was sent over PCI Express to the SP605 and then read back.

The updated hardware design was tested with SBM sizes of 8x8 for 2D and 8x8x8 for 3D. The tests are detailed in Appendix A; each one has a short description and a list of the instructions it verifies. Together, the 11 tests cover all instructions.

Note that due to some instructions being dependent on others, it is not always possible to know which instruction is failing.

B. Results

The communication unit returns the correct data in the correct order, which means it passes its test.

The 2D design passes all tests except for 6 and 8, while the 3D design failes test 1, 3, 6, 7 and 8. This means that the two most crucial components, the SBM and development unit, is working in neither. The 3D design also has some additional issues.

The status of each instruction is listed in Table II, while descriptions of the issues are listed in Section VII-C.

Some issues were solved in the alotted time. They are not included in these results, but are listed in Section VII-D for completeness.

TABLE II
Implementational status of instructions

Instruction	Works in 2D	Works in 3D
break	Yes	Yes
clearBRAM	Yes	Yes
config	Yes	Undecidable
devstep	Undecidable	Undecidable
doFitness	Undecidable	Undecidable
end	Yes	Yes
jump	Yes	Yes
jumpEqual	Yes	Yes
nop	Yes	Yes
readback	Yes	Undecidable
readFitness	Undecidable	Undecidable
readRuleVector	No	No
readState	Yes	Yes
readStates	Yes	Yes
readSums	Undecidable	Undecidable
readType	Yes	Yes
readTypes	Yes	Yes
readUsedRules	Undecidable	No
resetDevCounter	Yes	Yes
run	Undecidable	Undecidable
setNumberOfLastRule	Undecidable	Undecidable
startDFT	Undecidable	Undecidable
store	Yes	Yes
switch	Yes	Yes
writeLUTConv	Undecidable	Undecidable
writeRule	Undecidable	Undecidable
writeState	Yes	Yes
writeStates	Yes	No
writeType	Yes	Yes
writeTypes	Yes	No

C. Remaining issues

ReadStates prints garbage in top 32 bits. This occurs due to a buffering error in the LSS unit. However, only the least 32 bits are used by the api, which means that this issue only impacts performance and not functionality.

WriteStates and WriteTypes does nothing in 3D. This issue is only present on the board, not in any simulations, which makes it hard to track down the cause.

ReadRuleVector sends incorrect data. The first execution of the instruction produces an extra word; a repetition of the first. Following executions produces the correct amount of words, but the order is offset by one.

ReadUsedRules fails for SBM widths less than 16. The simulator crashes with an index-out-of-bounds error, due to the instruction treating a $[width \cdot 4]$ bits wide signal as if it is 64 bits wide. How ISE is able to implement the design despite illegal indexing is a mystery, but the instruction produces only zeroes when executed on the board.

Development rules does not activate. This issue is also only present on the board, making it hard to analyze. The root cause could be with any of the following instructions: devstep, writeRule and setNumberOfLastRule.

Config does not properly write states in 3D. Simulations show that the BRAM address fluctuates, causing the states to be overwritten by 0 in the following cycle.

Runstep causes every state to become zero. When a runstep is performed, all states in the SBM are reset.

D. Solved issues

States and types were written to the wrong location. When writing single states or types, a half-row is read from BRAM, combined with the new data, and written back to BRAM. Due to the usage of non-implementable code to specify how the data should be combined, the new data always ended up in the middle of the half-row.

Development ran indefinitely. Comparison of signals of different widths always return false. Due to the comparison of a parameterized signal with a constant, the development unit would not iterate through the cells, and thus never finish.

WriteRule did not follow spesification. When the api transformed a 3D rule struct into an instruction, the position of up/down and north/south were swapped.

PrintTypes used wrong offsets for decomposition. When decomposing a row of types into individual types, an offset

of 5 was used instead of 8. This entailed that the printed values appeared as garbage.

ReadVector and PrintVector used 32-bit words. The functions were not updated from using 32-bit to 64-bit words. ReadVector would therefore expect twice the number of words provided by the hardware platform, causing the program to wait for nonexistant data.

VIII. Discussion

A. Challenges

There was a lot of concern during initial hardware testing, as the SP605 was not detected by the computer. A slightly curved circuit board led to the belief that there might be something wrong with the hardware. Luckily, it proved not to be a hardware fault, but a mistake in the hardware setup guide; the position of SW1 was reversed, causing the board to operate in a completely different mode.

The SP605 was pre-installed with an example design implementing communication over PCI Express with DMA. However, the accompanying driver did not support newer Linux kernels. Additionaly, the design was written in verilog while CARP is written in VHDL, which meant extra effort to integrate the two. There was some effort applied to update the driver, but it was abandoned due to near-untraceable segfaults.

The USB cable driver for usage of JTAG provided by Xilinx also had the problem of not being compatible with newer Linux kernels. Thankfully, a third-party driver found at [21] is compatible and solves the problem.

For unknown reasons, collisions occur on vital signals in all post-map and post-place-and-route simulations[7], causing them to be of no use. This makes it impossible to analyze issues that are present in implementation but not in post-translate simulation. Since ISE will not respect the *keep_hierarchy*[8]attribute for the unit in which the collision is first observed, tracing of the source has been unsuccessful.

B. Future work

The most important thing going forward is to fix the errors that are preventing the sblockmatrix and development units from working correctly. However, this is no easy task, as large parts of the design are highly complex and difficult to debug. The most extreme cases are the development and LSS units which each consists of a single large file, around 1200 lines long, of complex pipelined code. Simplification and modularization of these units is therefore imperative.

Another reason to simplify the development unit is it's extreme memory bandwidth requirement against the SBM BRAM. Currently, it is designed so that N rules are loaded, applied to every cell, then the N next rules loaded and so on. This means that the BRAM must supply 5 rows each cycle to test 1 row per cycle (or 8 for 2) in 3D, while N rules are needed per matrix iteration. In addition, after the first pass, center cells has to be read from BRAM B instead of BRAM A, since they might have changed.

A simplified process is to read 5 rows, apply one rule to the center row each cycle, then read the next 5 rows, and so on. This will greatly lessen the bandwidth requirements against the BRAM, as new rows can be read in sequence while each rule is being applied to the current. Assuming there are more rules than the number of rows that must be read (highly likely), there is no performance loss. Additionally, this would allow development to only read from BRAM A, simplifying the dataflow.

There are still some remains of having two clock domains, more specificaly a pair of flipflops used for clock-synchronization in the fetch and lss units. It does not affect functionality, and has therefore been of low priority, but it does add a slight delay between the communication unit and the fetch and lss units when reading data.

Currently, the platform only supports SBM sizes that are powers of 2. It would be beneficial to be able to select any size, allowing for fine-tuning of the resource usage, to get most out of the FPGA.

As noted in [18], Støvneng increased the base instruction and data sizes from 32 to 64 bits. Although it is one way to accomodate for longer instructions, the decision is a little odd, considering both PCI and PCI Express are based on 32-bit word sizes. This means that conversion is currently required between the LSS and communication units. Since only 6 out of 30 instructions require more than 32 bits, communication could be simplified and optimized by going back to a 32-bit base size.

The current design makes use of a lot of internal tristate buffers. These are not supported in modern FPGAs [23], and therefore need to be converted into other logic during synthesis. The synthesis tool gently hints at this misuse by producing several warnings. Removing tristate buffers will therefore result in code that more closely relates to its implementation.

Another feature that can beneficially be removed is the global asynchronous reset. Since all Xilinx FPGAs start in a well defined state, a reset signal is only needed in very spesific cases [24] [25]. Otherwise, it only serves to take up valuable resources.

Having the software interface automaticly adapt to the implemented hardware platform would be nifty. This would remove the need to specify the sblock matrix size and type at compilation. It could be accomplished by adding an instruction that returns the size of the sblock matrix.

A feature that could be interesting is the ability to enable or disable wrap-around for the edges of the sblock matrix. Disabling it would mean that the size of the matrix can not be exploited to create an oscillating signal by something continually moving in one direction.

[7] The order of the implementation process is: Synthesize, translate, map, place-and-route.

[8] The *keep_hierarchy* attribute informs the synthesis tool that it should not flatten hardware design units to allow further optimizations. This is useful since the opimizations makes it near-impossible to trace signals.

Finally, a unification of the 2D and 3D designs would give a more generic design and allow less code to be maintained.

With the current need for major fixes, simplification of development and LSS, reducing the need for extreme memory bandwidth, removing tristates, removing resets, and unification, it might be a good idea to rebuild the platform from the ground up. Starting from a clean slate, thoroughly evaluating every part of the design, replacing the bad features and improving the good, will likely result in a greatly improved platform for CA research.

IX. Conclusion

In this paper, a new hardware platform has been set up. A PCI Express communication unit has been designed and implemented in hardware. A corresponding api has been created in software. It has all been successfully integrated into the the existing platform.

Unfortunately, verification shows that the recently upgraded design has a lot of issues. Some issues could be fixed within the alotted time, but several critical remain. Neither the sblock matrix nor the development unit is functional, which means the platform is currently unusable. These issues along with a long list of proposed changes suggest that a major rewrite might be in order.

Appendix A
Test Descriptions

Test	Description	Verifies
0	Write and read single types	WriteType, ReadType
1	Write and read multiple types	WriteTypes, ReadTypes
2	Write and read single states	WriteState, ReadState
3	Write and read multiple states	WriteStates, ReadStates
4	Write states and types, clear BRAM, check BRAM is empty	ClearBRAM
5	Write states and types, switch BRAM, check data is gone, switch again, check data is back	SwitchSBM
6	Write rules and types, run devstep, check rules have triggered and types have updated	DevStep, WriteRule, ReadRuleVector, ReadUsedRules
7	Write and read state to/from sblockmatrix	Config, Readback
8	Write states, types and LUTConv, run sblockmatrix, check states have changed	Run, WriteLUTConv
9	Store program that prints 1 and then stops, jump to program address 3 times, check for three 1's	Store, End, Jump, Break
10	Execute program that prints 1, runs devstep and jumps to itself unless 3 devsteps has run, check for three 1's	JumpEqual, ResetDevCounter

Appendix B
Attached Files

Directory structure:

- hardware
  - 2D
  - 3D
  - common
    - communication
    - inferers
    - utility
  - ipcore_dir
  - sp605
- software
  - 2D
  - 3D
  - common

New files:

- hardware/common/communication
  - *com40_compatibility_layer.vhd*: The compatibility layer between the control unit and the communication unit.
  - *communication.vhd*: The new communication module.
  - *communication_sim.vhd*: A "fake" communication module that provides external access to the buffers, used to allow simulation.
  - *rx_engine.vhd*: The reception engine, responsible for parsing TLPs.
  - *tx_engine.vhd*: The transmission engine, responsible for building TLPs.
  - *rq_special.vhd*: The special request handler.
- hardware/common/utility
  - *combiner.vhd*: A module for inserting an entry into a word, used in the control unit to fix the WriteState and WriteType bug.
  - *fifo.vhd*: A fifo buffer.
  - *shifter.vhd*: A static shifter, used in the dynamic shifter.
  - *shifter_dynamic.vhd*: A dynamic shifter, used in the combiner.
- hardware/ipcore_dir
  - *sp605_pcie.xise*: Project file for the Spartan-6 PCI Express endpoint core.
  - *sp605_pcie.xco*: Core generator file for the Spartan-6 PCI Express endpoint core.
- hardware/sp605
  - *constraints.ucf*: Timing and placement constraints for the SP605.
  - *constraints_sim.ucf*: Timing constraints for simulation.
  - *pcie_wrapper.vhd*: A wrapper for the Spartan-6 PCI Express endpoint core to remove all unneeded signals from cluttering other parts of the design.
- hardware

  - *carp2D.xise*: Project file for the 2D design.
  - *carp2Dsim.xise*: Project file for simulation of the 2D design.
  - *carp3D.xise*: Project file for the 3D design.
  - *carp3Dsim.xise*: Project file for simulation of the 3D design.
- software/common
  - *pci.h*: Header file for pci.c
  - *pci.c*: A general interface to find and memory-map PCI device memory regions.
  - *sp605.h*: Header file for sp605.c
  - *sp605.c*: Provides an open/close/read/write interface towards the communication module implemented on the SP605.

Modified files:

- hardware/2D
  - *lss.vhd*: The LoadSendStore unit; a part of the control unit.
- hardware/3D
  - *dev.vhd*: The development unit.
  - *lss.vhd*: The LoadSendStore unit; a part of the control unit.
- software/2D
  - *read_print.c*: Convenience functions for printing states and types.
  - *sblocktest.c*: Tests.
  - *sblocklib.h*: Header file for sblocklib.h
  - *sblocklib.c*: The main API.
- software/3D
  - *read_print.c*: Convenience functions for printing states and types.
  - *sblocktest.c*: Tests.
  - *sblocklib.h*: Header file for sblocklib.h
  - *sblocklib.c*: The main API.

References

[1] A. Thompson, "An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics," in *Evolvable Systems: From Biology to Hardware*. Springer, 1997, pp. 390–405.

[2] A. Thompson and P. Layzell, "Analysis of Unconventional Evolved Electronics," *Communications of the ACM*, vol. 42, no. 4, pp. 71–79, 1999.

[3] S. Harding and W. Banzhaf, "Artificial Development," in *Organic Computing*. Springer, 2008, pp. 201–219.

[4] G. Tufte, "From Evo to EvoDevo: Mapping and Adaptation in Artificial Development," *Development*, 2008.

[5] S. Wolfram, "Universality and complexity in cellular automata," *Physica D: Nonlinear Phenomena*, vol. 10, no. 1, pp. 1–35, 1984.

[6] C. G. Langton, "Computation at the Edge of Chaos: Phase Transitions and Emergent Computation," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1, pp. 12–37, 1990.

[7] J. von Neumann and A. W. Burks, "Theory of self-reproducing automata," 1966.

[8] E. F. Codd, *Cellular automata*. Academic Press, 1968.

[9] *Spartan-6 Family Overview*, Xilinx, 2011, DS160 (v2.0).

[10] L. Huelsbergen, E. Rietman, and R. Slous, "Evolution of Astable Multivibrators in Silico," in *Evolvable Systems: From Biology to Hardware*. Springer, 1998, pp. 66–77.

[11] *Spartan-6 FPGA Configuration User Guide*, Xilinx, 2014, UG380 (v2.7).

[12] *Virtex Series Configuration Architecture User Guide*, Xilinx, 2004, XAPP151 (v1.7).

[13] P. C. Haddow and G. Tufte, "An evolvable hardware FPGA for adaptive hardware," in *Evolutionary Computation, 2000. Proceedings of the 2000 Congress*, vol. 1. IEEE, 2000, pp. 553–560.

[14] PCI-SIG, "PCI Express Base Specifications Revision 3.0," *PCI-SIG*, 2010.

[15] A. Djupdal, "Konstruksjon av maskinvare for kjøring av sblokkbaserte eksperimenter," 2003.

[16] K. Aamodt, "Kunstig utvikling: Utvidelse av FPGA-basert SBlock-plattform," 2005.

[17] S. Berg, "Evolution of Cellular Automata using Lindenmayer Systems and Fourier Transforms," 2013.

[18] O. M. T. Støvneng, "Extending an sblock platform for a new target hardware," 2014.

[19] *SP605 Hardware User Guide*, Xilinx, 2011, UG526 (v1.6).

[20] DistroWatch, "DistroWatch Page Hit Ranking," http://distrowatch.com/dwres.php?resource=popularity, [Accessed: 2014-12-15].

[21] Michael Gernoth, "XILINX JTAG tools on Linux without proprietary kernel modules," http://rmdir.de/~michael/xilinx/, [Accessed: 2014-12-16].

[22] *Spartan-6 FPGA Integrated Endpoint Block for PCI Express*, Xilinx, 2011, UG672 (v1.1).

[23] D. Koch, C. Haubelt, and J. Teich, "Efficient Reconfigurable On-Chip Buses for FPGAs," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*. IEEE, 2008, pp. 287–290.

[24] *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*, Xilinx, 2012, UG687 (v14.3).

[25] K. Chapman, "Get Smart About Reset: Think Local, Not Global," Xilinx, Tech. Rep., 2008, WP272.