A Cellular Automata Accellerator for SHMAC

Einar Johan Trøan Sømåen

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

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

For grandpa Paul Konrad Sømåen

Assignment text

The Single-ISA Heterogeneous MAny-core Computer (SHMAC) prototype is an ongoing research project within the Energy Efficient Computing Systems (EECS) strategic research area. SHMAC is planned to run in a Field-Programmable Gate Array (FPGA) and be an evaluation platform for research on heterogeneous multi-core systems. Due to the Dark silicon effect, future computing systems are expected to be power limited. The goal of the SHMAC project is to propose software and hardware solutions for future power-limited heterogeneous systems.

In this project the goal is to include a cellular computing inspired accelerator core to the SHMAC processor. The work aims at producing a Hardware Description Language (HDL)-implementation that includes a cellular array that can be integrated in the SHMAC architecture. Further, a suitable application should be proposed and demonstrated.

Supervisor: Gunnar Tufte

Abstract

Energy usage is an increasingly limiting factor for modern micro processors. One of these limitations is that it is no longer possible to power up all the transistors of the processor at the same time. The Single-ISA Heterogeneous MAny-core Computer (SHMAC) project considers how a heterogeneous multi core architecture can be used to reduce these limitations. A heterogeneous multi core architecture can consist of a selection of different cores, where each of these cores can be better suited for a specific task. Some of these cores can be, or contain, accelerators; hardware designed to perform very specific tasks efficiently.

This thesis covers the implementation of a Cellular Automata (CA) accelerator, which can be configured to evolve time steps for 1D or 2D CA. A functional implementation integrated into SHMAC is presented and demonstrated by a few example applications, including pseudorandom number generation in a 1D CA and Conway's Game of Life in a 2D CA. The resulting system merges an unconventional accelerator with a conventional processor.

Sammendrag

Energiforbruk er i økende grad en begrensende faktor for moderne prosessorer. En av disse begrensningene er at man ikke lenger kan få strøm nok til å slå på alle transistorene i prosessoren samtidig. Single-ISA Heterogeneous MAny-core Computer (SHMAC)-prosjektet undersøker hvordan en hetrogen multikjernearkitektur kan brukes for å redusere effekten av disse begrensningene. En hetrogen multikjernearkitektur kan bestå av et utvalg av forskjellige kjerner, der hver av disse kjernene kan være bedre egnet for en spesifikk oppgave. Noen av disse kjernene kan være, eller inneholde, akselleratorer; maskinvare utviklet for å utføre veldig spesifikke oppgaver effektivt.

Denne oppgaven omhandler implementasjonen av en aksellerator for Cellulære Automata (CA). Akselleratoren kan konfigureres slik at den kan utvikle tidssteg for både 1D og 2D CA. En funksjonell implementasjon integrert i SHMAC blir presentert og demonstrert ved hjelp av noen eksempler på bruksområder som f.eks. generering av pseudotilfeldige tall i en 1D CA og Conway's Game of Life i en 2D CA. Det resulterende systemet kombinerer en ukonvensjonell aksellerator med en konvensjonell prosessorkjerne.

Preface

This thesis is submitted to the Norwegian University of Science and Technology in partial fulfilment of the requirements for a Master's degree.

Acknowledgements

I'd like to thank my supervisor Gunnar Tufte for his insights and guidance with this project. Additionally I'd also like to thank the following people: Stefano Nichele for proof reading, and assistance with finding relevant papers for the use cases of this accelerator, Jakob Dagsland Knutsen for insights about, and assistance with getting the RealView machines running in the lab, Asbjørn Djupdal for lending out a lab computer, and assisting with getting the JTAG interface running, so that work could be done outside the lab at IDI, Caroline Sæhle for proof reading, and finally thanks to my girlfriend Kjersti Rise for her moral support, as well as for proof reading this thesis.

Table of Contents

Assignment text i

Abstract iii

Sammendrag v

Preface vii

Table of Contents xi

List of Tables xiii

List of Figures xvi

List of Listings xvii

Abbreviations xix

1 Introduction 1
1.1 Conventional Computer architecture 1
1.2 Alternatives to the von Neumann Architecture 2
1.2.1 Cellular Computing 2
1.2.2 CAM Brain 2
1.2.3 Developing cellular computing machines 2
1.3 Bridging the gap 3
1.4 Organization of this thesis 3

2 Background 5
2.1 Complexity and Emergence 5
2.1.1 Cellular Automata (CA) 6
2.1.2 The edge of chaos 7
2.2 Genetic algorithms (GA) 8

2.3 Turing-machines, von Neumann architectures and beyond . . . 9
2.4 The Single-ISA Heterogeneous MAny-core Computer (SHMAC) . . . 10
2.4.1 Single-ISA Heterogeneous Multicore Architectures . . . 10
2.4.2 SHMAC in detail . . . 11
2.5 Hardware acceleration . . . 12

3 Cellular Automata (CA) 15
3.1 Formal definition . . . 15
3.2 Introduction to Cellular Automata . . . 16
3.2.1 Neighbourhoods . . . 16
3.2.2 Number of states . . . 17
3.2.3 Uniformity of rules . . . 17
3.2.4 The layout of a rule . . . 18
3.3 Initial state . . . 18
3.3.1 Boundary conditions . . . 19
3.4 Von Neumann's Universal Constructor . . . 22
3.5 Hardware implementations of Cellular Automata . . . 23
3.5.1 CAM6 . . . 23
3.5.2 Pipelined FPGA-implementation . . . 23

4 Accelerator Design 25
4.1 Notation . . . 25
4.2 Design of the CA-accelerator . . . 25
4.2.1 Comparison to other implementations . . . 25
4.2.2 The cell . . . 27
4.2.3 The rule table . . . 28
4.2.4 The cell grid . . . 28
4.2.5 The grid controller . . . 31
4.2.6 The CA-toplevel . . . 32
4.2.7 The coprocessor interface . . . 33
4.2.8 Configurability . . . 33

5 Using the accelerator 35
5.1 Coprocessor interface . . . 35
5.2 Description of operations . . . 37
5.2.1 OP_HALT . . . 37
5.2.2 OP_RUN . . . 37
5.2.3 OP_SETMASK . . . 37
5.2.4 OP_GETINFO . . . 38
5.2.5 OP_SETRULE . . . 38
5.2.6 OP_SETEDGE . . . 38
5.2.7 OP_LOADCOL . . . 38
5.2.8 OP_RST . . . 39
5.3 Example of assembly usage . . . 39
5.4 C interface . . . 40
5.5 The layout of a rule . . . 44

6 Testing and Demonstration **47**
6.1 Software CA implementation . . . 47
6.2 Simulation vs FPGA-tests . . . 47
6.3 Realview setup . . . 49
6.4 Wolfram rules 0-255 . . . 49
6.5 Pseudorandom number generation . . . 50
6.6 Density Classification . . . 52
6.7 Conway's Game of Life . . . 53
6.8 Genetic Algorithms . . . 57
6.9 Continued usage test . . . 60

7 Conclusion **61**
7.1 Speed of the accelerator . . . 62
7.2 Discussion of limitations and trade offs . . . 62
7.2.1 Overhead in communication . . . 62
7.2.2 Fixed direction of input/output . . . 63
7.2.3 Complex system for rule description . . . 63
7.2.4 Size limitations . . . 63

8 Future Work **65**
8.1 Other configurations . . . 65
8.2 Variations on the wrapping configuration . . . 66
8.3 Scheduling issues . . . 66
8.4 Size optimizations . . . 67
8.5 Different I/O approaches . . . 67
8.6 Rule table optimizations . . . 68
8.7 Power optimizations . . . 68
8.8 Power usage considerations . . . 68

Bibliography **69**

List of Tables

3.1 Rule 30 in Wolfram notation 18

4.1 The VHDL generics available for configuring the size, neighbourhood and amount of serial updates at synthesis time. 34

5.1 List of available opcodes in the accelerator, along with the sizes of the arguments. 36

5.2 The relative order of the neighbours in the bitstring that define a neighbourhood state. The shorter bit strings are centred to demonstrate how the larger neighbourhood sizes expand upon the case of a neighbourhood size of 3 in both ends of the bitstring. 45

List of Figures

1.1 The components of the von Neumann Architecture. 1

2.1 Evolution of an elementary CA where the rule is such that cells change their state to 1 in the next time step if they have exactly one neighbour with state 1 and they themselves are state 0, in all other cases they change their state to 0. 6
2.2 The Wolfram classes as placed in the λ-scheme by Langton[1], demonstrating how the complexity of the behaviour (Class 4) increases when approaching the ”edge of chaos”. 7
2.3 The high-level design of the SHMAC processor[2], showing how the processor is subdivided into a number of *tiles*, which are cores that may differ from each other in various ways. 11

3.1 Some examples of neighbourhoods of a cell. 16
3.2 The von Neumann-neighbourhood with a size of 5(left) and the Moore-neighbourhood(right) with a size of 9, both with $r = 1$. 17
3.3 Rule 60 evolved with fixed and wrapping boundary conditions 19
3.4 Rule 90 evolved with fixed and wrapping boundary conditions from a single off-centre cell with state 1 20
3.5 Rule 90 evolved from a centred single cell with state 1, with fixed and wrapping boundary conditions 21
3.6 Rule 110 evolved with, and without wrapping boundary conditions 22

4.1 The communication possibilities between the CAS and the accelerator interface. Cells can only communicate with their direct neighbours, the input can be used as the western neighbour on the western boundary, the output can be used as the eastern neighbour on the eastern boundary. 26
4.2 The communication responsibilities of a single cell. 27
4.3 The update flow in a *cell grid* with 3 cells. 29
4.4 The connections exposed by a grid for connecting grids together. 30
4.5 Six *cell grids* connected together, updating in parallel with each other. . . 30

4.6 A demonstration of how the output information from a southern grid is used by multiple northern nodes when the neighbourhood size is 9 31
4.7 A conceptual view of the CA-toplevel, demonstrating the components it contains. 32
4.8 The three different neighbourhood size configurations available in the accelerator at synthesis time. 34

6.1 The initial configuration for the rule 30 based pseudorandom generation . 51
6.2 The distribution of the first 65536 pseudorandom bytes from rule 30. . . . 52
6.3 15 time steps of short period Game of Life patterns, evolved on the CA-accelerator 54
6.4 The Gosper Glider Gun. 54
6.5 55 time steps of the Gosper Glider Gun, time steps are listed top to bottom, left to right. 55
6.6 Additional 30 time steps of the Gosper Glider Gun, demonstrating the glider stopping at the boundary. 56
6.7 The same time steps as in the preceding figure, with wrapping boundary conditions. 56
6.8 Some additional time steps of the Gosper Glider Gun with wrapping boundary conditions, demonstrating the destructive glider. 57
6.9 Fitness values across a run where the fitness function was defined as the fraction of cells with state 1. Each genome was run 5 times, thus 5.0 is the highest possible fitness value. 59
6.10 Fitness values for the GA-generations where the steep increase in best and average fitness were visible in the previous figure. 59

List of Listings

5.1 An example of assembly code that makes the accelerator evolve rule 30 for 512 time steps. 39
5.2 An example of using the C-interface to evolve 50 time steps of rule 30 . . 43
6.1 An example of pseudo-assembly for evolving 10 time steps of rule 30, comparing the results to those of the software implementation afterwards. 48

Abbreviations

CA	=	Cellular Automata
CAS	=	Cellular Automata System
DNA	=	Deoxyribonucleic Acid
FPGA	=	Field-Programmable Gate Array
GA	=	Genetic Algorithm
IFM	=	Interface Module
ISA	=	Instruction Set Architecture
SHMAC	=	Single-ISA Heterogeneous MAny-Core Computer
VHDL	=	VHSIC Hardware Description Language

Chapter 1

Introduction

1.1 Conventional Computer architecture

Most, if not all, of the commercially available computer systems today are instances of the von Neumann architecture[3]. On a general level, the von Neumann architecture consists of five components, shown in **Fig. 1.1**. This subdivision implies a certain division of labour between the components that the system consists of. There is a *memory* that is used for both instructions and data, a *central arithmetic* unit that is responsible for performing various arithmetic operations, a *central control* unit which is responsible for managing the current instruction to be performed, particularly the ordering of instructions. Finally there are 2 conceptual components that represent communication with the ”outside world”, namely *input* and *output*. In short, this architecture has an instruction flow where the instructions that are to be performed, along with any relevant data, are read from the *memory*-unit, and then these instructions are performed by the *central arithmetic* unit, writing back any results to the *memory*-unit. This flow is handled by the *central control*-unit. Overall this design consists of components that have very clear areas of responsibility, and very specific parts of the system perform the actual computations involved. One thing to note about this architecture, is that it involves the concept of a stored program (i.e. storing a sequence of instructions in memory).

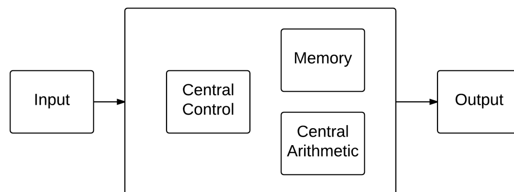


Figure 1.1: The components of the von Neumann Architecture.

However, this is not the only way to design a computer, a slight variation can be found in the Harvard architecture[4, p. L4], where the memory used for instructions is separate from that used for data. This slight change does not however deviate very much in the way the flow of execution is performed, retaining the stored program concept.

1.2 Alternatives to the von Neumann Architecture

A more radical departure from the von Neumann architecture can be found in systems that move away from the very concept of a central component responsible for performing the actual computation, opting instead for quite different computation methods.

1.2.1 Cellular Computing

One possibility for computing without the same kind of global communication that is used in the von Neumann architecture, is the *cellular computing* approach[5], where the central control, and complex arithmetic units of the von Neumann architecture are replaced by a (potentially) large number of simple, vastly parallel and locally interacting *cells*. A specific example of a cellular computing system is Cellular Automata (CA), where all communication is local (i.e. between neighbouring cells). While each composing cell of such a system only reacts to local phenomena, the results that can be observed across the system of cellular automata can demonstrate emergent complexity at a global level. This system can then be used to perform various tasks in ways that are very different from those that would be used in conventional systems. A cursory introduction to CA will be given in Chapter 2.1.1, with a more involved treatment being given in Chapter 3.

1.2.2 CAM Brain

Another way to approach the problem of computation, is to try to create an artificial brain. De Garis attempted just that in his work on the CAM Brain[6], an artificial brain intended to control a robot cat. In the CAM Brain project, a CA was used as the growing medium for a neural network. The CA implementation involved a large set of hand written CA rules that described the movement and reaction of a set of states, making up the building elements and signals of the neural network. A sequence of these building blocks were evolved using a Genetic Algorithm (GA), and applied to the CA as input, so that the paths of the neural network could be built as extensions on a small initial structure.

1.2.3 Developing cellular computing machines

Tufte and Haddow[7] considered a different approach to creating a cellular computing machine, in their work on artificially developing such a machine. Their approach uses ideas inspired by how biological cells grow, in that their non-uniform CA had cells of different types, that would grow (i.e. expand into neighbouring cells, so that some neighbouring cell becomes the same type) or be created at various positions in the CA, depending on the types of cells they were neighbours of. In this way, the development is a reaction to purely local information, similar to how a CA cell reacts, with the minor difference that

development can lead to a cell changing one of its neighbours upon growth. The point of a developmental approach, is that the resulting CA will be a result of a set of rules for how development should proceed, instead of having to be described in terms of exactly what ends up where as would be the case if the CA behaviour was specified by i.e. applying a configuration directly to an FPGA.

1.3 Bridging the gap

The Single-ISA Heterogeneous Multi-Core Architecture (SHMAC), which will be introduced in further depth in Chapter 2.4, is at its core a von Neumann architecture machine. However, as will be demonstrated, the particular design of the SHMAC, allows for introducing very different or specialized processor cores onto the same chip as quite conventional cores. It is thus possible to combine a conventional a with an unconventional design, such as that of cellular automata. This thesis covers the design and demonstration of a cellular automata based accelerator for use in SHMAC, which results in exactly that, a way to combine a von Neumann-architecture core with an unconventional approach to computing.

1.4 Organization of this thesis

The structure of the remainder of this thesis is as follows:

- Chapter 2 discusses the theory and history behind the cellular computing approach, as well giving some background on the SHMAC-project and accelerators in general.
- Chapter 3 gives an in-depth introduction to how Cellular Automata behave, and provides a few definitions that will be recurring in the following chapters.
- Chapter 4 describes the design of the CA-accelerator in a bottom-up fashion, going into detail on how the various parts of the accelerator cooperate in allowing for the evolution of successive CA time steps.
- Chapter 5 gives a description of how the accelerator can be used, both from ARMv3 assembly and through a C interface.
- Chapter 6 presents some sample applications of the accelerator, including pseudo-random number generation, density classification, Conway's Game of Life and a genetic algorithm implementation.
- Chapter 7 summarizes the work that has been done, and discusses some of the design choices that were made, in particular the limitations of the design as currently implemented.
- Chapter 8 presents ideas for how the accelerator can be further improved upon, or applied for new research opportunities in the SHMAC-project.

Chapter 2

Background

This chapter will give a some background on cellular computing, comparing the approach to the conventional approach to computing. Additionally, some background on the SHMAC project and accelerators in general will be covered.

2.1 Complexity and Emergence

The classical way of trying to understand phenomena in the universe is reductionism, the act of taking things apart to see how their parts work. If this is repeated to deeper and deeper levels, the expected outcome would be to understand how the thing at hand works, as a consequence of how the various parts of it work. This approach will however not consider any effects that only happen when the pieces are actually working together, or as Douglas Adams wrote ”If you try and take a cat apart to see how it works, the first thing you have on your hands is a nonworking cat”[8, p. 135]. There might be behaviour in a system that only happens when the components of said system work together, but which can not be observed readily on the individual component level, effectively making ”the sum greater than the sum of its parts”. Such behaviour is referred to as *emergent behaviour*[9, p. 13].

Emergent complex behaviour can be observed across a variety of fields, a quite varied selection of which is demonstrated in Mitchell’s ”Complexity: A guided tour”[9, p. 3-12], with examples ranging from the behaviour of a colony of ants (where collective behaviour that adapts to the colony’s needs can be seen, without any single controlling entity), to global economies and the way the human brain works. Such emergent behaviour can be seen as complexity being built on top of less complex parts. A variety of approaches have been made to try to understand such phenomena, the mechanisms behind them, as well as the effects thereof. One way of examining such effects, is to try to build a controlled system consisting of simple parts, that still exhibits emergent behaviour, and then look at the behaviour of such a system under a variety of different conditions.

2.1.1 Cellular Automata (CA)

One example of a system consisting of simple parts that is capable of exhibiting emergent complexity can be found in Cellular Automata (CA). CA are systems of simple *cells*. Each cell in such a system can be in one of a variety of discrete states, and will change this state at discrete points in time depending on the states of the cells that it is connected to, deciding its reaction according to some *rule*. The process of creating a new set of states in a CA is referred to as *evolving* a *time step* of the CA. Further details on CA will be given in Chapter 3). In the simplest form of CA, an infinity of equivalently behaving cells are connected in a line, such that all the cells have exactly 2 cells that they are connected to as neighbours, and only 2 different states that each of the cells can be in, this particular type of CA is called an *elementary CA*[10]. **Fig. 2.1** demonstrates three time steps of an elementary CA. The circles correspond to cells, whose state is shown as a number inside the circle, the arrows denote a connection between two cells. The rule that each cell follows is as follows: If the cell's own state is 0, and exactly one of its neighbours (the cells that it has connections to) is 1, then the cell changes its state to 1 in the next time step. In all other cases its state becomes 0 in the next time step. Note that this example uses a circular finite configuration of the cells, instead of an infinite configuration to simplify the presentation.

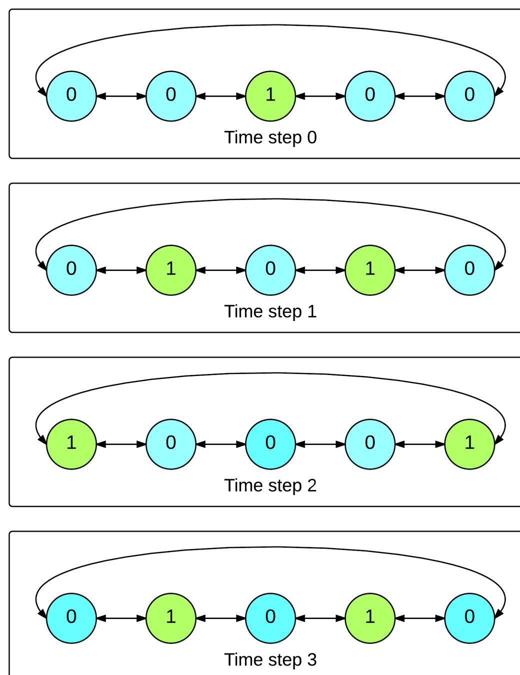


Figure 2.1: Evolution of an elementary CA where the rule is such that cells change their state to 1 in the next time step if they have exactly one neighbour with state 1 and they themselves are state 0, in all other cases they change their state to 0.

The possible behavioural rules of each cell in elementary CA have been studied quite thorougly by Stephen Wolfram[11, Chap. 3][10], who also classified the resulting global behaviour of the system of cells into 4 different categories[12]:

- Class 1: the cases where the system eventually enters a specific fixed state.
- Class 2: the cases where the system eventually enters a periodic sequences of states (possibly static).
- Class 3: the cases where the system behaves in a chaotic fashion.
- Class 4: the cases that exhibit *complex* behaviour. That is, the behaviour has some emergent structure.

While the original paper by Wolfram divided all the elementary CA neatly into Class 1-3, with none of them falling into Class 4, it has later been shown that at least one of them can be used as a Turing machine[13], which would imply complex behaviour (i.e. Class 4-behaviour). This then means that even *elementary* CA can serve as an example of a system of very simple components that still can exhibit emergent complexity.

2.1.2 The edge of chaos

The question of which, if any, kinds of CA, are suitable for computation was considered by Langton[1]. As Class 1 and Class 2 automata exhibit ordered behaviour, and Class 3 automata exhibit chaotic behaviour, Langton found that the the complexity of behaviour increases as rules become less ordered (or equivalently more chaotic), until a point, where chaos starts to take over. The potential for complex behaviour was thus found to be higher when a good balance between order and chaos is in place, on what was called the ”edge of chaos”. Langton used λ as a way to describe a CA-rule’s degree of chaotic behaviour. λ is defined such that for some arbitrary, specified state, λ specifies the fraction of neighbourhood states that transition into random states *different* from this state. Thus, at $\lambda = 0$ all neighbourhood states will transition to this state, and at $\lambda = 1$ every neighbourhood states transition to a state different from this state. He then considered the behaviour of rules for different values of λ, finding the results seen in **Fig. 2.2**.

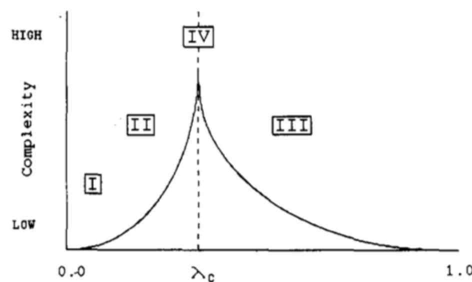


Figure 2.2: The Wolfram classes as placed in the λ-scheme by Langton[1], demonstrating how the complexity of the behaviour (Class 4) increases when approaching the ”edge of chaos”.

Ordered behaviour (Class 1/2) decreases with increasing λ, while chaotic behaviour decreases with decreasing λ, as order approaches chaos, and chaos approaches order, the complexity of the behaviour increase, demonstrating that Class 4-behaviour is found close to the transition between mostly ordered and mostly chaotic, the ”edge of chaos”.

2.2 Genetic algorithms (GA)

Finding a solution to a problem from a set of potential solutions can take a prohibitively long time unless a systematic approach is carefully selected[14, p.17]. One such approach is found in the field of *genetic algorithms*. Genetic Algorithms (GA) approach the problem of finding a solution, by borrowing some concepts from sexual reproduction and evolution in nature.

Simplifying somewhat for the sake of analogy, the Deoxyribonucleic Acid (DNA) in living beings acts as the ”blueprint” that is used to construct the individual and leads to its various traits. Whenever such individuals reproduce sexually, the DNA of the parents is combined (not necessarily flawlessly) to create the DNA for a new individual[15, p.5-6]. At this point both parents have proven themselves capable of surviving long enough in their environment to produce offspring. Any of the traits that allowed them to survive this far that were a result of their DNA might (depending on the recombination) now be passed on to their offspring, which might in turn pass this DNA on to its offspring if it survives long enough. The parents that are better suited for their environment will have more offspring, and thus, their DNA will be more widespread and combined in more ways than those of the less suitable individuals.

Genetic algorithms as invented by Holland[14] take these concepts and applies them to general problem solving. A genetic algorithm starts with some population of potential solutions to a given problem, each of which is described by a set of ”*chromosomes*” in a fashion inspired by DNA, that is, in a way that allows for recombination and mutation to take place. Such a complete description is referred to as a *genome*, and its components are *chromosomes*. The population of genomes is used to attempt to solve the given problem, and their results are evaluated, giving the solutions *fitness*-values proportionate to their ability to solve the problem. Then a new population is created from the existing population. This is done by selecting individuals for reproduction from the current population in a random way that allows for solutions with higher fitness values to have a higher probability of reproduction than the lower scoring individuals. The selected individuals are then potentially subjected to recombination and/or mutation before the process is repeated. The process stops after some number of *generations*[15, p.10-12].

Genetic algorithms allow for searching quite large search spaces, in a directed fashion, by exploring various promising solutions[14, p.140]. They test variations of these in such a way that the chromosomes of the solutions that correlate with higher fitness will be more widespread over time through recombination, while avoiding losing chromosomes that might still be useful (for instance to avoid being stuck at a locally optimal fitness ”peak”) in some yet untested combination by allowing for mutation to reintroduce lost chromosomes[14, p.110]. One application of GA is to find rules for Cellular Automata, as the amount of possible rules of such CA grows very rapidly with the neighbourhood size. (See Chapter 3.2.4).

2.3 Turing-machines, von Neumann architectures and beyond

The classical Turing-machine[16] is the concept of a machine that can move back and forth along a tape of infinite length, reading and writing from that tape, performing actions according its own state and the contents of said tape. The concept of such a machine being used to run another Turing-machine was used by Alan Turing to describe the *halting problem*, and thereby to state that not all problems can be decided. The process that this machine performs, in transforming the content of its tape, is *computation*[9, p.146].

Where the Turing machine by design separates data and instructions, a Turing machine that simulates another Turing machine (i.e. by having the Turing-machine behaviour as a result of its ”reactions” to the tape contents, thereby having the specific Turing-machine's behaviour encoded on the tape) is referred to as a *Universal Turing Machine*[9, p. 64]. Since a Universal Turing Machine is capable of simulating any other Turing Machine, any problem that is solvable by some Turing Machine (i.e. a problem that can be solved by a definite sequence of operations), can be solved by a Universal Turing Machine[16]. In terms of terminology, if something is equivalent to a Universal Turing Machine, it is capable of *universal computation*[9, p.149].

Demonstrating that some specific instances of the von Neumann Architecture can be capable of universal computation can be done by demonstrating that the actions of a Universal Turing Machine can be performed by said machine. The parallels in the design of the von Neumann architecture to the concept of a Turing machine is at times quite direct; the *tape* finds a corresponding component in the *memory*-component, while the movement of the position relative to that tape finds a parallel in the *central control*-component, and the set of potential actions is covered by the *central arithmetic*-component. There are however some deviations; the *memory*-component is not defined to be of infinite size, and the operations available from the *central arithmetic*-component in specific implementations might not be enough to construct a *universal* Turing Machine in a von Neumann architecture. If the memory component is idealized to have infinite size, then the question becomes whether some (finite) set of available operations in the *central central arithmetic* component can be shown to be able to perform the actions of a universal Turing Machine. One example of work that has been done to demonstrate a small set of operations that are sufficient for this purpose is shown in Rojas' work on demonstrating that unconditional branching is not a requirement for universal computation[17].

The construction of patterns that have behaviour equivalent to Universal Turing Machines has also been demonstrated in some configurations of Cellular Automata, such as the elementary CA rule 110[13]. This implies that at least some CA are capable of universal computation, just like some von Neumann-architecture machiness are. However, they diverge from the von Neumann-architecture in a number of ways, their state is separate from the rules that decide how they act, they lack a specific component corresponding to *Central Arithmetic*, and (perhaps apart from a common timer to synchronize them) they also lack the *Central Control*-component. In practical implementations they are however very likely to have some sort of *Input* and/or *Output*-component, so that at least their initial state can be written, and their final state can be read. These details boil down to the realization that CA are not in fact von Neumann-architectures. They are actually referred to

as an example of non-von Neumann architecture machines. As pointed out by Mitchell[9, p.149], there is some irony in this naming, as von Neumann was one of the early pioneers of the field that led to the modern Cellular Automata-concept.

It is also worth mentioning, as Sipper[5] does, that while demonstrating equivalence between certain configurations of CA and a Universal Turing Machine does demonstrate the potential for universal computation in CA, it also somewhat misses the point, as it demonstrates the potential for universal computation in a cellular computation environment by introducing the serial concepts of the Turing Machine, which avoids making use of the inherent parallelity of a cellular computation approach. Thus, approaching computation problems on the premises of the cellular computation-philosophy, instead of trying to bend cellular computation into mimicing solutions used in traditional architectures is an interesting field. Or to put it slightly differently; although computation implies a task achievable by a Turing Machine, it does not require the task to actually be performed by a (Universal) Turing Machine.

This then leads to the interesting idea of implementing a non von Neumann-architecture (Cellular Automata) accelerator in a system that is otherwise a von Neumann-architecture (SHMAC).

2.4 The Single-ISA Heterogeneous MAny-core Computer (SHMAC)

2.4.1 Single-ISA Heterogeneous Multicore Architectures

For the past ten or so years, the development of new generations of processors has had to change the design goals away from the traditional goal of mainly increasing the clock rate to increase the speed compared to the previous generations, instead having to find other ways to increase performance. One of the reasons behind this change are the increasing power requirements that followed the steady increase in clock rate, and the amount of heat generated by using ever increasing amounts of power.[18, p.39-42][19, p.21-24]

This heat/power problem is even more relevant when considering battery powered devices that contain processors. Unlike a desktop computer, battery powered devices such as smart phones and tablets not only have to worry about heat, but also about draining the battery too quickly[20]

When power usage and heat became more central as design goals, new design trends were approached, with the aim now changing from strictly increasing performance, to increasing performance while taking power requirements into account. One potential way to increase performance in a new, *potentially* less power demanding fashion, is increasing the amount of cores. Each of the cores in a multi core architecture can run at somewhat slower clock speeds than the single core that could have been created using the same die area. However, as long as there is enough potential for exploiting paralellism[18, 41], or the cores themselves are more efficient than the single core they replace[19, p.55], the resultant multi-core CPU could still get ahead performance wise.

Most mass market multi-core processors are designs with homogeneous cores, that is, each of the cores are exactly the same as any other core. This might not necessarily be the most optimal configuration for all use cases. Consider for instance a use case where high

performance is in reality only necessary for some threads of execution, while other threads perform work that is not necessarily needed for a while. If these threads could have been performed in a slower fashion while keeping the end result the same with respect to overall runtime, a lower amount of power could potentially have been used to solve the same problem. An approach to allowing for such a scenario, is to diverge from the homogeneous multi core design, and instead consider a heterogeneous multi core architecture. Such architectures are not new concepts, Kumar et al.[21] considered an architecture where multiple generations of the Alpha architecture were fitted together on a single CPU-die, and attempted various schemes for deciding when to transition between the various cores, and demonstrated that for many of their benchmarks, power consumption could be reduced by a very noticeable amount without reducing performance by anything close to the same amount.

2.4.2 SHMAC in detail

A power related design issue that must be taken into account when designing future processor architectures, is that of dark silicon. ”Dark silicon” refers to the problem that after Dennard scaling of voltage failed, there is a limit to the percentage of transistors that can be powered up at the same time, leaving the remaining transistors ”dark” [22]. Thus, efficient use of the area that at any time is powered up is essential if performance is to be increased.

SHMAC is a project at the Computer Architecture and Design group at IDI, that builds upon the original SHMAC implementation as presented in Rusten et al’s masters thesis[23]. The SHMAC project[2] aims to research how a heterogeneous multi-core architecture can be advantageous in reducing the limitations introduced by the dark silicon effect. The SHMAC processor consists of a number of *tiles*, which can be either a processor tile, an accelerator tile, or one of a variety of support tiles, the layout of such tiles is demonstrated in the high-level design of SHMAC in **Fig. 2.3**,

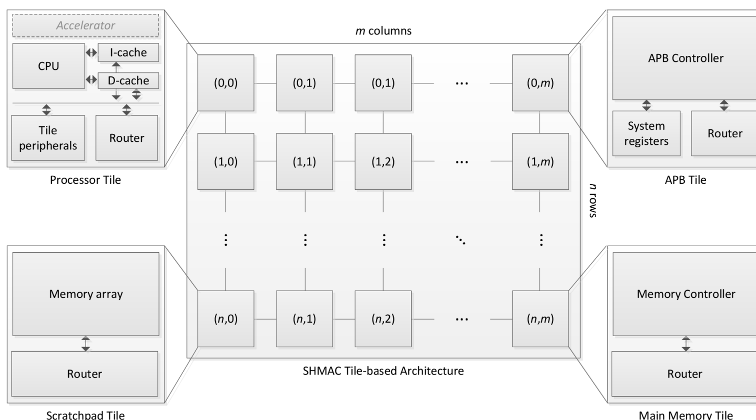


Figure 2.3: The high-level design of the SHMAC processor[2], showing how the processor is subdivided into a number of *tiles*, which are cores that may differ from each other in various ways.

While all the processor tiles follow the same Instruction Set Architecture (ISA), some processor tiles may be better suited for running a certain task in a more speed- or energy-efficient fashion, an example of which can be found in for instance the Turbo Amber processor-tile[24] which has, among other things improved multiplication performance when compared to the Amber processor-tile.

If for instance SHMAC is configured with one slow but energy efficient tile, and one fast, but more power hungry tile, then it might be more beneficial overall to allow some tasks to be run on the faster tile, both when considering execution time and power usage. If the faster tile is faster enough at the specific task compared to the slower tile, it can end up completing the task using less power than the slower tile, by performing the task in less time and then going into a low power state, this is called *race to halt*[19, p. 26]. Similarly, if we are performing a number of parallel tasks, where some of the tasks takes less time than other tasks, then such tasks can run on slower tiles while the lengthier tasks are run on faster tiles, any (short) serial portions can be run on a fast tile to reduce the serial bottleneck somewhat. This allows for a way to assign resources based on need, in such a fashion that the resources in the CPU is used in an effective way, allowing for balancing power usage with execution time. It also allows for reducing the limitations posed by the dark silicon effect somewhat by disabling the resources that are not currently needed, enabling the CPU to power up the necessary parts instead.

2.5 Hardware acceleration

There are various ways to allow for SHMAC-tiles that excel at specific tasks. The tile can for instance have; a faster ALU, a different pipeline, or be outfitted with specific functionality to allow for very specific operations to be performed in a faster fashion. One such example is a processor tile that has hardware support for performing floating point operations faster. Such a tile would (hopefully) be faster at performing tasks that depend on floating point numbers when compared to a tile that has to perform its floating point operations by way of a software implementation. The hardware component used for performing the floating point arithmetic in that example, is an *accelerator* for floating point arithmetic. This particular accelerator would be placed inside the processor tile, as shown in the upper left corner of **Fig. 2.3**, making the CPU component in such tiles capable of offloading some tasks that the accelerator is suited for, such as floating point operations in this case, to the accelerator. The CPU component can do this by communicating the necessary information and configuration to the accelerator, and then (potentially) continue doing other operations while the accelerator works.

Another tile variation is also possible, where the specialized *accelerator* functionality resides in a tile of its own. Such a tile would not contain the CPU-component that a normal processor tile contains, but could be used as an accelerator by other SHMAC tiles. The difference between these two configurations is that when the accelerator is internal to a processor tile, then that accelerator is usable by that tile alone, for instance if tile (0,0) in **Fig. 2.3** is a processor tile that has an accelerator, then only tile (0,0) can use that accelerator. If however tile (0,1) is an accelerator tile, then other tiles, such as tile (0,0) can use that accelerator by communicating with it through the internal SHMAC-communication[23, p.41], allowing for the accelerator to be shared among the processor

tiles.

There are a multitude of potential tasks that can be *accelerated*, that is, performed faster or more energy effectively. The market already has examples of hardware that accelerates operations often used in graphics, like matrix- and vector-operations which often go hand in hand with floating point operations. Some examples are Intel's x86 SSE-instruction-set[19, p.282-283], or the PowerPC's AltiVec[19, p.A-31], both of which expand the capabilities of the CPU in such a way that specific operations can be performed in a faster way, while still allowing for software written and compiled without knowledge of the extensions to continue to work.

Matrix/vector operations are suitable for acceleration by way of specialized hardware, as there is a lot of repeated logic involved in for instance performing a Matrix-Vector product which can be performed in parallel. Similarly, evolving successive time steps of cellular automata involve a lot of independent operations that can be performed in parallel. As will be demonstrated in Chapter 3, by definition, all the cells in a CA only communicate locally. Since the evolution of a CA time step only depends on the state from the preceding time step, almost all of the work necessary to evolve a new time step can be done in parallel. The only requirement being that the previous time step is not overwritten until the necessary work for all the cells are completed.

This leads to the accelerator presented in this thesis; an accelerator for the SHMAC architecture that exploits the inherent parallelism of CA to allow for evolving successive CA time steps of CA in a fashion that is faster than what a pure software implementation on the general processor tiles would allow.

Chapter 3

Cellular Automata (CA)

This chapter will give an introduction to Cellular Automata, terminology and definitions will be given, and some examples of evolution of time steps of CA will be demonstrated.

3.1 Formal definition

It is worthwhile to open a discussion of Cellular Automata with a formal definition of Cellular Automata by Langton[1]: ”Formally, a cellular automaton is a D-dimensional lattice with a finite automaton residing at each lattice site. Each automaton takes as input the states of the automata within some *finite* local region of the lattice, defined by a neighborhood template $\mathscr{N}$, where the dimension of $\mathscr{N} \leq$ D. By convention an automaton is considered to be a member of its own neighbourhood. [...]

Each finite automaton consists of a finite set of *cell states* Σ, a finite *input alphabet* α, and a transition function Δ, which is a mapping from the set of neighborhood states to the set of cell states.
Letting $N = |\mathscr{N}|$:
$\Delta : \Sigma^N \rightarrow \Sigma$.
The *state* of a neighborhood is the cross product of the states of the automata covered by the neighborhood template. Thus, the input alphabet α for each automaton consists of the set of possible neighborhood states: $\alpha = \Sigma^N$. Letting $K = |\Sigma|$ (the number of cell states) the size of α is equal to the number of possible neighborhood states
$|\alpha| = |\Delta| = |\Sigma^N| = K^N$.
To define a transition function Δ, one must associate a unique next state in Σ with each possible neighborhood state. Since there are $K = |\Sigma|$ choices of state to assign as the next state of each of the $|\sigma^N|$ possible neighborhood states, there are $K^{(K^N)}$ possible transition functions Δ that can be defined.”.

A similar formal definition can be found in Codd[25].

3.2 Introduction to Cellular Automata

As the formal definition above suggests, CA are collections of cells that have a state, a neighbourhood of connected cells and a transition function that defines how its state changes. The behaviour of a CA is discrete in time and space[10]; A cell has a fixed position, which may not change, and a state at time step t. It will change its state to a (potentially different) state at time step $t + 1$ according to its defined transition function. This transition function defines the state of a cell at time step $t + 1$ depending on the *neighbourhood state* at time step t. In the context of this thesis, such transition functions will be referred to as *rules*, following the convention used by i.e. Wolfram[10].

3.2.1 Neighbourhoods

One of the central definitions for cellular automata is that cells are limited to only communicating with their *neighbours*[1], no direct global communication is involved. Thus it is necessary for a given CA to unambiguously define exactly which cells are considered to be neighbours of a particular cell, that is, which cells are part of the *neighbourhood* of any given cell.

When defining the neighbourhood of a cell one of the first consideration to take into account is what directions any given cell should find its neighbours in, specifically how many dimensions should be considered. This is referred to as the *dimensionality* of the CA, **Fig. 3.1** demonstrates the cases of 1D and 2D neighbourhoods. For clarity, multiple rows are not shown in the 1D-case, but a 1D CA can exist in a 2D system[26, p. 95], although as long as the single dimension taken into account is as shown in the **Fig. 3.1**, each of the rows would be completely independent of every other row. The dimensionality

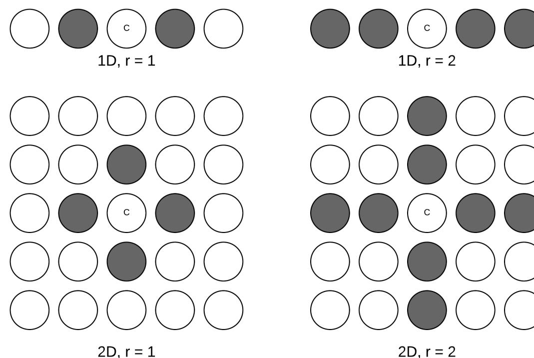


Figure 3.1: Some examples of neighbourhoods of a cell.

alone does not completely define which cells are neighbours, it is also necessary to define how many cells in any spatial direction that are to be treated as neighbours. Following the notation[27] commonly used to describe CA, the radius r is the number of cells that are considered part of the neighbourhood of a cell in any relevant direction, such that if $r = 1$,

only immediate neighbours are considered, while if $r = 2$, the neighbourhood will include 2 cells in any relevant direction. These two cases are also demonstrated in **Fig. 3.1**.

For the case of 1D CA, the dimensionality, and r is enough to decide which cells are neighbours, however for the higher dimensions there is still some ambiguity left, for instance, in the case of 2D CA with $r = 1$, it is not immediately clear whether some or all of the diagonally adjacent cells are considered part of the neighbourhood. This can be resolved by also defining the total number cells that are part of a neighbourhood, that is, the neighbourhood's *size*. As demonstrated in **Fig. 3.2**, the two different 2D $r = 1$ neighbourhoods, commonly referred to as the *von Neumann-neighbourhood* and the *Moore-neighbourhood* respectively[26, p.60], can be distinguished by their neighbourhood sizes of 5 and 9 (including the central cell itself). The intermediate sizes (such as 7) are not well defined in this matter, and will not be further elaborated upon, as their behaviour can be followed by a CA with size 9 that follows rules that ignore 2 out of the 9 cells in the neighbourhood.

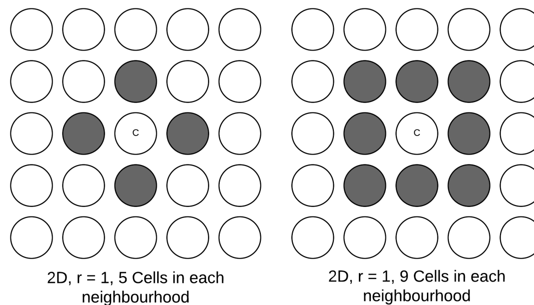


Figure 3.2: The von Neumann-neighbourhood with a size of 5(left) and the Moore-neighbourhood(right) with a size of 9, both with $r = 1$.

In the rest of this thesis, $r = 1$ will be used, thus the neighbourhood size alone suffices for defining the neighbourhood: a size of 3 implies a 1D CA (having neighbours in the same row), sizes of 5 or 9 implies a 2D CA.

3.2.2 Number of states

It is also necessary to define the state-space of each cell. The states in a CA-cell can be either continuous or discrete, where the discrete-state CA is the one that will be considered here. It is common to define the size of the state space for a discrete CA by the constant k. In a binary CA $k = 2$, however other choices are also possible, such as $k = 29$ as used in the automata proposed by von Neumann[28]. In the rest of this thesis only binary CA will be considered.

3.2.3 Uniformity of rules

All of the cells in a CA follow some *rule* to decide their state in the next discrete time step. An important property to consider for a CA is whether all the cells follow the same rule, or

whether some, or all of the cells follow potentially different rules. If all the cells follow the same rules, then the CA is said to be *uniform*, otherwise it is referred to as *non-uniform*. If the amount of different rules is quite low, compared to the amount of cells, then the CA can be referred to as as *quasi-uniform*[27, p.27].

3.2.4 The layout of a rule

A complete rule for a CA describes the next state given any possible preceding state. In an elementary CA, that is a simple 1D CA with $r = 1$ and $k = 2$, there are 3 cells to consider as part of the neighbourhood, and thus $2^3 = 8$ possible states for the neighbourhood, each of which needs to have a corresponding *next state* defined in the rule. Since each of the 8 possible states can map to one of the k different values, there are k^8 different complete rule sets possible for such a CA. In the specific case of 1D CA with $r = 1$ and $k = 2$ these $2^8 = 256$ values can be referred to by a number, as specified by Wolfram[11, p. 53].

An example of Wolfram's notation for rule 30 is shown in **Table 3.1**. The possible states of a neighbourhood in the CA are listed in descending order from left to right, and the corresponding next state for each of these possible states is listed in the second row. The binary representation that results from this full enumeration of all the next states, is then the number for this rule. For clarity, such rule defining numbers will, when discussed in general, be referred to as *Wolfram numbers* in the context of this thesis. For convenience of the example, the third row of the table lists the values that the corresponding positions in a binary number would represent. It is thus clear that this rule set can be represented by the binary number 00011110_2, which in decimal becomes 16+8+4+2 = 30, and thus the name of the rule; *rule 30*.

State	**111**	**110**	**101**	**100**	**011**	**010**	**001**	**000**
Next state	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**
Value	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**

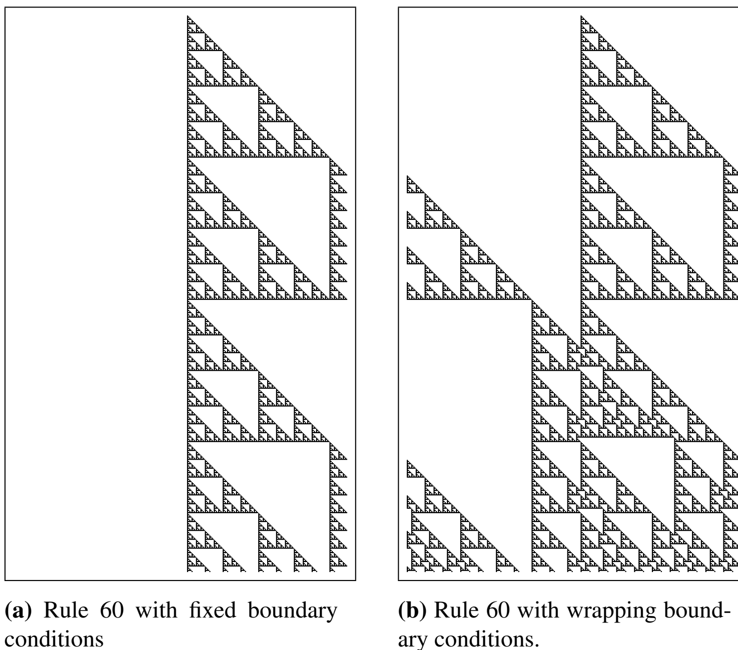
Table 3.1: Rule 30 in Wolfram notation

3.3 Initial state

A CA with well defined r, k, dimensionality, neighbourhood size and rules still lacks a final ingredient to begin evolving new time steps, namely the initial state. One choice for the initial state in CA with $k = 2$, is to have the initial state of all cells set to 0, except a single cell that is set to 1. This particular choice is the one used in Chapter 3 of A New Kind of Science[11]. Other options include selecting a completely random initial configuration, or, if the CA allows for changing the rules; the final state of some previous CA evolution.

3.3.1 Boundary conditions

Practical implementations of CA have to have some limit to the actual number of cells involved, the question then becomes, how should the boundary conditions (i.e. the cells that are situated at the ”end” of the CA) be handled? One potential solution to this, is to ”wrap” the boundary, i.e. let the cells situated at one end, be neighbours to the cells at the opposite end (i.e. the westernmost cells cells could have the easternmost cells as their western neighbours and vice versa). Another option is to define the boundary as having a fixed state. Since the states exposed by the boundary cells would potentially differ between these solutions, this will affect the resulting CA's time steps. An example of the difference is shown in **Fig.3.3**, both figures show the resulting time steps from evolving rule 60 from an initial state consisting of a single cell with state 1 (represented by a black point, cells with state 0 are represented by white points). The figure has one time step per row, and time increasing downwards (A common convention used by i.e. Wolfram[11] that will be followed for visual depiction of 1D CA in the rest of this thesis). **Fig. 3.3a** shows how the evolution progresses with fixed boundary conditions (the boundary has a state of 0), and **Fig. 3.3b** shows how the evolution progresses with wrapping boundary conditions. While the non-wrapping case ”collides” with the boundary, and continues in a periodic creation of triangles, the wrapping case proceeds to create a triangle on the left side, that collides with the the second big triangle from the left, at which point it affects the cell it collides with, creating a different sequence of triangles at the point of collision compared to the fixed boundary-condition case.

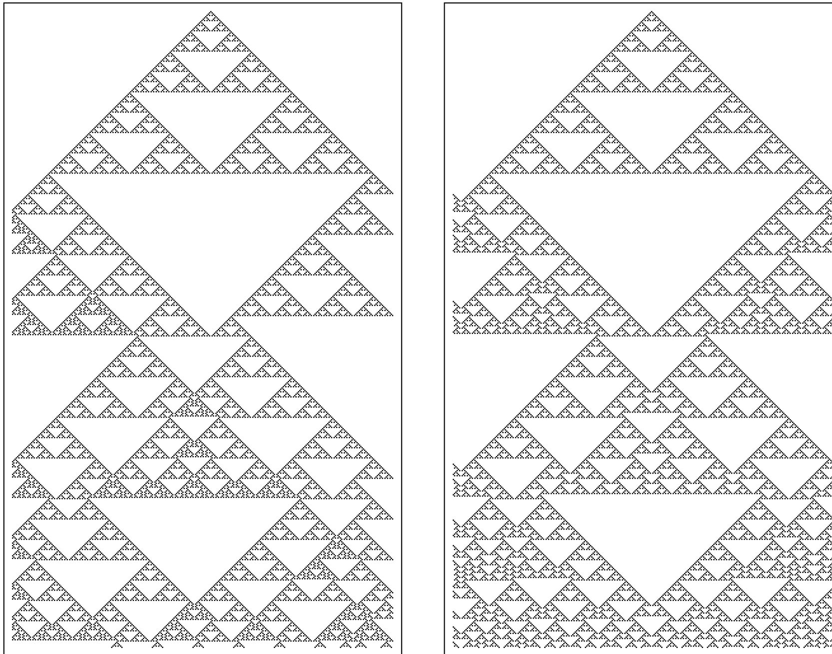


(a) Rule 60 with fixed boundary conditions

(b) Rule 60 with wrapping boundary conditions.

Figure 3.3: Rule 60 evolved with fixed and wrapping boundary conditions

Another example, evolving rule 90 from an initial configuration where the single cell with state 1 is slightly off-centre can be seen in **Fig. 3.4**. Rule 90 produces a fractal shape of triangles. With fixed boundary conditions (**Fig. 3.4a**), the fixed boundary makes the triangles ”bounce” of the boundary, creating ever larger triangles, until those triangles misalign with the central triangle (owing to the single 1 not being perfectly in centre). With wrapping boundary conditions (**Fig. 3.4a**), there is no increase in triangle size at the boundary, instead, there is a slight collision on the left side after the first triangle wraps around, this makes for a different evolution on the left side from then on. This deviation eventually propagates, so that a different result can be seen in the last lines of the figure.

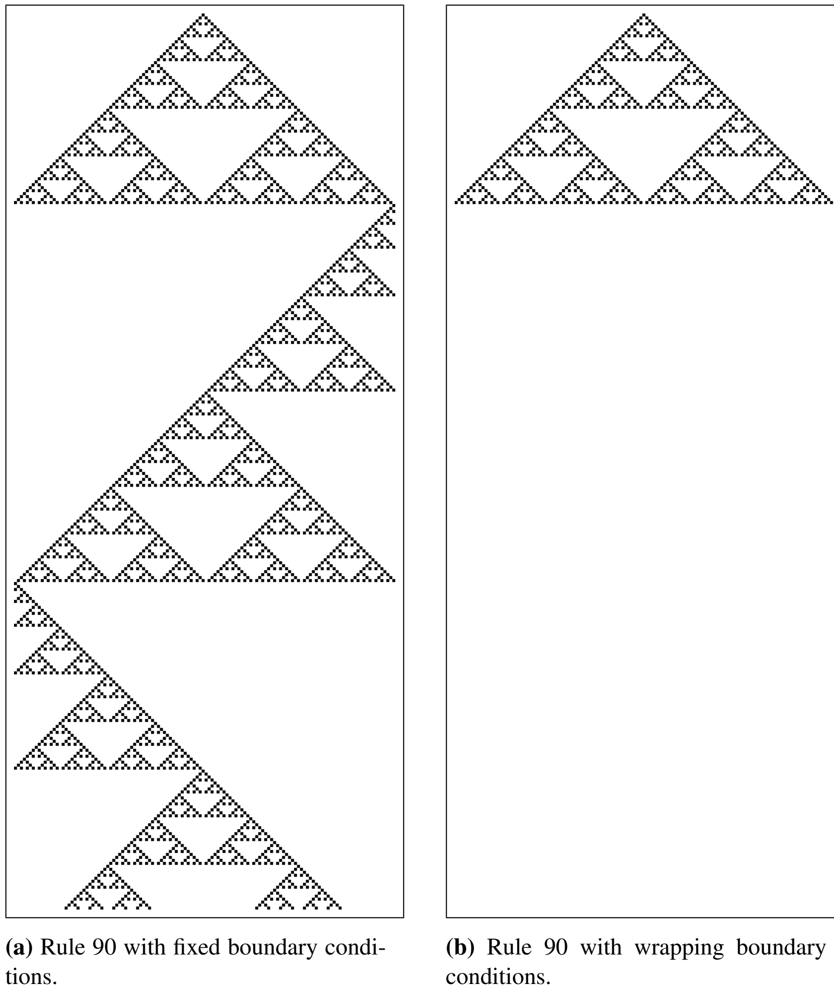


(a) Rule 90 evolved with fixed boundary conditions.

(b) Rule 90 with wrapping boundary conditions.

Figure 3.4: Rule 90 evolved with fixed and wrapping boundary conditions from a single off-centre cell with state 1

An interesting variation on this happens when the single initial 1 cell *is* perfectly centred, as can be seen in **Fig. 3.5**, here the collision with the boundary in the fixed case, makes the triangle production ”bounce” of the right boundary, moving leftwards, until it once again ”bounces” of the left boundary. With wrapping boundary conditions, the ”corners” of the triangle collide with each other, stopping the production of new triangles.

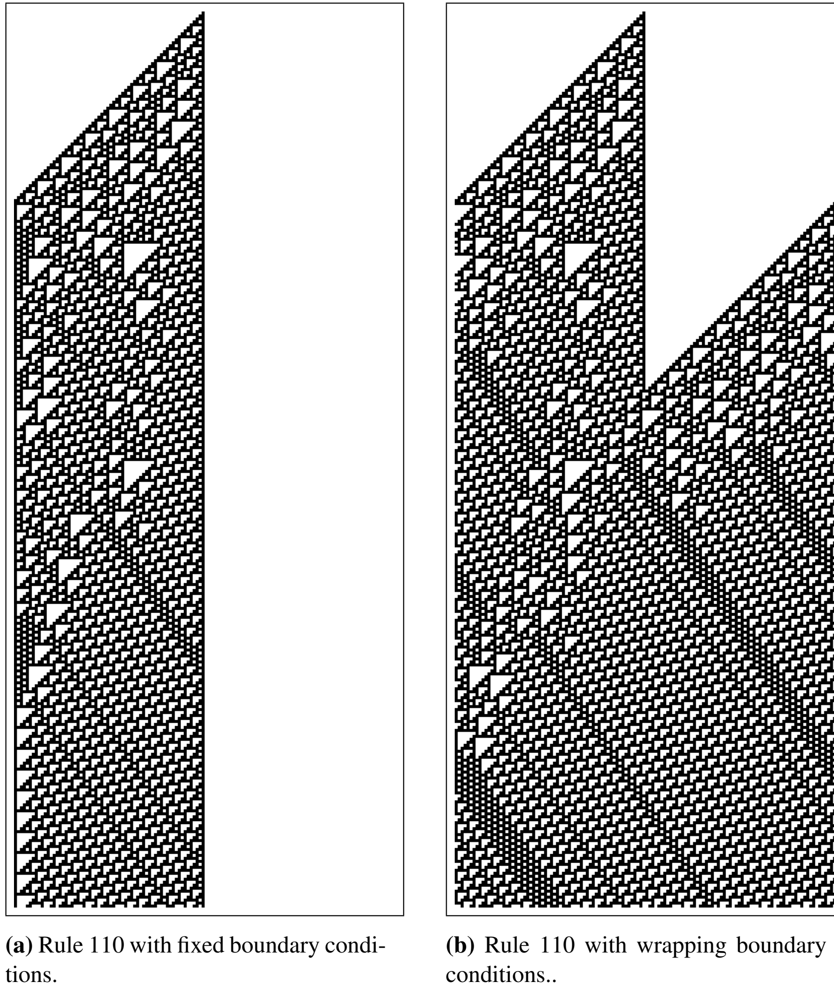


(a) Rule 90 with fixed boundary conditions.

(b) Rule 90 with wrapping boundary conditions.

Figure 3.5: Rule 90 evolved from a centred single cell with state 1, with fixed and wrapping boundary conditions

Finally, **Fig. 3.6** demonstrates Rule 110, with fixed and wrapping boundary conditions, the most obvious difference being that the wrapping boundary conditions very visibly wrap around to the right hand side. However, further inspection also reveals that the interaction with the fixed boundary condition differs from the results of having the additional room to grow in on the right side; as for instance can be seen in the triangles produced at the left boundary once the collision (or wrap-around) happens.



(a) Rule 110 with fixed boundary conditions.

(b) Rule 110 with wrapping boundary conditions..

Figure 3.6: Rule 110 evolved with, and without wrapping boundary conditions

3.4 Von Neumann's Universal Constructor

In his works to create an automaton capable of self-reproduction, von Neumann designed a universal constructor in a 29-state CA[28] (work that was later finished by Burks). This universal constructor consisted of a *constructor unit*, a *tape* containing a description of the constructor unit and a *memory control*-unit for reading/writing the tape. The constructor could construct the machinery described by the tape, including a copy of the tape and the support machinery for reading and writing the tape, thus making the constructor capable of self reproduction. In keeping with the terminology used in this thesis, this CA had $k = 29$, $r = 1$, and a neighbourhood size of 5, the von Neumann neighbourhood. It was later demonstrated by Codd that a universal computer constructor, capable of self-reproduction

and universal computation could be built in a $k = 8$ CA[25].

3.5 Hardware implementations of Cellular Automata

The idea of implementing Cellular Automata in hardware is not a new one, a variety of implementations have been made, with varying degree of reconfigurability. To give a sample of earlier implementations, this section will cover two examples: the CAM6, which is quite close to the design that will be proposed in this thesis, and a pipelined FPGA implementation, which takes a somewhat different approach.

3.5.1 CAM6

CAM6[26] is a hardware implementation of CA, which consists of a hardware expansion card for IBM PC-machines. The module is designed to be configured and controlled by way of FORTH-code. The hardware implementation of CA is divided into two halves: CAM-A, and CAM-B, these two can be configured either to work side by side (creating a wider CA), or to work independently while allowing each cell in one half to react to the cell that is in its corresponding position in the other half (as well as the rest of its neighbourhood). Additionally, each half has two *bit-planes*, which allow for running two sets of state ”on top of each other”, where the cells in the two planes are able to communicate with the corresponding cells in the other layer. These two details allow for configurations where some temporal information is retained (i.e. by letting the second plane contain the previous time step of the first plane, or going even further by using the other half for another 2 planes of temporal state). Another potential use for the multiple layers, is allowing for k-values different from $k = 2$, by storing the additional bits on the other planes. The case of temporal state is a special case of this, since a CA that defines its next state as a function of the state in the previous x steps, would have a state space of $k = 2^x$.

Internally, the CAM6 consists of a set of *plane modules*, each of which have 256x256 cells[26, App. B]. Each of these plane modules are internally pipelined (the cells share a lookup table defining the current rule), while the updating of the system of these modules are parallel.

3.5.2 Pipelined FPGA-implementation

Another possible hardware implementation is a pipelined FPGA-based implementation, an example of which can be found in the implementation by Kobori et al.[29]. In this implementation the FPGA contains the necessary components to evolve time steps for the cells it is processing. The FPGA gets the state of the cells from external memory in rows, and the rows of cells are shifted through the pipeline as it progresses its evolution. Each pipeline stage contains the row, along with the relevant neighbours of the same time step, thus all the pipeline stages beyond the first can evolve a new time step in parallel. The pipeline basically allows the FPGA to evolve multiple time steps for the cells each time they are read from memory, reducing the amount of memory read/writes necessary to complete the process.

Chapter 4

Accelerator Design

This chapter will present the design of the CA-accelerator, starting from the smallest components and building op from there. At the end of the chapter the different ways of configuring the CA at synthesis time will be presented.

4.1 Notation

To allow for a consistent notation the directions in which a cell is neighbouring to another cell will be referred to with compass directions relative to the cell itself. Since the accelerator only allows for $r = 1$ and neighbourhood sizes of 3, 5 or 9 cells, any given cell can thus have neighbours in the western, eastern, northern and southern direction, as well as the diagonal neighbours at the north-east, north-west, south-east and south-west positions. Similarly, to differentiate between the conceptual CA, and the specific synthesized CA, the term Cellular Automata System (CAS) will be used to refer to the latter.

4.2 Design of the CA-accelerator

The CA-accelerator shown herein was originally implemented in VHDL and tested as a stand alone Field Programmable Gate Array (FPGA) CA-implementation during the master project that was performed before the start of this thesis, and was then modified and improved upon during the course of work on this thesis so that it could be integrated as a coprocessor in SHMAC.

4.2.1 Comparison to other implementations

The CA-accelerator's implementation deviates heavily from the design for a CA tile proposed by Rusten et al. in Chapter 9.1.3.5 of the original SHMAC-thesis[23]. Their proposed design uses a master/slave layout of tiles, where some tiles contain state and rules,

while other tiles perform the actions needed to evolve new time steps from these rules. Finally, their design proposes using DMA to allow the other processor tiles to communicate with the CA-tiles. The accelerator presented herein is implemented as a coprocessor local to a specific processor tile, and does not split its design across multiple tiles. The interface for communication is the coprocessor interface of the ARMv3-ISA (as made available through the coprocessor Interface Module (IFM)). Design-wise it bears some resemblance to the CAM6[26]-architecture, in that parts of the CA share a rule table, and that there is some degree of choice about how the boundary condition is configured. However, the CAM6 allows for a more flexible neighbourhood connection system than what is allowed for in this accelerator's design. Other differences can be found in that CAM6, through its layer-stacking approach, allows for configurations with more states than just the binary case. The CA-accelerator has a permanently fixed number of possible states ($k = 2$). Additionally a number of properties of the design of this CA-accelerator are fixed but configurable at synthesis-time, namely the size, the number of cells that update in parallel and the neighbourhood size. These choices stem directly from the choice of a direct implementation of the entire CAS in hardware, instead of for instance using a pipelined approach as suggested in Kobori et al.[29], or a multi-tiled approach as suggested by Rusten et al[23]. The accelerator is designed first and foremost with uniform cellular automata in mind, although some degree of quasi-uniform behaviour is possible.

The choice of a fixed size makes it possible to realize all the cells as actual entities in hardware. This allows the implementation itself to benefit from the locality of communication that CA-cells by definition exhibit, opening a potential for consistent, and potentially quite high rates of time steps per clock cycle.

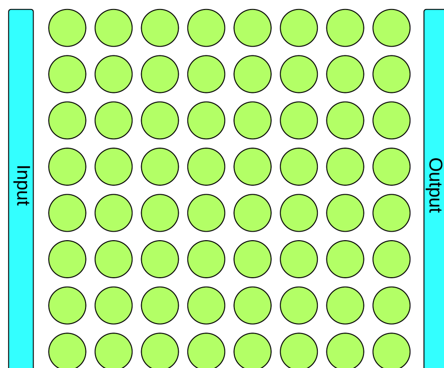


Figure 4.1: The communication possibilities between the CAS and the accelerator interface. Cells can only communicate with their direct neighbours, the input can be used as the western neighbour on the western boundary, the output can be used as the eastern neighbour on the eastern boundary.

To avoid introducing non-local communication, while still allowing for writing in an initial state to all cells, as well as reading out the result of all cells, a solution where only the state at the boundary of the CAS are available to read or modify by the end user was chosen, as demonstrated in **Fig. 4.1**. Writing an initial state into the grid, is possible by setting the boundary condition on the western edge to the wanted value(s), and using a rule

where each cell takes on the state of its western neighbour. Similarly, the eastern boundary is available on the output ports of the accelerator. This means that the state of any cell in the grid can be read by using the same rule to *shift* the wanted cell into the easternmost position, making it available on the output port. This can be done non-destructively if the accelerator is configured to have wrapping boundary-conditions in the east-west direction while performing this shift.

4.2.2 The cell

A single cell has rather simple responsibilities, it must keep its current state available to its neighbours, and upon request update its state according to a set of rules. This also implies that it needs to know the state of its neighbours. Thus a cell has two connections to all of its neighbours, the connection that publishes its current state to its neighbours, and the neighbours' corresponding connection. This is demonstrated in **Fig. 4.2**, where all the cells that belong to the cell's neighbourhood are combined into the rectangle labeled *Neighbours*. These connections in sum allow any cell access to the state of its entire

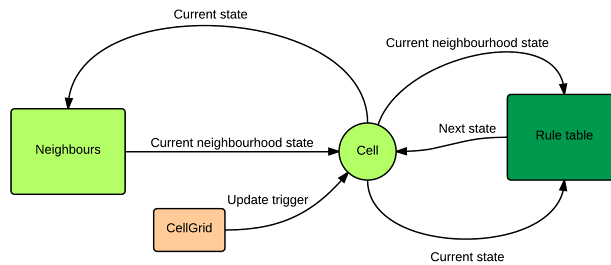


Figure 4.2: The communication responsibilities of a single cell.

neighbourhood. But this alone is not enough, the cell also needs to know what the next state that corresponds to the current state of the neighbourhood is. To allow for this, the cell has two connections to a *rule table*. The first of these connections passes in the combined state of itself and its neighbours to be used as a lookup value in the *rule table*. The second connection is used to allow the cell to get a new state back from the *rule table*. The connections to the *rule table* are time shared with 0 or more other cells, depending on the number of cells that are grouped together in the *cell grid* that the cell belongs to. Finally, the cell has an *update trigger*, that makes it take on the *next state* it got back from the *rule table* as its *current state*.

Since the cell always announces its current state to its neighbours, all cells are always kept up to date with their neighbourhoods state. This allows for a rather simple mode of operation for evolving successive time steps: At some point, the cell's output is connected to the *rule table*, which in the next cycle returns the corresponding next state. When all the cells in the *cell grid* have gotten their chance to read their *next state* from the *rule table*, the *cell grid* triggers their *update trigger*, and cells change their state to the next state. Since

all the cells change their state at the same point, the inputs to their neighbours will also change at that point, completing all the preparations needed for repeating the procedure in the next CA-time step.

4.2.3 The rule table

The *rule table* is designed as a bit-adressable memory, and is used as a lookup table for the cells. For all neighbourhood state bit-strings, it has room for storing exactly one bit defining the corresponding *next state*. One peculiarity of its design, is that it is designed to be read 1-bit at a time and written 8-bits at a time. The reason behind this is that all the use cases for reading this table, are 1-bit reads from the cells, while writes are only performed from the CA-toplevel, and are designed to be 8-bit to reduce the amount of writes necessary to fill a complete table. The table is also able to have all its values reset to a single value using a single operation, allowing for even fewer writes to fill the table, as it is thus only necessary to write the values that are different from this single value.

Since the overall design is made so that the only accessible cells for reading are the westernmost cells, and the only cells available for writing are the easternmost cells, it is necessary to be able to *shift* the cells eastwards rather often, for instance when loading in the initial state of cells, or reading out the states of all the cells. To allow for doing this without having to overwrite the current rules, the *rule tables* have a hardcoded rule for this mode of operation, which simply outputs the part of its input that corresponds to the relevant cell's western neighbour's state as *next state*. This mode of operation is triggered whenever the *rule table* is signalled to work in *load mode*.

4.2.4 The cell grid

A *cell grid* is a row of one or more cells that share a *rule table*. Since the *rule table* can only handle a single request at a time, these cells will have to be updated serially. It is the *cell grid*'s responsibility to handle the sharing of this *rule table* in such a way that the cells get updated with the correct rule, maintaining a behaviour equal to a situation where each cell had their own *rule table*. The *cell grid* is therefore responsible for signalling its cells when they should update, and connecting the proper cell to the *rule table* at any given time during a sequence of updates to the cells.

Since all the *cell grids* in this implementation have the exact same number of cells within them, all the *cell grids* will be updating the same cell (if any) at any given moment, this leads to the opportunity of moving the counting logic needed to keep track of the current cell to update out of the *cell grid* and into a global *grid controller*. As shown in **Fig. 4.3**, when the *cell grid* is updating cells, it gets an index number from the grid controller, telling it which cell to connect to its *rule table*. The corresponding result from the *rule table* is then set as the input *next state* for the relevant cell. The *cell grid* is thus responsible for maintaining the correct *next state* input for each cell (since the cells can't stay connected to the *rule table* output) after it has been read from the *rule table*. Finally, when all the cells have gotten their corresponding *next state* set, the *cell grid* is responsible for signalling all the cells to change their *current state* to the *next state*, completing the evolution of a time step. To keep the design simple and consistent, this propagation is performed as a special case of the normal flow of selecting an indexed cell for updating

where the index that is one larger than the number of cells is used by the *grid controller* to signal that the propagation should take place. Since the normal flow of operation uses two cycles to update a cell (one to deliver the current neighbourhood state to the rule table, one to get a value back), this propagation also uses two cycles.

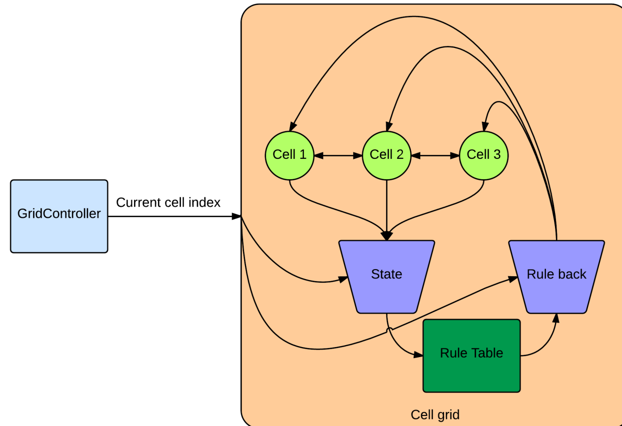


Figure 4.3: The update flow in a *cell grid* with 3 cells.

The *cell grid* also serves as an intermediary between the toplevel interface of the CA-accelerator, and the cells and *rule table* contained within, effectively encapsulating the *rule table* and cells from the outside world. Each *cell grid* is in itself a CA, and exposes the connections necessary to connect itself together with other *cell grids*, as shown in **Fig. 4.4**. All the cells in two neighbouring grids, will get their relevant neighbour input/outputs connected to the neighbouring grid in the direction that the grids are neighbours.

This makes it possible to create a large system of *cell grids*, where all the *cell grids* update in parallel, as demonstrated for six grids in **Fig. 4.5**. The actual implementation is simplified a bit from that shown in the figure, by the fact that one grids output in one direction, is the neighbouring grid's input in the opposite direction.

The examples shown in **Fig. 4.4** and **Fig. 4.5** only demonstrate how this works for a neighbourhood size of 5, but it can also be extended to cover the case of a neighbourhood size of 9 without adding additional I/O-connections. This is possible because the southern neighbour of a cell at position n, is the south-western neighbour of the cell at position $n + 1$ (if positions are increasing eastwards), thus the input from south (i.e. the output of the cell at position n in the grid directly to the south) can be reused as the south-west (and south-east) input. This can be done by allowing multiple cells to connect to the same I/O-connection, shifted by one in the eastern or western direction, depending on whether they need the state information as their south-eastern or south-western neighbour-input. A demonstration of how this works in the south-to-north direction is shown in **Fig. 4.6**, the circles in the I/O rectangles are not cells themselves, but I/O connections to the state that is output by the relevant southern cell. Note how the same connection is used as input to

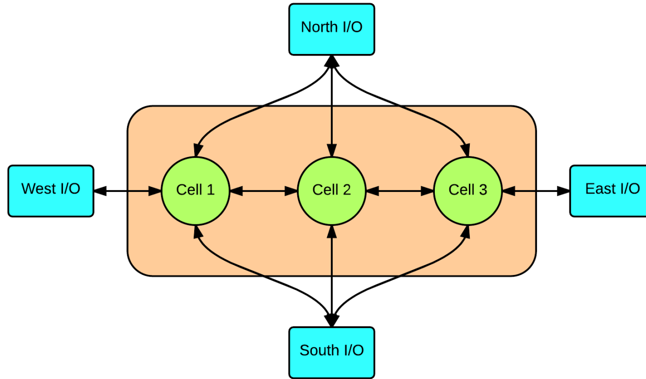


Figure 4.4: The connections exposed by a grid for connecting grids together.

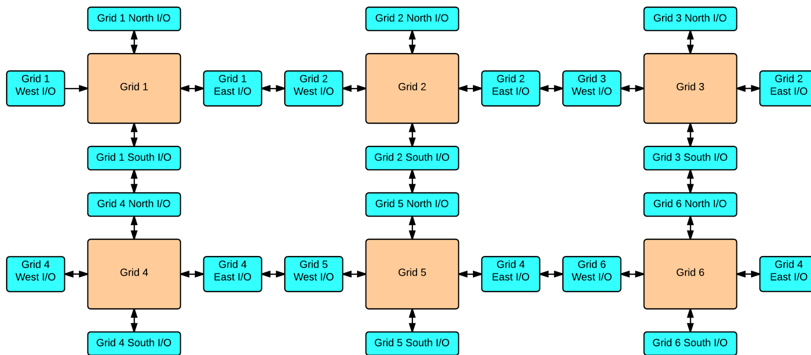


Figure 4.5: Six *cell grids* connected together, updating in parallel with each other.

three different cells. The figure also hints towards how the western-most and eastern-most cells in each grid get connected to the neighbouring grid's relevant I/O-connections. The same solution is used in the north-south direction. It is of course also simple to scale the interconnection down to the case where each cell has two neighbours, by removing the north/south-interconnections.

The grid-connections that are on the boundary, that is, the grid-connections without a neighbouring grid to connect to, are connected up by the accelerator toplevel so that it can decide the boundary conditions to be either a fixed value, or wrap around, so that for instance the cells in the northernmost grid are neighbours with the cells in the southernmost grid (and similarly for the east-west situation).

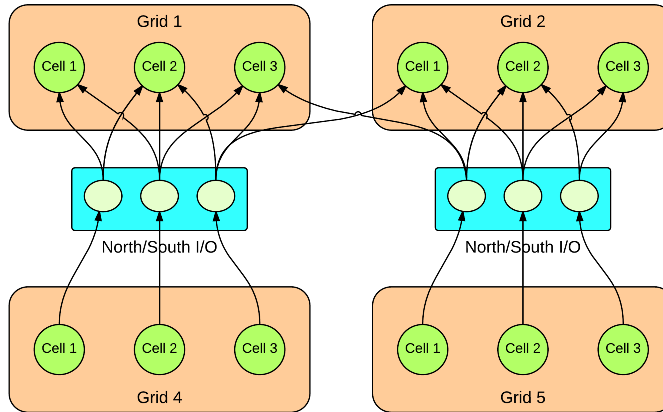


Figure 4.6: A demonstration of how the output information from a southern grid is used by multiple northern nodes when the neighbourhood size is 9

Non-uniformity

The write operations that change the *rule tables* are designed such that all the *rule tables* in the entire CAS get the same input, which is useful when applying uniform rules for all the cells in the CAS. However to allow for some degree of non-uniformity in the CAS, each *rule table* also has a *write-mask*, which behaves as a write-enable flag. Whenever the write-enable flag is set to 0 for the *rule table*, any attempts at writing new rules to the table are simply ignored. This way, different rules can be written to the various cells by way of a sequence of rule-writes followed by changes to the *write-mask* to disable writing to some of the *rule tables*, and then another round of writes that only affect the *rule tables* that should have a different set of rules.

Due to the fact that the cells in each *cell grid* share one *rule table*, the write masks are in effect a per cell-grid property. This means that it is only possible to create a fully non-uniform CA when the number of cells per *cell grid* is 1.

4.2.5 The grid controller

All the *cell grids* that make up the complete system have the same number of cells, and are designed to update the same specific cell within them at the same time as their neighbouring *cell grids*. The grid controller has the responsibility of keeping track of which of the cells within all the *cell grids* are to be updated at this point in time. The *cell grids* could keep track of this themselves, but then the logic would be duplicated in all the *cell grids* for a count that would be globally exactly the same across all the *cell grids*, thus the functionality is upfactored so that only a single entity has to have the logic required to keep track of this. The grid controller is also responsible for keeping track of how many time steps have been evolved, so that the grids will only evolve the wanted amount of time steps. Finally, the grid controller is responsible for signalling the CA-toplevel when

the requested amount of CA-time steps are done, allowing the CA-toplevel to deliver the output, and possibly perform new operations.

4.2.6 The CA-toplevel

The CA-toplevel is the point where the *cell grids* are instantiated, it is responsible for creating the connections between neighbouring *cell grids*, as well as the connection between the *cell grids* and the grid controller. A conceptual view of the components contained within the CA-toplevel, is shown in **Fig. 4.7**. The CA-toplevel is also responsible for managing the additional connections needed to enable or disable *boundary wrapping*. That is, the CA-toplevel is responsible for managing what is connected to the *cell grid* inputs and outputs that are on the boundary of the entire CAS, and thus not directly connecting two *cell grids*. These connections can either be connected to some static value, or connected to each other in such a fashion that the northernmost I/O-connections are connected to the southernmost cells (and vice verca), as well as a similar connection for the easternmost and westernmost I/O-connections, creating wrapping boundary conditions. In the current implementation, the static value on the western boundary of the grid is user configurable, and used for loading in columns, while all the other boundaries are hard coded to 0 unless boundary wrapping is enabled.

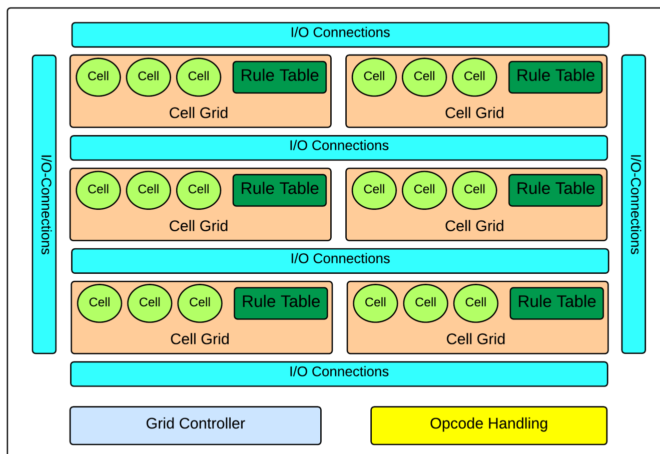


Figure 4.7: A conceptual view of the CA-toplevel, demonstrating the components it contains.

The CA-toplevel is also where the connections to the ”outside” is handled, the CA-toplevel handles all signals that is received through the coprocessor Interface Module (IFM), and decides how to react upon all of those signals. In general, the CA-toplevel reacts to *opcodes*, that have defined behaviour that either configures the system in some fashion (e.g. enabling/disabling the boundary-wrapping), or makes the system perform some action on the cells within (e.g. evolving some time steps). The CA-toplevel is also responsible for delivering output to the IFM, through an output port that can either

hold the easternmost column of the CAS (effectively the output part of the easternmost I/O-connection), or configuration information about the CAS. Finally, the toplevel is responsible for signalling the IFM that it is ready to handle a new opcode, by setting the done-flag of the IFM high.

4.2.7 The coprocessor interface

The coprocessor IFM is the interface between the accelerator and the CPU-core, which allows the accelerator to be used as a coprocessor. The IFM was generated for a specific processor tile type, the amber, by using the `tileGenerator.py`-script[30]. Some manual intervention was necessary to allow the coprocessor to be used; The script creates a modified amber tile, so some name conflicts had to be resolved to allow it to be synthesized together with regular amber-tiles (i.e. tiles without the added coprocessor IFM). The resulting amber tile also had an issue with accepting the coprocessor opcodes `mrc` and `mcr`, since those were set to be illegal operations by default, a problem that was resolved by explicitly allowing those operations in the modified amber tile that contains the CA-accelerator.

The coprocessor IFM was chosen as the accelerators design needs a fair bit of interaction between the software that is using the accelerator and the accelerator to perform some tasks. Specifically, initial configuration and reading the state of the entire CAS. Therefore the coprocessor IFM was a good fit, compared to the other options, which use DMA for interaction. The use of the coprocessor interface to communicate with the accelerator is covered in Chapter 5.

4.2.8 Configurability

At synthesis-time, some details of the accelerator's synthesized *cell grid* can be configured by way of VHDL generics: the size of the grid, the number of neighbours taken into account as part of a cells neighbourhood, and the amount of cells that should share a *rule table* (effectively, the amount of cells that are updated in serial). The relevant VHDL generics for configuring these details are summarized in **Table 4.1**. Of particular note, are the possible neighbourhood configurations, as shown in **Fig. 4.8**.

Since a 2D CAS with neighbourhood size 3 differs from a 1D CAS with neighbourhood size 3 only in the existence of the additional rows of cells, there are effectively 3 different options to choose from: 1D (where only the west-east variant is supported), 2D with neighbourhood size 5, and 2D with neighbourhood size 9. The choice between these can be made by selecting the proper value for `ca_num_neighbours`, a VHDL generic integer that selects the amount of neighbours connected to each cell. As such its value does not count the cell itself, the possible values are therefore 2, 4 and 8, corresponding directly to the configurations shown in the lower part of **Fig. 4.8** from left to right. For consistency, the rest of this thesis refers to the *neighbourhood size* instead of the number of connections, and will therefore continue to use the values previously established to refer to these neighbourhood sizes; 3, 5 and 9.
Note that while the value of `ca_num_neighbours` does limit the amount of cells connected to each other cell, creating an *available neighbourhood* of the specified size, the

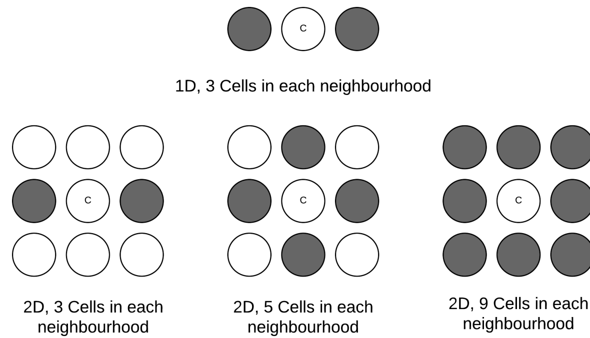


Figure 4.8: The three different neighbourhood size configurations available in the accelerator at synthesis time.

effective rules are free to ignore as many of these connections as they want. The neighbourhoods with larger sizes are supersets of the ones with smaller sizes, and thus capable of evolving time steps according to rules for the smaller neighbourhoods, by extending the rules to the synthesized size while ignoring the additional inputs, further details about this will be given in Chapter 5.5.

Configuring the size of the synthesized CA, is dependent on three VHDL generics: `ca_cells_per_sub`, `ca_num_grids` and `ca_height`. The effective size of the CA has `ca_cells_per_grid*ca_num_grids` columns, and `ca_height` columns. The amount of cells in each *cell grid* is decided by `ca_cells_per_sub`, which thus directly allows for deciding how many cells are updated in serial in each row.

VHDL-generic	**Determines the number of**
`ca_num_grids`	*Cell grids* that each row in the CAS will contain.
`ca_height`	Rows in the CAS
`ca_cells_per_sub`	Cells in each *cell grid* (Number of serially updated cells).
`ca_num_neighbours`	Neighbours a cell is connected to other than itself (2, 4, 8).

Table 4.1: The VHDL generics available for configuring the size, neighbourhood and amount of serial updates at synthesis time.

Each *rule table* has to be able to fit 2^s bits of information (where s is the number of cells in each neighbourhood) , which does not scale too well when going from 3 to 9 cells per neighbourhood, since each *cell grid* contains one of these. Therefore the design allows for adjusting the degree of parallel updates down in this fashion, so that larger CA can be synthesized at the cost of the number of time steps that can be evolved per clock cycle.

All of these VHDL generics are adjusted in the code where the IFM instanciates the accelerator, some manual adjustment of the register mappings for the IFM is neccessary whenever a 32-bit boundary of height is crossed, as the I/O mappings of the coprocessor IFM must use 32-bit coprocessor registers for its communication.

Chapter 5

Using the accelerator

An accelerator is not very useful unless there is a way to communicate with it. This chapter will explain the interfaces exposed for communicating with the CA-accelerator. In particular, the ARMv3-assembly-level communication will be covered in Chapter 5.1, and a C interface is demonstrated in Chapter 5.4. Common details regarding how rules are defined is the topic of Chapter 5.5.

5.1 Coprocessor interface

Since the CA-accelerator is connected to a coprocessor IFM, the accelerator is available for use from ARMv3 assembly through the coprocessor register read/write opcodes *mrc* and *mcr*. The coprocessor number is 14, and the most relevant coprocessor register is *cr15*, which is used for writing the *options*-field and reading the *done*-flag, as specified in the IFM-description[30], and *cr0*, which is where output from the accelerator can be read.

The CAS can have a variety of sizes, and all input and output of cell states are designed to happen by reading the easternmost column, or writing to the boundary-conditions of the westernmost column. Therefore it is possible to create configurations where the 28 available bits of input in the *operations*-field would not be enough to fully define the state of the CA, and where the 32 bits of available output in *cr0* would not be enough to completely read out the entire state of the CA. For those arguments that can't fit completely in *cr15*, the necessary amount of registers starting from *cr0* and up will be used instead. Note that some minor adjustments are necessary in the IFM files to enable the usage of the coprocessor registers beyond *cr1*, as each additional register needs a few lines of manual mapping between the accelerator and the IFM.

As will be covered in the next section, the CA-accelerator itself is controlled by *opcodes*, which define an operation or configuration change that it should perform. These are performed by writing them in the *options*-field (i.e. *cr15*). The IFM-implementation is designed such that a write to this coprocessor register has the side effect of also setting the *start-flag* high, effectively telling the CA-accelerator to handle the opcode immediately. Since the CA-accelerator is designed to only handle opcodes when the start flag is

high, and the IFM is designed to pull the start flag low again once the done-output goes low (as a signal that the accelerator is working), no further involvement is needed from the programmer's side to avoid the accelerator from performing the operation defined by the opcode more than once. If however the programmer wants the accelerator to react to the same operation more than once, it is necessary to write to *cr15* multiple times. The CA-accelerator signals that it is done performing its opcode by setting the *done*-flag to 1, which is readable from *cr15*. No further opcodes should be attempted unless this flag is 1.

At assembly-level, the rough flow of using the coprocessor-interface to the accelerator, is as follows:

- Wait until a read from *cr15* results in a 1, using *mrc*
- Write any CA-toplevel arguments to *cr0* (and up), using *mcr*
- Write a CA-toplevel opcode to *cr15* using *mcr*
- Wait until a read from *cr15* results in a 1, using *mrc*
- Read any output from *cr0* (and up) using *mrc* if needed.

Repeating this as necessary if more opcodes need to be performed. As previously mentioned, the accelerator is designed to react to a set of *opcodes* that control its operation. The available opcodes, along with the sizes of their arguments are shown in **Table. 5.1**. h is the height of the CA, n is the size of the neighbourhood taken into account for each cell. For space reasons, *opcode* has been shortened to *op* in the table. Further details regarding the arguments will be given in Chapter 5.2. Note that while most of these argu-

Op	**Name**	**Arg 1**	**Arg 2**	**Arg 3**	**Arg 4**	**Arg 5**
000	OP_HALT					
001	OP_RUN	iterations(28)				
010	OP_SETMASK	fullRow(1)	fullCol(1)	value(1)	maskX(9)	maskY(9)
011	OP_GETINFO	whichInfo(1)				
100	OP_SETRULE	value(8)	key(n - 3)			
101	OP_SETEDGE	modeEW(1)	modeNS(1)	value(h)		
110	OP_LOADCOL	times(28)	value(h)			
111	OP_RST	value(1)				

Table 5.1: List of available opcodes in the accelerator, along with the sizes of the arguments.

ments are delivered through *cr0*, the operations that need to deliver h (*height*)-sized input, *OP_SETEDGE* and *OP_LOADCOL*, get those arguments from *cr0* (continuing in *cr1* and up if the *height* is larger than what can fit in a single 32-bit register). The ordering of the arguments are ascending from most significant bit to least significant bit, using the 3 most significant bits for the opcode, and the rest for arguments (`OP_SETEDGE` is an exception to this rule, due to a slight oversight that was made when the opcodes were modified to use *cr0* for h-sized input, it has its arguments at bit 25 and 24).

5.2 Description of operations

A detailed description of the various accelerator-level *opcodes*' behaviour and argument-layout follows below. Each *opcode* takes 0 or more arguments (either as part of the opcode, or in a few cases through *cr0* and up), and performs some operation. The instruction is triggered by writing to *cr15*, thus if more than one coprocessor register is used for input, the other registers have to be written to before *cr15*. Additionally, opcodes should only be applied when the *done*-flag is high, which can be queried by reading *cr15*.

5.2.1 OP_HALT

Arguments: None.
Operation: None, used as a No Operation (NOOP)-instruction.

5.2.2 OP_RUN

Arguments: iterations, 29 bit integer. (bits 28-0)
Operation: Evolves the CAS with the current configuration for *iterations* time steps.

5.2.3 OP_SETMASK

Arguments: fullRow (bit 28), fullCol (bit 27), value (bit 26), maskX (bits 25 - 17), maskY (bits 16 to 8)
Operation: Enables/disables rule writes for specific cell grids
This modifies the *write mask* for one or more cell grids, allowing for locking the current rule table in use in those cell grids, thereby making some degree of non-uniformity possible. The cell grid that will have its write mask changed is specified by *maskX* and *maskY*, and its new state is defined by *value* (1 is write-enable, 0 is write-protected).

- If *fullRow* is 1, then all write masks with Y=*maskY* will be set to *value*.
- If *fullCol* is 1, then all write masks with X=*maskX* will be set to *value*.
- If both *fullRow* and *fullCol* are 1, then every mask will be set to *value*.
- If both *fullRow* and *fullCol* are 0, then only the single mask for X=*maskX*, Y=*maskY* will be set to *value*.

Note that coordinates are defined so that $x = 0, y = 0$ is in the south-eastern corner of the accelerator.

5.2.4 OP_GETINFO

Arguments: whichInfo (bit 28)
Operation: Sets the output to be one of two pieces of information about the accelerators synthesized configuration. The result is 2 16-bit integers (bit 31-16, and bit 15-0 in the output)
When whichInfo is 1, the result is:

- bit 31-16: `ca_num_grids` - The number of cell grids in the CAS.
- bit 15-0: `ca_height` - The height of the CAS.

Otherwise, the result is:

- bit 31-16: `ca_cells_per_sub` - The number of cells that each cell grid contains (the total horizontal amount of cells is this value multiplied by the amount of grids).
- bit 15-0: `ca_num_neighbours` - The number of neighbours taken into account in this accelerator, can be 2, 4 or 8.

5.2.5 OP_SETRULE

Arguments: value (bit 28 to 21) key (bit 20, n - 3 bits)
Operation: Set a rule in the active rule tables
A rule is specified by its state configuration and the corresponding next state. This opcode takes in a rule as a combination of the upper bits of the state as *key*, and the sequence of states corresponding to the 3 lowest bits of the state as *value*. In this fashion, if the neighbourhood size is 3, then 0 key bits will be used, and thus those rules can be set directly by Wolfram number. Similarly, this allows for a setup where the program that is using the accelerator stores a rule for neighbourhood sizes of 5 or 9 as an array of bytes, effectively making a 64 or 512-bit sequence describing the rule that way.

5.2.6 OP_SETEDGE

Arguments: eastWest (bit 25) northSouth (bit 24) value (height bits from *cr0* and up)
Operation: Configures the boundary conditions of the CAS.
eastWest and *northSouth* enables wrap-around on the relevant boundaries when 1. If wrap-around is not enabled, then the *value* decides what the constant value on the western boundary should be, all other boundaries will be 0.

5.2.7 OP_LOADCOL

Arguments: times (29 bits), value (height bits from *cr0* and up),
Operation: Loads a column defined by *value* into the accelerator *times* times.

The load is performed by putting *value* on the westernmost boundary of the CAS, and thus it has behaviour dependant on the current boundary-configuration:

- If boundary-wrapping is disabled, then the shifting will result in $value$ being placed in the $times$ westernmost columns, shifting the preceding content of the CAS eastwards while doing so. This results in the $times$ easternmost columns being lost as it is shifted over the eastern boundary of the CAS.

- If boundary-wrapping is enabled, this will merely shift the current contents of the CAS *times* times in a rotating fashion.

  This rotation can also be performed by using OP_SETEDGE with the same value (i.e. with wrapping boundary conditions). And then running OP_RUN for *times* iterations while having the rule set to a rule that makes the cells take on their western neighbour's value. But this instruction allows for doing the same operation without having to overwrite the rules. The shifting result of using this operation with wrapping boundary conditions is also useful for reading the entire state of the grid, as the easternmost row will be available in the output registers, so a sequence of w such opcode calls, where w is the width of the CAS is enough to read out the state of every cell in the grid. Since the rules are not overwritten, and the cells will be shifted back into their previous position, the cells are then ready to continue their operation again without having to have their rules written again.

5.2.8 OP_RST

Arguments: value (bit 28)
Operation: Resets all the cells, and all the next states in the rule tables to *value*.

5.3 Example of assembly usage

The code in **Listing 5.1** demonstrates how to make the accelerator evolve rule 30 for 512 time steps in a 1D CA configured to have a neighbourhood size of 3, and a size of one row and eight columns. The initial state is such that all cells have state 0, except the westernmost cell. The resulting state of the easternmost cell is stored in *r0* when the program completes.

Listing 5.1: An example of assembly code that makes the accelerator evolve rule 30 for 512 time steps.

```
    ldr r0, =0xe0000000 @ OP_RST(0)
    bl runCellFunc
    ldr r1, =0x00000000
    ldr r0, =0xa0000000 @ OP_SETEDGE(0,0,0)
    bl runCellFunc
    ldr r0, =0x83c00000 @ OP_SETRULE(30)
    bl runCellFunc
```

```
    ldr r1, =0x00000001
    ldr r0, =0xc0000001 @ OP_LOADCOL
    bl runCellFunc
    ldr r1, =0x00000000
    ldr r0, =0xa3000000 @ OP_SETEDGE(1,1,0)
    bl runCellFunc
    ldr r0, =0x20000200 @ OP_RUN(512)
    bl runCellFunc
    b endProg
runCellFuncLoop: @ Busy wait for the done flag to go high
    ldr r10, =0x00000001
    mrc 14, 0, r8, cr15, cr5
    cmp r10, r8
    bne runCellFuncLoop
    mov pc, lr
runCellFunc: @ When done, perform the action specified by r0
    mov r9, lr
    mcr 14, 0, r1, cr0, cr5 @ Load the arguments, if any.
    mcr 14, 0, r0, cr15, cr5
    bl runCellFuncLoop
    mrc 14, 0, r0, cr0, cr5 @ Place the result into r0
    bl runCellFuncLoop
    mov lr, r9
endProg:
    b endProg
```

Note that the use of *cr5* as the final argument for the *mcr*/*mrc*-opcodes is arbitrary, and just follows the same choice as shown in the examples of communication with the coprocessor IFM by Teilgård[30, p. 76], who also states that the last argument is ignored by the IFM.

5.4 C interface

Using the accelerator directly through its opcodes follows a repeated flow of supplying some opcode to the accelerator when it is ready, then waiting until it is ready again to repeat the process. The exact layout of opcodes is also directly dependant on the configuration of the accelerator, as some of the opcodes have variable width fields depending on the number of neighbours and the height of the CAS. To remove the need to maintain the busy wait-loop by hand, as well as to allow code that is portable across different configurations of the accelerator, a C interface to the accelerator has been written. The available functionality in this interface is listed below:

- `void cam_init()`
  Initializes the internal configuration of the interface by querying the accelerator about its configuration. As the interface depends on knowing the width and neighbourhood size of the CAS, this function must be run once before any other C interface functions are called.

- `void cam_wait_for_done()`
  Busy waits until the accelerator is done and ready for a new opcode.
- `void cam_reset_to(unsigned int value)`
  Performs an OP_RST, resetting the state of all cells and rules to *value*
- `void cam_set_edge(int modeEW, int modeNS, int value)`
  Performs and OP_SETEDGE, setting the boundary conditions according to *modeEW*, *modeNS* and *value*.
- `void cam_load_column(unsigned int value, unsigned int times)`
  Performs an OP_LOADCOL, loading a column specified by *value*, *times* times into the CAS. Disables boundary-wrapping as a side-effect.
- `void cam_load_big_column(const char *column, unsigned int times)`
  Performs an OP_LOADCOL, loading a column specified by the C-string *column*, *times* times into the CA. The string is allowed to be shorter or longer than the height of the CAS, if it is shorter, then the missing values will be taken as leading 0s, if it is longer, then the excess values will be ignored at the end of the C-string. Disables boundary-wrapping as a side-effect.
- `void cam_shift_column(unsigned int times)`
  Rotate the CA *times* times to the right. Enables boundary wrapping in all directions as a side effect.
- `void cam_run(unsigned int generations)`
  Evolves the CA for *generations* time steps, with the currently specified rules and state.
- `void cam_set_mask(unsigned int fullRow, unsigned int fullCol, unsigned int value, unsigned int maskX, unsigned int maskY)`
  Sets the rule-mask, using OP_SETMASK, using the same arguments as that opcode, see Chapter 5.2.3 for details. Note that this might have side effects on other functions, as the rest of the C interface is written with the assumption of a uniform CA, some care is thus needed when applying masks.
- `void cam_set_rule(unsigned int key, unsigned int value)`
  Sets a single byte of rule data, following the key-value concept. If the CAS has a neighbourhood size of exactly 3, then this is enough to set a full rule, otherwise multiple calls are necessary.
- `void cam_set_wolfram_rule(unsigned int rule)`
  Sets the rule of the CA to be the rule corresponding to Wolfram number *rule* in the west-east direction, handles all neighbourhood sizes by ignoring the neighbours in all other directions.
- `void cam_set_wolfram_rule_ns(unsigned int rule)`
  Same as `cam_set_wolfram_rule`, except that the neighbours that are taken into

consideration are the north and south neighbours, thus it only works for neighbourhood sizes of 5 and 9.

- `void cam_set_rule_from_array(unsigned char *rule_array)`
  Sets the rule of the CA to the rule described by *rule_array*, with keys matching the array indices.

- `unsigned int cam_read_output(int index)`
  Returns the easternmost column of the CAS, can be combines with `cam_shift_column` to read out the entire state. *index* specifies which of the 32-bit *cr*-registers to read from, where the lower register numbers correspond to the southernmost cells.

- `unsigned int cam_get_row(unsigned int row)`
  Returns up to 32 bits of state from the specified *row*, starting from the eastern most position, going westwards. Enables boundary wrapping as a side effect.

- `void cam_get_full_row(unsigned int row, unsigned int *dst)`
  Stores the states of all cells in the specified *row* into the array pointed to by *dst*. Enables boundary wrapping as a side effect.

- `void cam_print_grid()`
  Prints the current state of the entire CAS column by column to stdout. Enables boundary wrapping as a side effect.

- `void cam_print_grid_rotated()`
  Prints the current state of the entire CAS row by row to stdout. Enables boundary wrapping as a side effect.

- `void cam_create_multi_gen()`
  Evolves the CA for *height* time steps, keeping track of the state of the northernmost row in each time step, then loads these states into each row, such that the oldest time step is northernmost, and the latest time step is southernmost. Intended to be used when multiple time steps of 1D CAs are needed at the same time. Enables boundary wrapping as a side effect.

- `void cam_load_single_north_east_1()`
  Performs the operations necessary to set the initial state of the entire CAS to all 0, except for a single 1 positioned at the north-eastern-most position. Uses `cam_load_big_column` for this purpose, and thus disables boundary wrapping as a side effect.

- `void cam_load_full_grid(const char *dst)`
  Uses `cam_load_big_column` to fill the entire CAS with the state defined by the C-string *dst*. The format of the string is the same as for `cam_load_big_column`, with the only change being that successive columns follow immediately after each other in the string, with no 0-termination before the end of the entire string. Enables boundary-wrapping as a side effect.

- `void cam_get_full_grid(char *dst)`
  Reads the state of the entire CAS, using `cam_read_output`, storing the result to

the C-string pointed to by `dst`. The format of the output is suitable for use by `cam_load_full_grid`. Enables boundary-wrapping as a side effect.

- `void set_rule_to_game_of_life()`
  Sets the rule of the CA to Conway's Game of Life.

There are also two helper functions available to simplify entry, and display of data:

- `void hex2bin(unsigned int hex, char *result, int size)`
  Converts a hexadecimal value (or indeed any unsigned int) to a c-string of its binary representation. Useful for working with the results from `cam_read_output()`.

- `unsigned int bin2hex(char *hex, int size)`
  Converts a c-string representation of a binary value to hex (or indeed an unsigned int), useful for working with `cam_load_column()`.

A simple example that evolves rule 30 from an initial state with only the south-westernmost cell having state 1 for 50 time steps, and then prints the result to stdout is shown in **Listing 5.2**.

Listing 5.2: An example of using the C-interface to evolve 50 time steps of rule 30

```
void example1() {
    // Initialize the C interface
    cam_init();
    // Reset the CA state and rules to all 0.
    cam_reset_to(0);
    // Disable boundary-wrapping
    cam_set_edge(0, 0, 0);
    // Load a column into the CA
    cam_load_big_column("1", 1);
    // Set the rule set to Wolfram rule 30.
    cam_set_wolfram_rule(30);
    // Enable boundary-wrapping
    cam_set_edge(1, 1, 0);
    // Run 50 time steps.
    cam_run(50);
    // Print the resulting state of the CA row-by-row.
    cam_print_grid_rotated();
}
```

The C interface provides a few convenience functions, for reading out the states of the entire CAS, as well as for setting the states of the entire CAS. While the most space efficient storage of the states would be a bitstring, the interface is designed to (mostly) use regular C-strings for describing states and rules, as this allows for easier printing, as well as being easier to work with in text editors. Since the interface is used as a proof-of-concept for the accelerator, the added overhead of converting back and forth between the bitstrings used by the accelerator-opcodes and C-strings is ignored.

It is worth mentioning the `cam_create_multigen`-function, which follows the approach for running 1D CA in a 2D CAS suggested by Toffoli et al.[26, p. 95], in that it configures the initial state in such a fashion that only one row effectively evolves *new* time steps, while the other rows contain the results from the previous time steps. This is done by first evolving h time steps according to the initial configuration and rule, storing these externally, and writing them back into one row each, so that the configuration after running the function has one row corresponding to each of the h time steps. Subsequent time steps of the 2D CA will then continue to follow this pattern, where one row contains the new time step, and the other rows will ”echo” the preceding time steps, as a result of their initial configuration being preceding time steps of the row creating new time steps.

Toffoli et al.[26, p. 95] suggests three ways to make use of a 2D CA for evolving time steps of a 1D CA: Running multiple separate rows of a 1D CA, running only one row, keeping the previous time steps available in the other rows, and finally configuring the CA so that the boundaries wrap in a ”spiral fashion”, turning the multiple rows of the CA into a single row for the purpose of the 1D CA. Of these options, `cam_create_multigen` allows for the history-variant. The independent rows variation can be followed by applying a 1D rule extended to a 2D format (which is necessary in both cases), by using the `cam_set_wolfram_rule`-function (or optionally `cam_set_wolfram_rule_ns` if the rule should be followed by each column instead), and then setting the rows (or columns) to separate initial states (i.e. with `cam_load_big_column`). The final possibility suggested by Toffoli et al. (wrapping the boundaries in a spiral fashion) is not possible in the current implementation of the accelerator, as the configuration of *how* the boundaries wrap is not configurable.

5.5 The layout of a rule

To be able to define a rule for use in the accelerator, a description of the format that the rules are described in is necessary. The scheme chosen for the accelerator builds on the basic Wolfram number-scheme presented in chapter 3.2.4. In that scheme, rules for $r = 1$, $k = 2$ CA were defined by the number that resulted from listing all the possible neighbourhood states and the corresponding state of the central cell in the next time step in descending order. The rule's number was then found by reading the resulting binary number made from the corresponding states.

In such a scheme, it is theoretically possible to let any of the neighbours in the neighbourhood map to any of the positions in the 3-bit bitstring that describes the current state, however, to follow the approach used in i.e. Wolfram's examples[11, Chap. 3] in a way that the results can be comparable, a specific ordering should be chosen. This particular order is West, Centre, East from most significant bit to least significant bit, where Centre denotes the central cell that is to take on the *next state* in question. It would also be possible to take on the reverse ordering, which would lead to mirrored results.

Since neighbourhood sizes larger than 3 is possible with this accelerator, namely the 2D CAs with neighbourhood size of 5 or 9, it is important to define a consistent way to describe a neighbourhood state in such neighbourhoods. This has been done by expanding the notation used for the 1D, $r = 2$, $k = 1$ case, such that the West, Centre, East order is kept intact. The additional neighbours directions are added on either side of the original

1D notation, making the case of including diagonal neighbours (neighbourhood size 9) a further expansion of the case of neighbourhood size 5, as listed in **Table 5.2**.

3 cells				W	C	E			
5 cells			N	W	C	E	S		
9 cells	NW	SW	N	W	C	E	S	NE	SE

Table 5.2: The relative order of the neighbours in the bitstring that define a neighbourhood state. The shorter bit strings are centred to demonstrate how the larger neighbourhood sizes expand upon the case of a neighbourhood size of 3 in both ends of the bitstring.

This has the benefit of keeping the relative ordering of the various neighbours fixed when increasing the neighbourhood size. However it is important to keep in mind that the accelerator is designed to have its rules set one byte at a time. When the accelerator is configured as a 1D CA with a neighbourhood size of 3, then this byte corresponds directly to a Wolfram number. When the CA is configured to have neighbourhood sizes of 5 or 9 cells then this becomes a bit more complex.

Rules stored in these cases are still stored a byte at a time, but since this byte only describes the state of 3 cells, and the 5 and 9 cases have 2 and 5 additional cells in their neighbourhoods respectively, some further considerations are necessary. To store a rule in these cases, a *prefix* must be passed to the accelerator, consisting of additional bits of state, referred to as the *key* in the implementation, as well as the byte, referred to as the *value*. Together, these completely describe the next state for 8 preceding neighbourhood states. As a matter of convention, the byte always describes the next state mapping to the 3 least significant bits of the preceding neighbourhood state, following the order listed in **Table 5.2**.

As an example, to make a CA synthesized with 9 cells per neighbourhood follow rule 30 in the west-east direction, the effective rule is the same as listed in **Table 3.1**, with the addition of 6 more bits of state that the rule doesn't take into account, thus the 8 bits of *value* has to be the same for every combination of these 6 bits. The rule can then be stated as X X X rule 30 X X X, where X denotes "don't care". Since the 3 bits that *are* taken into account are not positioned in the 3 least significant bits of the rule, the byte of *value* that will be stored for any key of this rule has a value of either 11111111_2 or 00000000_2 depending on whether the bits corresponding to West Centre and East in the *prefix* map to a *next state* of 0 or 1. A similar layout can be used for applying the rule in the north-south direction, or in general to apply a rule that takes fewer neighbours than the available synthesized connections allow for.

This prefix + value layout very neatly maps to usage from C code, since a complete rule set can be handled in C as an array of 1, 4 or 64 bytes (depending on the neighbourhood size). In this way, the bit-sequence made up of these array elements can be seen as an extended form of the Wolfram-number notation used for the case with 3 cells in a neighbourhood.

Finally, it might not always be necessary to write all the key-value pairs, as OP_RST (and its C-counter part `cam_reset_to()`) resets every entry in the rule tables to the value passed to them, thus with careful choice of reset-value the only key-value pairs that needs

writing afterwards, are the ones that have values different from all 1 or all 0 (depending on the choice of reset value).

Chapter 6

Testing and Demonstration

In this chapter, the approaches taken to testing the accelerator will be covered. Multiple different applications will also be demonstrated by using the accelerator to either apply known CA rules to solve some problem, or by using a genetic algorithm to attempt to find a CA rule to solve some problem.

6.1 Software CA implementation

During development of the original FPGA CA-implementation that the CA-accelerator was adapted from, a software CA-implementation was created as a reference implementation. The main purpose of the software implementation was two-fold, first, it allowed for a simple way to test out some design ideas, secondly, it allowed for evolving uniform CA in software, creating a useful reference to compare the results from the accelerator with. In addition to the simulator itself, a few useful tools to process the the textual output from the evolution of time steps were also written. In particular, tools to convert the resulting time steps to image-representations. These conversion tools, in combination with some manual image editing, were used to generate the time step images that will be shown in this chapter.

6.2 Simulation vs FPGA-tests

Before synthesizing the resultant SHMAC setup for use on an FPGA, a limited run of tests were performed in ModelSim, to make sure that the CA-accelerator behaved well after integration.

The tests performed in ModelSim were created by extending the software CA implementation, so that it was able to output the necessary ARMv3-assembly needed to make the accelerator perform the same actions that the software CA implementation performed. This was done by means of introducing a *pseudo assembly* set of keywords that mostly

mapped to the opcodes available in the CA-accelerator, together with a few macro keywords that allowed for adding comparisons with the results found by the software implementation. This way, a test could be defined in the pseudo-assembly, and then the software implementation would run the test, replacing the macros with comparisons against the results it found for the test, as well as replacing all the other keywords by proper ARMv3-assembly. A short example of pseudo-assembly code that would set the rule in the accelerator to rule 30, configure the intial state to contain a single cell of state 1, and run it for 10 time-steps, is shown in **Listing 6.1**.

Listing 6.1: An example of pseudo-assembly for evolving 10 time steps of rule 30, comparing the results to those of the software implementation afterwards.

```
#height(1);
#neighbours(2);
#width(8);
OP_RST(0);
OP_SETEDGE(0,0,0);
OP_SETRULE(00011110);
OP_LOADCOL(1,1);
OP_SETEDGE(1,1,0);
OP_RUN(10);
OP_VERIFY_ALL();
OP_MAKEFUNC();
```

In this example, `OP_VERIFY_ALL` makes the software implementation output additional ARMv3-assembly that compares all the columns at that point with the state of the same columns in its software evolution of the same configuration at that point. `OP_MAKEFUNC` generates a block of common code that the other pseudo-assembly operations use for writing operations, waiting for the accelerator to finish, and reading and verifying the results. The #-macros use the same conventions that the VHDL-generics used for configuring the accelerator at synthesis time uses, and are there to tell the software implementation how it should configure itself when performing its evolution, as well as to allow it to expand macros like `OP_VERIFY_ALL` correctly and output the operand-fields of the ARMv3-assembly with the correct operand sizes.

The layout of the operations were changed after the tests in pseudo-assembly were completed, to allow for handling larger columns, as well as to move the variable width operands to the least significant end of the opcode-bitstring, so that the fixed sized operands could have fixed positions in the opcode-bitstring. The pseudo-assembly code was not updated at this point, as the focus had moved to testing from C. Therefore the example above would require some adaptation of the conversion code to be able to run on the current version of the accelerator.

The resulting ARMv3-assembly was then compiled and converted to a .hex-file suitable for use with the testbench that was available with the SHMAC-source code. Since the assertions in the pseudo-assembly code were configured to halt with a specific value written to *r0*, manual inspection of the state of this register allowed for verifying that at least on a simulation level the tests that were run seemed to behave as they should.

Finally it is worth mentioning that during testing in ModelSim, it was observed that

with a fully parallel accelerator, multiple time steps could be evolved in the time between two ARMv3-instructions on the CPU, ”measured” by looking at the points in time where the program counter changed. While not an exact test, it does give a rough idea about the relative speed.

6.3 Realview setup

While simulated tests allow for following the logical flow quite well, the scope of such tests are limited by the speed at which such tests can run. Thus, after some initial tests had been performed in ModelSim, focus moved on to synthesizing a SHMAC-configuration on an FPGA, to run further tests on such a setup. The platform used for such testing was a RealView Platform Baseboard for ARM11 MPCore (PB11MPCore), which contains a host system (an ARM11 MPCore), and an FPGA[31]. The host system runs the Debian flavour Raspbian as a host OS, with the necessary configuration to allow for communicating with the synthesized SHMAC core via a virtual TTY-channel and is loaded with various custom scripts that allow for uploading software to the SHMAC core and starting/stopping the synthesized SHMAC core.

To allow for programming the FPGA with new bitfiles, the RealView machine requires either a computer running Windows XP (possibly in a Virtual Machine) to connect to its USB-interface, or a Joint Test Action Group (JTAG)-interface connected to some computer running Xilinx ISE. A machine connected to a JTAG interface and configured for remote access was set up in the hardware lab at IDI, enabling off-site programming of the RealView machine with new bitfiles.

Combining this with the C interface that was written for communicating with the accelerator, allowed for writing tests, and verifying them in the full context of a synthesized SHMAC core quite rapidly. The output from the tests were then printed to the TTY-channel that was configured as the SHMAC-core’s stdout, and could then be verified, either by visual inspection, or by way of comparison with output from a software implementation of the CA.

6.4 Wolfram rules 0-255

The 256 different rules possible in a 1D CA with $r = 1, k = 2$, numbered by the Wolfram number convention are few enough to be exhaustively tested for some input in a suitably short time. As such, a short run of these were performed in ModelSim prior to FPGA synthesis, evolving some time steps of each of the rules, comparing them against the results found in a similarly configured software simulation run.

Additionally, a run of 8192 time steps of each of the 256 elementary CA rules were performed on a synthesized version of the accelerator, verifying the final time step of every one of these by capturing the results from both the software CA implementation and the synthesized CA-accelerator, and doing a *diff* between the two runs.

6.5 Pseudorandom number generation

A more practically useful test, is the application of the CA-accelerator for generating random numbers. A number of approaches have been explored for using CAs to generate random numbers. One of these, the application of rule 30[32], is an example of a uniform CA used for this purpose, but there are also other suggested approaches that make use of a quasi-uniform CA (e.g. the application of rule 90 and rule 150[33]).

Of particular importance to the application of rule 30 to create random numbers, is the configuration of the CA, specifically the amount of cells involved, and the boundary conditions[32]. In an implementation in the accelerator, an infinitely wide CA would be impossible, since the accelerator has a fixed size. As Wolfram states, a CA with wrapped boundary conditions, is equivalent to an infinitely large CA with a periodic initial state, thus a wrapped boundary condition was chosen. Additionally it is necessary to have enough cells in the CA, as with too few cells, the cycles that the rule 30 CA will fall into are quite short (i.e. with 8 cells, the maximum cycle length found by Wolfram was 17). The maximum cycle lengths do however increase quite rapidly (although some specific sizes have shorter maximal cycle lengths than the next lowest size). Wolfram suggests using a configuration with at least 49 cells (implicitly for the purpose of creating 32-bit sequences of pseudorandom numbers), as the cycle lengths should then be in the range of 2^{32}[32, p. 161]. For the test of the accelerator, a configuration consisting of 32 cells in each row, configured to have wrapping boundary conditions was used to create pseudorandom bytes. Such a configuration should have a maximal cycle length of 2002272[32, p. 154], a number that is at least larger than the intended range of numbers that pseudorandom *bytes* would imply.

There are a few possibilities with regard to which cells' states to use as pseudorandom bits, Wolfram suggests either choosing a single cell[32, p. 162], or possibly more than one for efficiency with some caveats regarding correlation, making the relative positioning of these cells important. Hortensius et al. also considered sampling multiple cells, testing specific spacing arrangements, and finding that with 3 cells between each sampled cell, "the rule 30 CA PRNG is performing as well as a standard software PRNG"[33]. For the purpose of these tests, a scheme very similar to the one used in Mathematica[11, p.317] was used; sampling the "central" cell every time step. The configuration chosen for this test deviates in a few details from the one used in Mathematica though, while it does use wrapping boundary conditions, the number of cells in a row is quite low compared to the one used in Mathematica ("a few hundred"[11, p.317]), the initial state is also not randomly selected, opting instead for an initial state where the "central" cell (i.e. the sampled cell), is the only cell with state 1.

To simplify the task of reading out successive pseudorandom numbers, the "central" cell was placed in the easternmost position, as that column of cells is directly connected to the output of the accelerator. Further, a configuration with a height larger than 1 was chosen, were each row below the first one were initialized with the successive time steps of rule 30, as suggested by Toffoli[26, p. 95]. When performed with 8 or more rows, a pseudorandom byte will be available in the output of the accelerator immediately after the necessary amount of time steps have been performed, removing the need for multiple opcodes to get the results. This is possible because the random bits are found in the position where the single cell with initial value 1 was positioned, in the following time

steps. An example of such a configuration is shown in **Fig. 6.1**, where the accelerator's synthesized configuration has a width of 32, a height of 48, an *available neighbourhood* of 9 cells (configured to follow rule 30 in the east-west direction, ignoring the irrelevant neighbours), and a boundary-configuration where the easternmost boundary wraps around to the westernmost and vice versa. Note that the single black value representing the cell with state 1 in the first row is positioned in the easternmost position.

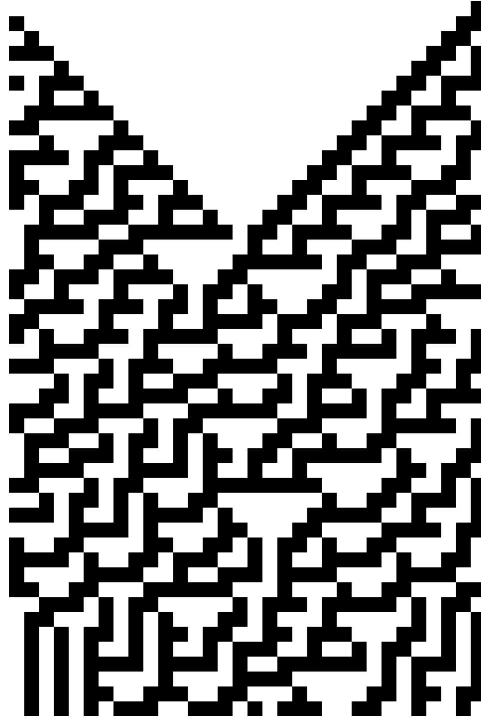


Figure 6.1: The initial configuration for the rule 30 based pseudorandom generation

To allow for testing the evolution of rule 30 in the accelerator as a source for pseudorandom numbers in general, a tiny set of functions were written in C to allow functionality similar to how `rand` works in C. That is, the functions keep a buffer of a certain number of pseudorandom numbers generated by rule 30, and automatically refills that buffer whenever it exhausts its supply of those numbers. To keep things simple, while allowing for the concept of a *seed* value in such a way that somewhat different sequences of pseudorandom numbers can be produced, a simple solution where the *seed* number is used to specify the number of time steps to evolve prior to the first sampling of pseudo random bits. This allowed the solution to keep the initial state as previously specified, a single cell with state 1 placed in the column where the pseudorandom bits will be sampled.

A sample run of this implementation was performed where the initial seed was 0 (i.e. no initial time steps performed), and the resulting numbers were copied into Matlab to create a histogram of the values, which is shown in **Fig. 6.2**. As can be seen from the histogram, all values between 0 and 255 are present, and the distribution looks roughly

uniform.

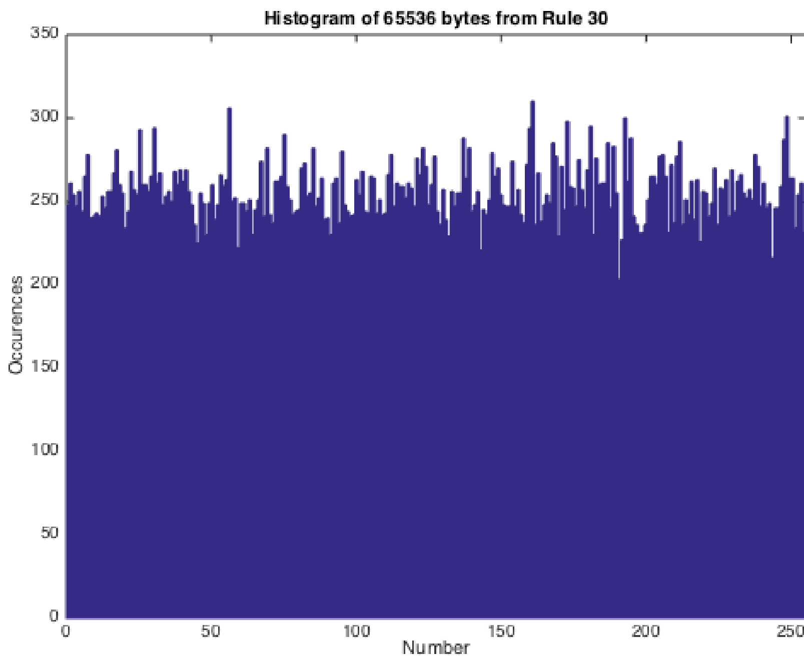


Figure 6.2: The distribution of the first 65536 pseudorandom bytes from rule 30.

6.6 Density Classification

Density classification in this context boils down to figuring out whether a specific bitstring has a majority of 0s or 1s, or a tie between the two. The specific goal is to replace a bitstring by either all 1s if the majority was 1s, or all 0s if the majority were 0s, or otherwise an alternating sequence of 0s and 1s if there is a tie.

This is an interesting problem to solve with CAs, as the cells have to be able to agree about a global issue through only local communication. One proposed way to solve this problem is by way of a pipeline of rules[34], where rule 184 is evolved first for $\frac{L-2}{2}$ time steps, where L is the width of the row of cells, followed by rule 232 for $\frac{L-1}{2}$ time steps.

A test on a CAS 48 cells tall and 32 cells wide was approached as follows: first rule 30 was used to create enough pseudorandom bits to fill all the cells (taking the central column from rule 30, and storing the results from as many time steps as there were cells to fill). These pseudorandom bits were loaded into the cells, and then a configuration where the 1D rules 184/232 were used in a north/south-fashion was applied for the necessary amount of time steps. According to Fukś $\lfloor(height-2)/2\rfloor$ time steps of rule 184 and $\lfloor(height-1)/2\rfloor$ time steps of rule 232 should suffice[34].

During the load of these pseudorandom bits into the cells, the columns were also counted by the C-code to get a classification to compare against. The entire test was thus automated, and designed so that it should be reasonably robust against variations in size of the CAS.

An initial test was performed in a stand-alone fashion where both the application of rule 30 for pseudorandom number generation, as well as the application of rule 184/232 for density classification were performed directly as part of the test. Since the initial state for rule 30 was fixed in this configuration to allow for a consistent test, an alternate approach was taken to testing multiple applications of rule 184/232; The responsibility for randomizing of the initial states to be classified were moved to the rule 30-based *rand*-implementation described in Chapter 6.5, since that implementation maintains the state of its rule 30 evolution between evolutions, and is thus able to create different random numbers each time it is called. The combined test was then applied on a sequence of 1000 randomly created initial states, and verified automatically against the C-reference values. None of these tests failed.

6.7 Conway's Game of Life

Even though the main focus during testing has been on the 1D functionality of the CA-accelerator, the CA-accelerator also has the possibility of functioning in a 2D configuration, with neighbourhood sizes of either 5 or 9. The latter of these allow for running tests of Conway's Game of Life.

Conway's Game of Life, invented by John Horton Conway and originally presented by Gardner [35] is a set of rules where a cell is referred to as *living* if it is 1, and *dead* if it is 0. The rules for Conway's Game of Life are as follows:

- A living cell survives into the next generation if it has exactly 2 or 3 living neighbours
- A dead cell becomes living if it has exactly 3 living neighbours.
- All other cells die (or remain dead).

As a quick test of Game of Life on the accelerator, the simple, short period patterns demonstrated in Gardner's article[35] were evolved for 15 time steps. The results of this evolution is shown top to bottom, left to right in **Fig. 6.3** (purple and orange lines have been added manually to separate the time steps). The top two patterns are supposed to die off after one time step, on the second "line" from the top, the first pattern should also die off in a similar fashion, while the second one transforms into a static *block*. Using Conway's names, the final pattern on the second line is a *blinker*, with a period of 2. The third line demonstrates two patterns that both evolve into a static *beehive*-pattern. On the fourth line, the first pattern also evolves into a *beehive* pattern, one time step after the patterns on the preceding line. Finally, the lower right pattern has a longer period than the others, but eventually, it too settles down into a period of 2.

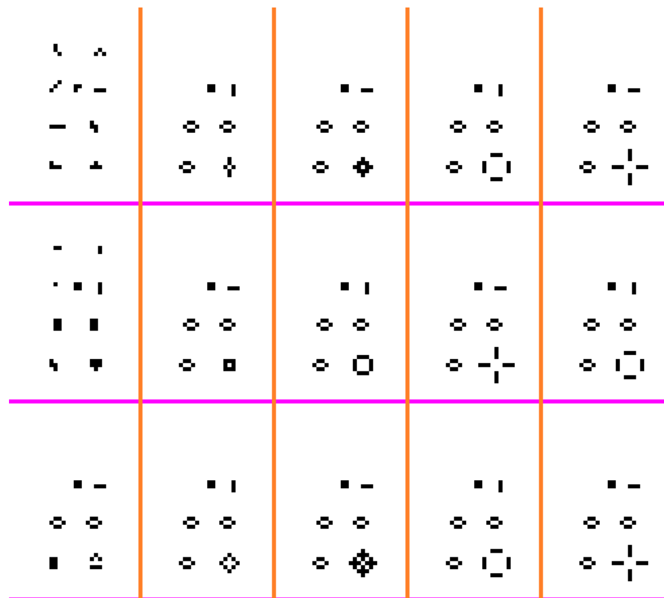


Figure 6.3: 15 time steps of short period Game of Life patterns, evolved on the CA-accelerator

Another interesting pattern in Game of Life, is the *Gosper Glider Gun*, as seen in **Fig. 6.4**, which shows infinite growth (it is actually the very first such pattern to have been found[36]). The gun has infinite growth, as it creates *gliders* while going through its periodic behaviour. A glider is so called because it "glides across the field"[35], that is, it repeats a sequence of shapes, but changes its position while it does so. Although any implementation within the accelerator has to exist within a finite size CAS, the cyclic nature of the glider itself can be verified and tested, similarly the reactions of the gun to the fixed size (whether with wrapped or fixed boundary conditions) can be at least triggered and commented upon.

Figure 6.4: The Gosper Glider Gun.

A rotated version of the Gosper Glider Gun was evolved to the 100th time step on the accelerator, and the results were compared to the same sequence being run in the software CA implementation. The 55 first time steps of this evolution are shown top to bottom, left to right, in **Fig. 6.5**. This sample is large enough to see the Gliders being produced, becoming quite clear in the fourth column from the left (time steps 15-19). The period is also quite visible, the topmost time step in the seventh column from the left shows the

same structure as time step 0 (upper left), except for the glider that has been created, visual inspection of the following time steps show that the gun structure itself continues on with its second period, while the glider leaves to the left. This particular run was performed

Figure 6.5: 55 time steps of the Gosper Glider Gun, time steps are listed top to bottom, left to right.

with fixed boundary conditions, to avoid the glider from wrapping around and coming in contact with the gun structure again. The fixed boundary condition is 0, which does affect the Glider as it is about the leave the system, as can be seen in **Fig. 6.6**. The reason for this specific behaviour at the boundary, is that the glider comes in contact with the "cells" at the boundary, which do not follow the rules of Game of Life. Since there are no cells to react to the glider, it can't move into the boundary, instead it ends up leaving behind a *block*, which is then collided into by the next glider.

If however, wrapping boundary conditions are enabled, no *block* is created, instead, the glider shows up on the opposite side of the CAS, as can be seen in **Fig. 6.7**. The figure skips the time steps that are equal between the wrap and no-wrap configurations (effectively all the states shown in **Fig. 6.5**, since no pattern has reached the boundary yet in the time steps shown there).

Wrapping boundary conditions do not substitute for the infinite size that would be required for infinite growth though, since actual room for patterns is finite infinite growth would suggest an increasing chance of collision. With infinite space, the gun would continue producing gliders, and the gliders would continue in their direction away from the gun. Since the gliders are following the same rule, they move at the same fixed speed, thus they should not be able to collide with each other. However, in a situation with wrapping boundary conditions, the gliders won't be moving away from the gun once they wrap

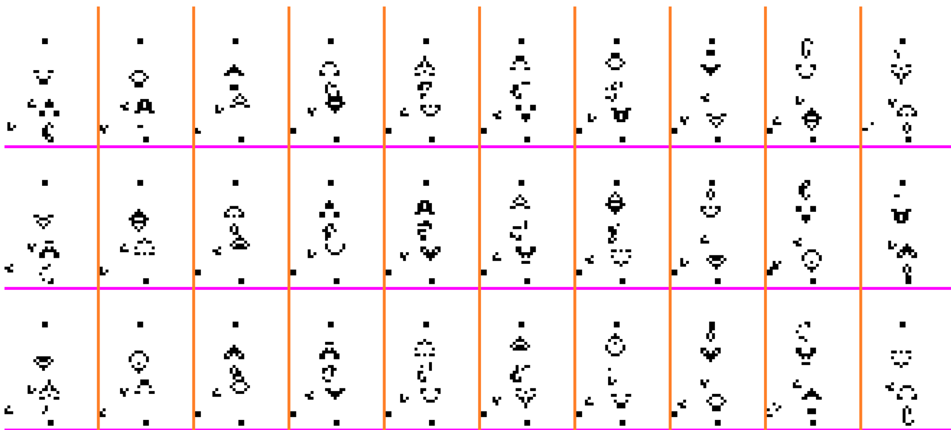


Figure 6.6: Additional 30 time steps of the Gosper Glider Gun, demonstrating the glider stopping at the boundary.

Figure 6.7: The same time steps as in the preceding figure, with wrapping boundary conditions.

around the boundary. At that point, they might be moving towards the gun instead, as can be seen in **Fig. 6.8**, where the following time steps are shown with wrapping boundary conditions enabled. The glider that wrapped around ends up colliding with the gun, breaking the gun's periodic production of gliders.

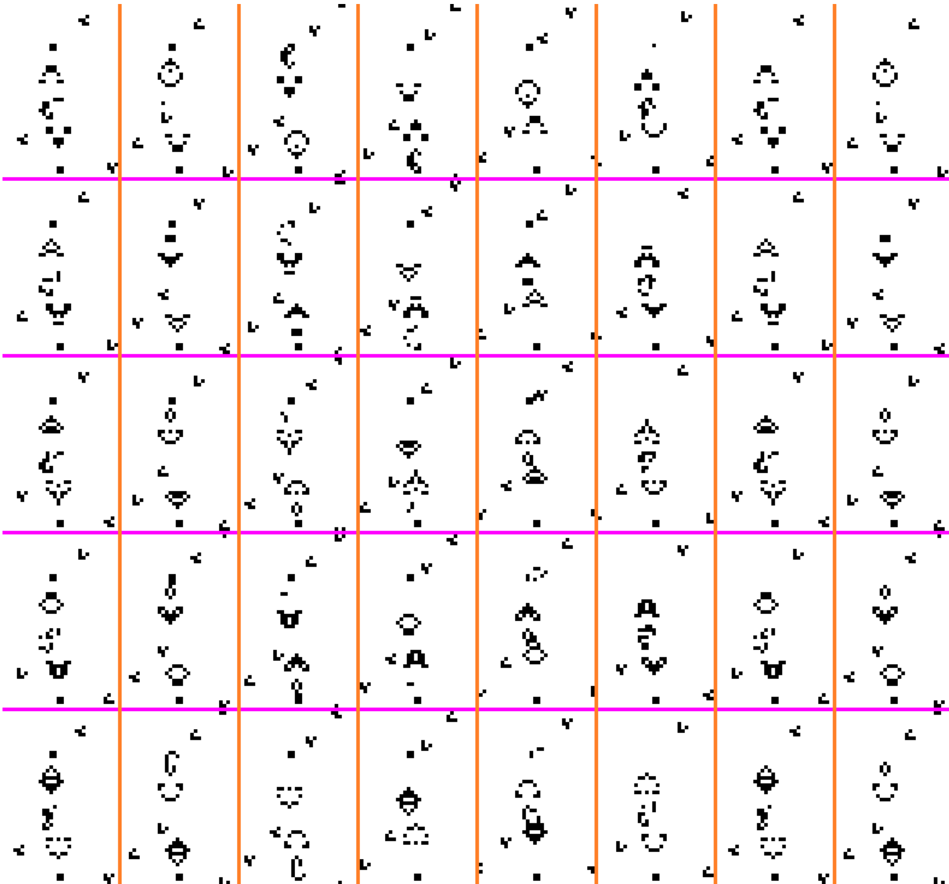


Figure 6.8: Some additional time steps of the Gosper Glider Gun with wrapping boundary conditions, demonstrating the destructive glider.

The major results from testing Game of Life in this fashion were firstly a demonstration of a 2D CA, and at the same time a demonstration and test of the boundary configuration. Both of which seem to behave at least in the same way that the software CA implementation does.

6.8 Genetic Algorithms

Finding solutions to problems using CAs can be a rather complex procedure, and the search space for possible solutions can be rather large. One approach to finding new potential CA

solutions to problems, is to apply a genetic algorithm. Genetic Algorithms (GA), as were covered in Chapter 2.2, apply a population of potential solutions to a problem, then subject them to a a process of selection, crossover and mutation dependant on their relative *fitness* (i.e. their success at solving the problem).

When applying a GA to finding a CA solution in this case, the candidate CA rule is the genome, and the bits that make up that rule are the chromosomes as demonstrated by Mitchell[15, p.48]. Different approaches could also have been taken, i.e. having a fixed rule set in the CA, and letting the initial states, or sequence of inputs be the genome (the latter was the approach taken by de Garis in the CAM-Brain project[6])

The steps of evolving a GA are[15, p.10]:

1. Create an initial population P of potential solutions (genomes) randomly.
2. Apply all of the genomes to the specified problem, and rate their fitness.
3. *Select* pairs of the genomes from P, with a probability relative to their fitness for reproduction into the next generation's population P'.
4. Apply *Crossover* to each pair of selected genomes with some probability.
   Crossover is the operation of selecting a random point in two individuals, and exchanging the chromosomes at that point. This way, if the "parent"-genomes were A and B, then the resulting offspring genomes A' and B' will end up like this if the genome size is n and the crossover position is c:
   A' = Chromosomes 1 to $c - 1$ from A, followed by chromosomes c to n from B
   B' = Chromosomes 1 to $c - 1$ from B, followed by chromosomes c to n from A
   If crossover is not applied, then A' = A, and B' = B.
5. Apply *Mutation* to each individual in the next generation with some probability.
   Mutation is the operation of modifying a random chromosome, since the chromosomes in this case are bits, this means that mutation is a bitflip.
6. Repeat from step 2, replacing P with P', unless a predetermined number of repetitions has been reached.

In the tests run on the accelerator, the source of random numbers used for the probabilistic parts of the GA were supplied by the rule 30-based *rand()*-replacement functionality described in Chapter 6.5.

One of the tests that were performed of genetic algorithms, was to have the percentage of cells with state 1 as fitness function[27, p.13], a problem which has a known (trivial) solution, namely rule 255 (or the equivalent rule for higher neighbourhood sizes). The expected behaviour is thus a rule that comes as close as possible to $f(x) = 1$ as a transition function. Crossover was configured to have a probability of 50%, and mutation was configured to have a 10% probability, additionally 2 randomly created genomes were added into each generation at selection, and the two top genomes were automatically selected at least once. For each generation, five randomly generated initial states were created, and each of the genomes were used as a CA rule with these five initial states, evolving each of the initial states for 1000 time steps. Since each genome was applied to five initial states, the fitness-function results from each of these were added together, giving a maximal possible fitness value of 5. The resulting sequence of average and best fitness values across

the GA-generations is shown in **Fig. 6.9**, where a steady sequence of little to no change is visible for a little over 2100 GA-generations, followed by a steep increase. A close up view of the fitness for the GA-generations between roughly 2200 and 2260 is shown in **Fig. 6.10**. Looking at the latter figure, it becomes clear that steep increase in fitness doesn't jump directly to 5, but goes through a few generations of increase before it reaches five, then oscillates a bit. The average fitness increases shortly thereafter.

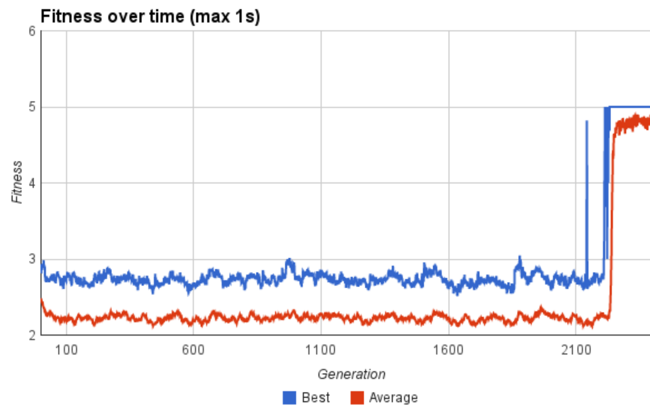


Figure 6.9: Fitness values across a run where the fitness function was defined as the fraction of cells with state 1. Each genome was run 5 times, thus 5.0 is the highest possible fitness value.

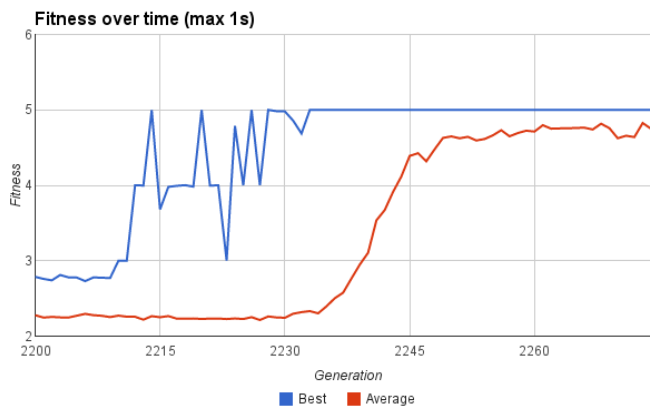


Figure 6.10: Fitness values for the GA-generations where the steep increase in best and average fitness were visible in the previous figure.

6.9 Continued usage test

Knowing whether the design can continue to work as expected over time is not a trivial task, but at least some testing was done in this regard. A GA much like the one described in the previous section, but with a different fitness function, was left running for more than a week of continuous production of new GA-generations. While the GA setup in question did not reach an optimal solution, the output from the continued production of GA-generation tests seemed to be stable, and as far as could be determined by purely looking at the fitness-results, the accelerator was still behaving as it was intended to. The specific task that was attempted to be solved genetically in this run was density classification by way of a single 2D rule, this task is known to not be perfectly solvable by a single rule[37], and as such, the GA-generations were not expected to reach a perfect fitness-result. It did however seem to reach at least a local maxima of some sort, where almost all the genomes had the exact same fitness-value (barring a rare few genomes, and genomes that were intentionally added randomly, a fitness value that was above the fitness-values produced by randomly created genomes added to the same generation.

Chapter 7

Conclusion

The goal of this project was to include a cellular computing inspired accelerator to the SHMAC architecture. The specific approach taken to cellular computing has been the evolution of time steps in cellular automata. An accelerator that is capable of evolving uniform 1D and 2D cellular automata in a fixed size cellular array has been designed and implemented in the HDL-language VHDL. The accelerator has then been integrated into a coprocessor Interface Module, resulting in a new SHMAC processor tile that contains a CA-accelerator. To allow software to make use of this accelerator without having to write specific assembly code, a programming interface for the accelerator, written in C, has been proposed and demonstrated.

A requirement of this project has been to propose and demonstrate a suitable application of the accelerator. Multiple applications have been demonstrated; The generation of pseudorandom numbers, density classification, and a genetic algorithm evolution of CA rules. In the genetic algorithm pseudorandom numbers generated by the accelerator were used for the probabilistic parts, in addition to using the accelerator to evolve the time steps of its population. All of these applications have been demonstrated by writing code that makes use of the C interface to control the accelerator. This code was then executed on a synthesized SHMAC configuration on an FPGA.

The behaviour of the accelerator has also been tested by comparing the CA time steps evolved by the accelerator to those of a software implementation of CA time step evolution. For these tests, a known set of 1D and 2D rules, consisting of the complete set of elementary CA rules and Conway's Game of Life were used. The time steps evolved by the accelerator in these tests have consistently matched those of the software implementation, demonstrating that, at least so far it performs the task it was designed to do correctly.

In short, the integration of an unconventional accelerator in a conventional CPU core on SHMAC has been demonstrated by synthesizing such a core, and writing software that makes use of the accelerator while running on the CPU core.

7.1 Speed of the accelerator

No specific benchmarks comparing the accelerator to a software implementation running on the same CPU have been performed. Any such benchmarks would have to take into account the fact that the speed of the accelerator is dependant on the degree of parallel updates that the CAS is configured to have. A rough estimate can however be performed to argue that the accelerator can be faster: A software implementation will have to perform at least 3 operations to perform a cell update, a read of the current state, a read of the relevant next state from the rule table, and a write of the next state. The accelerator performs the same operations, but can perform them in parallel for multiple cells. A full time step in a fully parallel configuration of the accelerator can be evolved every fourth clock cycle (See Chapter 4.2.4), thus if the software implementation were to be faster than the accelerator, it would have to perform 3 instructions per cell in the CA in less than 4 cycles. Which for 8 cells would be a total of 24 instructions, not counting overhead for branching logic and memory latency, in less than 4 cycles. Given the implications from the ModelSim tests performed in Chapter 6.2, where multiple time steps were evolved in the time span between two program counter values, this is very unlikely to be possible, thus the accelerator should be faster than performing the time step evolution in software.

7.2 Discussion of limitations and trade offs

In this section, some of the design choices that were made will be discussed, particularly the trade offs that have been made. The reason behind these choices will be given, and the limitations that these choices result in will be presented. Some suggestions for workarounds to these limitations will also be given.

7.2.1 Overhead in communication

The accelerator that is presented in this thesis aims to implement the evolution of cellular automata on their own premises. This can be seen in details such as the communication possibilities between the synthesized cell and the rest of the CPU. Non-local communication between cells is avoided and the only possible way to load and/or store state is by reading the state of the eastern edge, or writing input on the western edge. This allows for an implementation where each cell is capable of evolving new time steps quite rapidly, but it also means that the amount of accelerator opcodes necessary to read or write the entire state grows with the number of cells in each row of the accelerator. The worst case for overhead is a 1D single row configuration. In this case reading the entire state would require as many operations as there are cells in the accelerator. That particular case can however be reduced somewhat by configuring the 1D accelerator to have one column instead, in which case the overhead would fall drastically, as all the cell states would reside on the eastern edge, and thus be available for reading easily in the coprocessor registers. The latter solution isn't completely free from limitations and overhead though, but it gets quite close as there is a finite amount of coprocessor registers available[30] (see Chapter 8.1), if all of those are used then 480 bits of state can be read before it becomes necessary to shift the cells in the accelerator.

7.2.2 Fixed direction of input/output

The accelerator has a fixed size, and only one input/output direction; west-east. The lack of input/output options in the north/south-direction does make it slightly inconvenient to read rows from the accelerator, a problem which at least to some degree can be reduced by rotating the rules and initial state by 90 degrees, turning rows into columns and vice versa (as for instance was done with the density classification tests in this thesis). If a use case demands reading columns and rows at different times, such a rotation might not reduce the amount of overhead involved in reading the states, particularly if the fraction of row reads is close to the fraction of column reads so that the biggest of the two can't be reduced by rotation. The fixed size does allow for representing all the cells in hardware, and thus opens the potential for evolving time steps as fast as the rule table can respond, but it also limits the potential applications somewhat. As was seen with Conway's Game of Life in Chapter 6.7, the finite size does pose limitations, and unlike implementations that use RAM external to the CA to store the cell states, the accelerator can't change its size dynamically (i.e. trading space that would be used for rows to get space for additional columns in a sparse fashion).

7.2.3 Complex system for rule description

The implementation has a somewhat complicated system for handling rules, where the byte writes to the rule tables were added to reduce the overhead involved with configuring the rule tables. The reduced overhead is paid for with the addition of additional connections between the rule tables and the toplevel, which might complicate the design unneccessarily. The gain of the byte writes can be seen in that without them, the worst case scenario of writing a complete rule would involve writing 512 bits (or upto 256 bits if only writing the bits that are different from the bits written by the reset-opcode), the byte writing reduces this to 64 byte writes. While this greatly simplifies the amount of code neccessary to write a rule, the byte writes might not be necessary if something like the optimization suggested in Chapter 8.6 is implemented as a replacement.

Another issue with the rule tables is that the current approach to allowing some non-uniformity introduces a fair amount of opcode overhead as well, in that write masks have to be turned off and on to write rules to specific cell grids. This is a result of the lack of addressability of the rule tables, which is in turn a result of the overall design goal having been the implementation of *uniform* CA.

7.2.4 Size limitations

The current design has some limitations with regards to how many cells can be synthesized. Firstly, the coprocessor IFM, as used by the accelerator, has only 480 bits available for operands, which means that a maximum of 480 rows can have their state written, the limitation is slightly better on the output end, where 512 bits are available. The choice of the coprocessor interface was partly made to allow for fast regular interaction with the accelerator, since the way state is communicated to/from the accelerator involves a fair amount of interaction. The potentially synthesized size is also limited by the relative

amount of resources used by the rule tables, which can get fairly high when the neighbourhood size increases. Currently, each row has at least one cell grid, and no cell grid can belong to multiple rows, a choice that was made to allow for a simple consistent communication scheme between rows. This does have the side effect that within a fixed area usage, the potential size in terms of number of rows is lower than the potential size in terms of columns, as an increase in the number of rows will always imply an increase in the number of rule tables. Some approaches to reducing these issues are discussed in Chapter 8.1 and 8.4.

Chapter 8

Future Work

8.1 Other configurations

The current implementation of the accelerator is tailored towards use as a coprocessor, however the IFM interface allows for other configurations, such as the Wishbone-master- or Wishbone-slave-IFMs[30], interfaces which have not been approached in this thesis, as the coprocessor interface allowed for communicating with the accelerator rapidly with little added overhead. Since the operations involved in reading and writing state to the accelerator currently require quite a few operations, this was a useful property. One downside to the coprocessor IFM though, is that it doesn't use interrupts for signalling when the accelerator is done, meaning that the accelerator has to be polled to figure out when the next operation can be started. The Wishbone-IFMs use interrupts for this purpose, and might therefore be worth some further investigation. A downside to the Wishbone-IFMs, is that they use DMA for communication, which introduces some additional overhead in the communication, something which is suboptimal as the current design has a fair share of operations involved in the tasks of reading and writing state and rules. However, if the use case is running the same rules for prolonged periods of time (i.e. large numbers of time steps), and only rarely needing to read the state of some subset of the CAS, then they might still be a useful option to the coprocessor interface. Another limitation that the coprocessor interface has, that can be reduced by using the Wishbone-IFMs, is the number of bits of I/O-available, where the coprocessor allows for 16 32-bit I/O fields for communication with the accelerator (of which only 15 are available for input, as one of them is reserved for the *options*-field), the Wishbone IFMs allow for 512 fields of 32-bit I/O (with a similar use of one input field for the *options*-field). The current coprocessor interface thus limits the size of the CA states that can be loaded into the accelerator to $15 * 32 = 480$ rows, while the Wishbone IFMs would allow for $511 * 32 = 16352$ rows. Another potential idea is to use the extra I/O-fields to allow the accelerator to read/write multiple columns, letting the toplevel handle the details of shifting the CA cells instead of having to manually execute the shift-operations.

An even more different configuration from the coprocessor IFM would be an imple-

mentation where the CA-accelerator is is a SHMAC-tile on its own. An IFM that does this was suggested in Teilgård's master thesis, but has not yet been implemented[30, p.30-31]. This would allow for a CA-accelerator that is somewhat closer to the one suggested by Rusten et al[23]; With such a configuration, there would be further potential for sharing the CA results between processor tiles, as well as perhaps connecting together multiple such tiles to create larger CA with quasi-uniform behaviour (analogous to how CAM6 allowed for connecting/disconnecting parts of the CA from the rest, potentially running different rules on the various parts, and connecting them on demand). This would allow for some more flexibility, at the cost of more expensive communication with the CA, as communication would have to move from the coprocessor instructions to DMA-communication.

8.2 Variations on the wrapping configuration

Adjusting the possible configurations of the boundary wrapping so that i.e. spiral wrapping[26, p. 95] is achievable would allow for more flexibility in using 2D CA for evolving new time steps of 1D CA, by allowing the 2D CA to treat its multiple rows (or potentially columns) as extensions of the first row (or column).

8.3 Scheduling issues

The accelerator considered in this thesis has quite a bit of context data, in that it contains the state of all the cells, including the rules connected to these cells. When an operating system's scheduler wants to perform a context switch[38, p.321-324] it needs to be able to both save the context of the currently running process, as well as to load the context of the process it is switching to. The addition of the accelerator means that a decision needs to be made as to whether its state also should be saved/loaded as part of the process context (in addition to the usual CPU registers).

Adding the accelerator as part of the process context, would imply reading out all the data from the accelerator, and writing the relevant state from the next process context to it afterwards, which would (depending on the width of the CAS) possibly involve a fair amount of additional read/write operations, many of which may be unnecessary if only some of the processes use the accelerator.

Another approach would be to handle the contents and state of the accelerator at the OS kernel level, allowing access only through a well defined interface. This would mean that the accelerator would only be directly controlled by code executing at a higher privilege level than user space code. In this case, the accelerator might not need to save/load its state on all context switches, as the kernel can either keep track of which processes use the accelerator, and thus only save/load its state when a process that uses it is switched to, or it can prevent access to the accelerator by more than one process simultaneously (other processes would then have to wait, or queue for access to the accelerator, potentially opting for solving their problem by other means).

One particular part of the design presented herein complicates moving the state of the CA-accelerator in and out of a tile, namely the fact that rules and rule masks are write-only, that is, there is no operation defined that makes it possible to read the active rule(s). This

can to some degree be mediated by carefully designing the software-side interface that is used to communicate with the accelerator so that it keeps track of all the rules that are in effect, and thus is able to repeat the necessary setup-operations to reproduce the state of the accelerator.

Some processes might also want to perform operations that would fit well in the accelerator, even if no processor tiles with an accelerator is available at the moment, perhaps by performing their CA-tasks in software for the time being. Such processes might still benefit from access to an accelerator if one becomes available. Which means that it might be necessary to consider how to move CA-tasks between a software implementation on one tile, and onto a tile with the CA-coprocessor available, as well as what heuristics should be used to decide when and if such a transition should happen. Similarly, if multiple tiles with CA-accelerators are available, the operating system has to decide which of these are used by which process (perhaps trying to avoid running the same accelerated task on different tiles, to reduce the amount of state saving/loading).

8.4 Size optimizations

The limiting factor for the maximal size of the CAS that the accelerator can be synthesized with is the area usage of the accelerator. One of the components that is likely to use quite a few FPGA-resources is the rule table, especially when synthesized for a neighbourhood sizes of 9, since that means that every single rule table needs to store 512 bits of information. While the amount of rule tables is adjustable in each row by making more cells part of the same cell grid, this is currently not a possible solution in the columns. The reason behind this, is that each cell grid is defined to be exactly one row tall. Thus it is not currently possible to sacrifice parallel updates to get more *rows* of cells synthesized. Introducing the potential for adjusting the height of the cell grids would be a potential solution to this problem, however this would also require redesigning the I/O-connections between the cell grids, as well as the I/O connections used by the CA-toplevel to read and write at the boundaries. Considering other ways to increase the amount of cells that the accelerator can be synthesized with would also be interesting.

8.5 Different I/O approaches

The current CA-toplevel has a one to one mapping between bits in its opcode arguments and the cells that it writes state to, which means that the number of cells it can write to is limited by the IFM. This limitation can be reduced in a variety of ways; one possible solution would be to add an opcode to set a *window* into the CAS, deciding which rows that are currently affected by writes and output for reading. Other solutions would be to keep the fixed limits of which cells are directly available, but add a downwards shift to the CAS, so that the cells that are ”out of range” can be shifted into range. This solution would require some careful design to allow for loading, as shifts eastwards would still shift the entire CAS eastwards, potentially destroying the states that were shifted ”out of range”, but if i.e. boundary wrapping can be selectively applied, this is solvable.

8.6 Rule table optimizations

The overhead involved in storing/loading rules, particularly when switching back and forth between rules could possibly be reduced by introducing a few *rule buffers*, where rules could be stored, and then applied to the rule tables on demand, allowing for switching rules with perhaps as little as 1 accelerator opcode once the rule has been stored to the buffer. This would also simplify the overhead of work loads such as the 184/232 rule pipeline a bit. Another potential improvement would be to remove the byte-writes to the rule tables, and instead handle the byte-writing at the accelerator toplevel by reducing the byte write to a sequence of 8 1-bit writes, simplifiying the number of connections that the rule tables need to have to the CA-toplevel. It is also possible to consider modifying the current ”write to all”-paradigm into a system where the rule tables follow a layout similar to the one used for the cell-communication. Instead of writing to all the rule tables at once, the rule tables at the boundary could be writeable by the toplevel, and then a shifting system, like the one used for loading columns could be used to propagate these rules to the ”inner” rule tables. This would increase the number of components inside the accelerator that were communicating in a purely local fashion. Combining all of these suggestions would also be possible, by for instance making all rule writes go through rule buffers, which would perhaps allow for writing to them 32-bit (or more) at a time, the internal handling of the writing from rule buffer to rule table could then be a sequence of 1-bit writes, simplifying the rule tables, while improving the interface at the opcode-level.

8.7 Power optimizations

The accelerator currently propagates the clock to its components, whether it is actively running or not, to allow for a reduction in power consumption by the accelerator, it might be worthwhile to introduce logic that disables the clock propagation to the accelerators internal components when the accelerator is not evolving time steps.

8.8 Power usage considerations

The accelerator consists of a number of cells that may change their state every fourth clock cycle (if the accelerator is synthesized in a fully parallel configuration). This could make it a suitable target for considering power and heat consumption related issues with SHMAC as a whole, by configuring the accelerator to follow a rule where all the cells change their state every time step. Such a configuration might consume a larger amount of power than for instance average CPU operation, which would allow for considering how this affects the other parts of the SHMAC. For instance; should tasks that are run on tiles spatially close to the tile running the accelerator be moved or stopped to avoid further heat/power problems? Should the operating system selectively halt the accelerator at some point for the benefit of the other tiles?

Bibliography

[1] 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.

[2] (2015, 03). [Online]. Available: http://www.ntnu.edu/ime/eecs/shmac

[3] J. von Neumann, "First draft of a report on the edvac," in *The Origins of Digital Computers, Selected Papers*, B. Randell, Ed. Springer-Verlag, 1973, ch. 8.1, pp. 355–364.

[4] (2015, 03). [Online]. Available: http://booksite.elsevier.com/9780123838728/historical/appendix_l.pdf

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

[6] H. de Garis, ""cam-brain" atr's billion neuron artificial brain project a three year progress report," in *Fuzzy Logic, Neural Networks, and Evolutionary Computation*. Springer, 1996, pp. 215–243.

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

[8] D. Adams, *The Salmon of Doubt*. Macmillan, 2002.

[9] M. Mitchell, *Complexity: A guided tour*. Oxford University Press, 2009.

[10] S. Wolfram, "Statistical mechanics of cellular automata," *Reviews of modern physics*, vol. 55, no. 3, p. 601, 1983.

[11] ——, *A new kind of science*. Wolfram Media, inc., 2002.

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

[13] M. Cook, "Universality in elementary cellular automata," *Complex Systems*, vol. 15, no. 1, pp. 1–40, 2004.

[14] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* The University of Michigan Press, 1975.

[15] M. Mitchell, *An Introduction to Genetic Algorithms*. The MIT Press, 1996.

[16] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[17] R. Rojas, "Conditional branching is not necessary for universal computation in von neumann computers," *Journal of Universal Computer Science*, vol. 2, no. 11, pp. 756–768, 1996.

[18] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, The Hardware/Software Interface, 4th Edition*. Elsevier, 2009.

[19] J. L. Henessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach*, 5th ed. Elsevier, 2012.

[20] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[21] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 81–92.

[22] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[23] L. T. Rusten and G. I. Sortland, "Implementing a heterogeneous multi-core prototype in an fpga," Master's thesis, Norwegian University of Science and Technology, 2012.

[24] A. T. Akre and S. Bøe, "Turbo amber: A high-performance processor core for shmac," Master's thesis, Norwegian University of Science and Technology, 2014.

[25] E. F. Codd, *Cellular Automata*. Academic Press, 1968.

[26] T. Toffoli and N. Margolus, *Cellular automata machines: a new environment for modeling*. MIT press, 1987.

[27] M. Sipper, *Evolution of parallel cellular machines*. Springer Heidelberg, 1997.

[28] J. von Neumann and A. W. Walter, *Theory of self-reproducing automata. Edited and completed by Arthur W. Burks*. University of Illinois Press, 1966.

[29] T. Kobori, T. Maruyama, and T. Hoshino, "A cellular automata system with fpga," in *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. IEEE, 2001, pp. 120–129.

[30] M. L. Teilgård, "Integration of hardware accelerators on the shmac platform," Master's thesis, Norwegian University of Science and Technology, 2014.

[31] *RealView Platform Baseboard for ARM11 MPCore User Guide*. ARM Limited, 2007.

[32] S. Wolfram, "Random sequence generation by cellular automata," *Advances in applied mathematics*, vol. 7, no. 2, pp. 123–169, 1986.

[33] P. D. Hortensius, R. D. Mcleod, W. Pries, D. M. Miller, and H. C. Card, "Cellular automata-based pseudorandom number generators for built-in self-test," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 8, no. 8, pp. 842–859, 1989.

[34] H. Fukś, "Solution of the density classification problem with two cellular automata rules," *Physical Review E*, vol. 55, no. 3, p. R2081, 1997.

[35] M. Gardner, "Mathematical games: The fantastic combinations of john conway's new solitaire game "life"," *Scientific American*, vol. 223, no. 4, pp. 120–123, 1970.

[36] M. Mitchell *et al.*, "Computation in cellular automata: A selected review," *Nonstandard Computation*, pp. 95–140, 1996.

[37] M. Land and R. K. Belew, "No perfect two-state cellular automata for density classification exists," *Physical review letters*, vol. 74, no. 25, p. 5148, 1995.

[38] M. Wolf, *Computers as components principles of embedded computing system design*, 3rd ed. Elsevier, 2012.