NTNU – Trondheim
Norwegian University of
Science and Technology

2D and 3D On-Chip Development of Cellular Automata Machines

Ola Martin Tiseth Støvneng

Master of Science in Informatics
Submission date: May 2014
Supervisor: Gunnar Tufte, IDI

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

Abstract

Cellular automata (CAs) are massively parallel machines where many simple cells work together to solve a larger problem. Each cell is very simple and only interacts with its neighbouring cells. Evolutionary algorithms (EAs) are often used to evolve them because they are very difficult to program by conventional methods.

In evolvable hardware (EHW), EAs are used to evolve computer hardware designs. Field programmable gate arrays (FPGAs) are often used as a platform for implementing EHW because of their massive reconfigurability. Previous research at IDI NTNU have resulted in the construction of a virtual FPGA, better suited for EHW, implemented on an actual FPGA. The virtual FPGA is called an sblock matrix and is designed as a reconfigurable non-uniform two dimensional (2D) CA. Each cell in the CA is called an sblock. Also implemented on the physical FPGA is a developmental system based on cellular development. This system is used to develop the sblock matrix based on a set of development rules that activate new sblocks or changes the sblocks' behaviour based on neighbouring sblocks' attributes. The implementation also supports on chip fitness evaluation, which evaluates how well the current CA is performing its intended task. The fitness evaluation is based on some transform of the CA's output data. EAs need such a fitness value for their operations, so this is useful when evolving the CA.

This thesis' focus is on improving and extending the system for implementation on a newer and bigger FPGA chip with more features. There are three focus areas explored in the thesis, improving the performance of the system, extending the sblock matrix to be three dimensional (3D), and implementing a new transform for interpreting output data from the sblock matrix as part of the fitness evaluation.

The results show that the new FPGA allows for making many parts of the system at least 4 times faster, depending on the size of the sblock matrix. In the case of running the sblock matrix with fitness evaluation, the performance has been increased by at least an order of magnitude.

The 3D sblock matrix is much more expensive to implement than the 2D one, so for implementing larger ones, the performance is scaled back down to that of the original design.

The new transform of the CA's output implemented is the discrete Fourier transform (DFT), which require a lot of multiplication to be done. Implementing multiplicators on an FPGA is traditionally very expensive, however some newer FPGAs have dedicated multiplicators that can be taken advantage of. Implementing a highly parallel DFT using many such multiplicators lets the DFT be as fast as most other parts of the design without using too many additional resources.

Preface

This master's thesis is conducted at the Norwegian University of Science and Technology (NTNU), at the Faculty of Information Technology, Mathematics and Electrical Engineering (IME), at the Department of Computer and Information Science (IDI), with the The Computer Architecture and Design Group (CARD).

The thesis concludes my two year Master of Science degree in Informatics. It counts for 60 credits, being the sole work load for the last year of the degree ending in May 2014.

I would like to thank my supervisor Gunnar Tufte, who has been of tremendous help throughout my work on this thesis.

Ola Martin Tiseth Støvneng
Trondheim, May 30, 2014

Contents

1 Introduction **1**

2 Background Theory **3**
2.1 Cellular Automata 3
2.1.1 Grid Layout 3
2.1.2 Neighbourhoods 3
2.1.3 State Change 4
2.2 Genetic Algorithms 6
2.3 Development 8
2.4 Field Programmable Gate Array 9
2.5 Evolvable Hardware 10

3 Previous Work **11**
3.1 An Evolvable Hardware FPGA and Development 11
3.2 Sblock Matrix and Development Process in Hardware 12
3.2.1 Functionality 12
3.2.2 Architecture 13
3.2.3 Implementation 15
3.3 Fitness Function in Hardware 19
3.3.1 Run Step Function 19
3.3.2 Fitness Function 20

4 Optimising for New Hardware **23**
4.1 The New Hardware 23
4.2 Porting the Design 24
4.3 Parameterisation of the Design 25
4.4 Design Improvements 25
4.4.1 Block RAM 26
4.4.2 Development 27
4.4.3 Config 28
4.4.4 Run Step Function 29
4.4.5 New Instructions 29

5 Three Dimensional SBlock Matrix **31**
5.1 The SBlock Matrix . . . 32
5.2 BRAM and Development . . . 32
5.3 Config . . . 33
5.4 Run Step Function . . . 33
5.5 3D Instructions . . . 34

6 Cellular Automaton Output **35**
6.1 Discrete Fourier Transform . . . 35
6.2 Implementation . . . 37

7 Testing and Results **39**
7.1 General Correctness Testing . . . 39
7.2 Improvements . . . 40
7.3 DFT . . . 42
7.4 3D SBM . . . 44

8 Discussion **47**
8.1 Implementation . . . 48
8.2 Future Work . . . 48

Bibliography **51**

A Instruction Manual **53**

B Rule Format **61**

C Attached Files **63**
C.1 File Hierarchy . . . 63
C.2 Hardware Design Files . . . 64
C.3 Test Benches . . . 66
C.4 Synthesis Results . . . 66
C.5 Software . . . 67

D Software **69**

List of Figures

2.1 A 1D, 2D, and 3D grid 4
2.2 Three different 2D CA neighbourhoods. The C indicates what cell's neighbourhood it is, and the grey cells mark the neighbourhood in the grid. 4
2.3 Neighbourhood at the edge of a 2D cyclic CA 5
2.4 A seven cell neighbourhood for a 3D CA 5
2.5 Example run of a 2D uniform cyclic CA following the parity rule. Black cells indicate live cells. 7
2.6 General FPGA architecture. CLBs and slices are shown in more detail. 10

3.1 An sblock in an SBM. It consists of a configurable LUT and an output register, or FF. 12
3.2 Abstract overview of Djupdal's system 13
3.3 Architecture for Djupdal's design 14
3.4 Data-flow throughout the configuration of the SBM 16
3.5 Development unit architecture 17
3.6 Development: cells' neighbourhood vs BRAM layout 17
3.7 writeType instruction from [3] 18
3.8 Aamodt's additions to the design 20

4.1 Optimised parts in the new design, highlighted by thick lines and grey boxes. 26
4.2 BRAM layout for the new and old design. The grids represent SBM, and each cell represents a word of sblock data. The numbers indicate in what BRAM module the sblocks' data is stored. 27
4.3 Development: cells and their neighbourhoods vs BRAM layout in the new and the old design 28
4.4 Adder tree with clocked registers. It takes 1024 1-bit input signals and outputs the sum of them as an 11-bit number. 29

5.1 Parts changed for the 3D extension, highlighted in grey and thick lines. 31
5.2 3D SBM BRAM layout 32

5.3 Comparison of the writeType instruction from [1, 3] (top) and this thesis (bottom) . . . 34

6.1 Average live cells per step for a specific CA . . . 36
6.2 Average DFT of live cells per step for the same CA . . . 36
6.3 The DFT in the architecture . . . 37
6.4 Simplified DSP slice . . . 38

7.1 Functional test program . . . 40
7.2 A typical program . . . 41
7.3 FPGA LUT usage given array size for original and new design . . . 42
7.4 FPGA LUT usage given array size for design with and without DFT 43

List of Tables

2.1 A simple LUT for a 1D CA. W, C, and E, the states of the neighbouring cells, is the condition or index. Result is the output, the new state. . 5
2.2 A few different 1D CAs following different rules, and whether or not they are uniform. 5
2.3 Examples of 1D development rules. The three middle fields are conditions. The * means "do not care". 9
2.4 Development steps for a cyclic 1D CA following the development rules from table 2.3. 9
2.5 Type to LUT conversion table. Converts developed types to LUTs used in a CA. 9

4.1 Resource comparison of the old and the new FPGAs 24
4.2 Comparison of synthesis results for the original design on the VirtexE and the Spartan6 25

6.1 Twiddle BRAM content 38

7.1 Performance comparison for original and new design 40
7.2 Comparison of synthesis results for original and new design 41
7.3 Synthesis results with DFTs using different amounts of DSPs 43
7.4 Performances of the units of the 3D design, along with a comparison to the 2D design performance. 44
7.5 Synthesis results for the 3D design 45

Abbreviations

1D one dimensional

2D two dimensional

3D three dimensional

BRAM block RAM

CA cellular automaton

CLB configurable logic block

DFT discrete Fourier transform

DSP digital signal processing

EA evolutionary algorithm

EHW evolvable hardware

FF flip-flop

FPGA field-programmable gate array

GA genetic algorithm

I/O input/output

IOB input/output block

LC logic cell

LED light-emitting diode

LSS load, send, and store

LUT lookup table

PCB printed circuit board

PCI peripheral component interconnect

PCIe PCI express

RAM random access memory

RSF run step function

SBM sblock matrix

SRL shift register LUT

TBUF tri-state buffer

VHDL VHSIC hardware description language

Chapter 1

Introduction

A cellular automaton (CA) is a massively parallel machine made up of many simple cells in a grid, where each cell only interacts with its close neighbours. The qualities of such machines are similar to what we find in most cellular structures in nature [11], the merits of which are that many simple cells work together to perform tasks far beyond the capabilities of a single cell. Programming such a machine, hoping to make it perform specific tasks, using conventional methods is very difficult. So we turn to evolutionary algorithms (EAs) inspired by evolution in nature, to try to evolve these machines in much the same way that nature has evolved cellular structures with great success before [11].

In the same way that nature uses DNA as a set of rules to follow when making cellular structures, we can follow a set of evolved development rules for growing new cells or changing the function of existing cells, thereby building the CA.

Another difficulty with using a CA to solve problems is knowing how to interpret the output produced by it. Each cell in a CA has its own output value. One could use the values of some of the cells after some time as output, one could use some aspect of how the cells' outputs change over time, or something completely different. Output interpretation is very dependent on what task the CA is supposed to perform and how the CA is initialised with input.

When running a simulation of a CA on a standard processor we lose some of the merits of massive parallelity and simple cells. Implementing them on configurable hardware like field-programmable gate arrays (FPGAs) we can keep the massive parallelity and take advantage of the simplicity of the cells and the interconnection between them.

Evolvable CAs implemented in hardware is a specialisation of the more general term evolvable hardware (EHW), which is where computer hardware design meets bio-inspired artificial intelligence. Concepts from evolution are applied using EAs to *evolve* new hardware designs. Such evolution of hardware can lead to new and interesting design solutions that conventional methods would have a hard time finding. [9]

FPGAs have many qualities that are useful for EHW, like massive configurability. However, other qualities, like the typically long configuration times, are not desirable

[10]. For this reason a new type of FPGA specifically designed with EHW, and more specifically CAs, in mind has been proposed by Haddow and Tufte [7]. It consists of a two dimensional (2D) grid of simple configurable blocks, *sblocks*, with a very simple interconnect. Production of such FPGAs is probably not very feasible, as the target market is not large enough to justify the potential production costs. However, implementing it as a virtual FPGA on an actual FPGA lets us keep the massive parallelity wanted. This is what has been done by Djupdal [3], and later extended by Aamodt [1].

The goal of this thesis is to further improve and extend the virtual FPGA design in the anticipation of a newer and bigger FPGA. There are three main tasks covered in this thesis, general optimisations to make the design faster on the new hardware, extending the virtual FPGA to be a three dimensional (3D) grid of sblocks, and looking at new types of output from the CA.

The thesis structure is as follows:

- Chapter 2 introduces background theory necessary to understand the rest of this thesis.
- Chapter 3 introduces the previous work on EHW that this thesis is a continuation of.
- Chapters 4, 5, and 6 are the main chapters, each of which addresses one of the three main tasks covered in the thesis.
- Chapter 7 describes the testing methodology as well as the results of the work done in this thesis.
- Chapter 8 discusses the results and the hardware design work done in this thesis as well as possible future work.

Chapter 2

Background Theory

This chapter introduces theory related to CAs, EAs, development, FPGAs, and EHW. This is all background theory that should help the reader understand the main contents of this thesis.

2.1 Cellular Automata

A CA is a regular grid of cells, where each cell has a state which changes depending on the states of its neighbouring cells [12]. By varying the layout and size of the grid, the neighbourhood of the cells, and the rules used to change the state, many different types of CAs can be obtained. This section will take a short look at each variable.

2.1.1 Grid Layout

The grid of cells in a CA can be laid out in many ways. One important feature of such a layout is its number of dimensions, one or two are the most common, but three or more are also possible. Figure 2.1 shows the simplest types of layouts for one dimensional (1D), 2D, and 3D grids. 2D grids fit the internal architecture of most FPGAs very well, as most of them consist of a 2D grid of configurable logic blocks (CLBs), as described in section 2.4.

The 1D grid is simply a line, the 2D grid is in this case a rectangular grid, while the 3D one is a cubic grid. Other layouts can also be used, for example triangular or hexagonal grids for a 2D CA.

This thesis will concentrate on rectangular 2D and cubic 3D grids of various sizes.

2.1.2 Neighbourhoods

The neighbourhood of a cell in a CA defines which cells' states are to be evaluated when deciding the next state of the given cell. The neighbourhoods for the

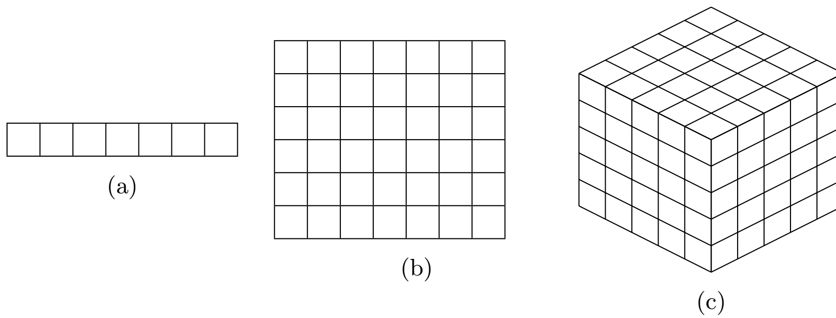

Figure 2.1: A 1D, 2D, and 3D grid

different cells are usually defined in the same way, by the relative position of a cell's neighbours to the cell's position in the grid. Neighbourhoods are normally defined in the straightforward way of having the cells closest to a given cell in its neighbourhood. The cell in question may or may not be a part of its own neighbourhood. A 2D CA may have its neighbourhoods defined in many different ways, some examples are shown in figure 2.2. The grey cells are the neighbourhood of cell C. In the neighbourhood in figure 2.2a, the neighbours are given the names *North* (N), *East* (E), *South* (S), *West* (W), and *Centre* (C).

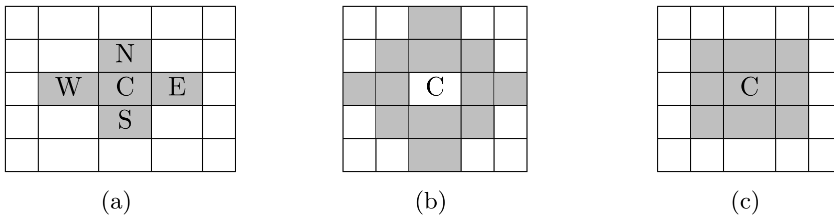

Figure 2.2: Three different 2D CA neighbourhoods. The C indicates what cell's neighbourhood it is, and the grey cells mark the neighbourhood in the grid.

A CA is often defined as cyclic, meaning that the edges are connected to the edges at the opposite side. This results in neighbourhoods as seen in figure 2.3, which shows the same neighbourhood as in figure 2.2a, but at the edge of a cyclic grid.

Figure 2.4 illustrates a neighbourhood for a 3D CA. This neighbourhood simply extends the 2D neighbourhood from figure 2.3 by adding an *Up* (U) and a *Down* (D) cell, extending into 3D space.

2.1.3 State Change

Each cell in a CA has a state, which has a number of possible values. These values are typically the same for all the cells in a CA. The most basic CAs operate with just two different values, *on* and *off*, which can simply be represented by '1' and '0'.

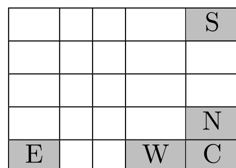

Figure 2.3: Neighbourhood at the edge of a 2D cyclic CA

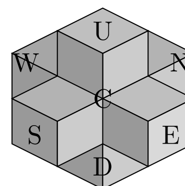

Figure 2.4: A seven cell neighbourhood for a 3D CA

CAs are usually run in steps, each of which are equal. At every step the states of all the cells are updated at the same time. To determine the next state, the states of the cells in the neighbourhood are evaluated using a function of some kind.

One way this function can be defined is by using a lookup table (LUT) with entries for all possible permutations of the states of the neighbours. Table 2.1 is such a LUT for a simple 1D neighbourhood. This LUT can also be described by one simple rule: *If exactly two neighbours of a cell are on, that cell is on in the next step, otherwise it is off.* In this case the LUT may seem cumbersome compared to the simple rule, however, it is very useful because it can describe any state change function. The index is simply a concatenation of the outputs of the three neighbours. (off on on → 011 → 3)

index	**W**	**C**	**E**	**Result**
0	off	off	off	off
1	off	off	on	off
2	off	on	off	off
3	off	on	on	on
4	on	off	off	off
5	on	off	on	on
6	on	on	off	on
7	on	on	on	off

Table 2.1: A simple LUT for a 1D CA. W, C, and E, the states of the neighbouring cells, is the condition or index. Result is the output, the new state.

Cell#	***1***	***2***	***3***	***4***	***5***	
CA#			**Rule**			**Uniformity**
1	30	30	30	30	30	uniform
2	110	110	110	110	110	uniform
3	30	35	100	35	30	non-uniform
4	1	2	3	4	5	non-uniform

Table 2.2: A few different 1D CAs following different rules, and whether or not they are uniform.

The LUT's size grows exponentially with the number of cells in a neighbourhood and the number of different states. Given two different states, the neighbourhood from figure 2.2a would require a LUT of size 32, while the neighbourhood from figure 2.2b would require a LUT of size 4096.

The LUT in table 2.1 could have a total of 256 configurations. Wolfram [14] labels these rules for basic 1D CAs with the numbers 0 through 255, where the number is the LUT. By writing out the results from high index to low for table 2.1 we get: off on on off on off off off, or 01101000, which is a binary number that translates to 104, meaning that the LUT described by the table is rule 104. Since the LUTs are complete, only 8 bits are needed to describe them. The rules for the neighbourhood in figure 2.2a can also be numbered in the same way, but with 32-bit numbers.

When it comes to state changes, uniformity is a very important variable. Do all cells follow the same rules (uniform), or do they follow different rules (non-uniform)?

If these are labelled as rules 0 through 255 [14], a CA with all its cells set to follow rule 30 would be described as uniform, while a non-uniform CA could potentially have all its cells follow different rules. Table 2.2 shows a few different uniform and non-uniform CAs.

Figure 2.5 shows an example run with a 2D 8x9 uniform cyclic CA with a 5 cell neighbourhood following the parity rule, i.e., a cell lives if an odd number of neighbouring cells were alive in the previous time step.

2.2 Genetic Algorithms

Evolutionary algorithms is a class of search heuristics inspired by Darwin's teachings on evolution and natural selection. It covers genetic algorithm (GA), genetic programming, evolutionary strategies, and more. This section will focus on GAs.

Genetic algorithms use concepts from genetics in combination with evolution to search for solutions to problems [5]. All GAs have a fairly similar execution, they all start with a population of individuals called a generation. Each individual has a genome which defines a possible solution to the problem. Further, they all follow some or all of these steps:

1. Calculate the *fitness* of each individual in the population.
2. *Select* a group of individuals based on fitness.
3. *Breed* selected individuals with each other to make new individuals for a new generation.
4. Make a selection of individuals from the old and the new generation that gets to live on and stay in the population.
5. Start over with the first step, calculating fitness, unless some termination criterion has been met.

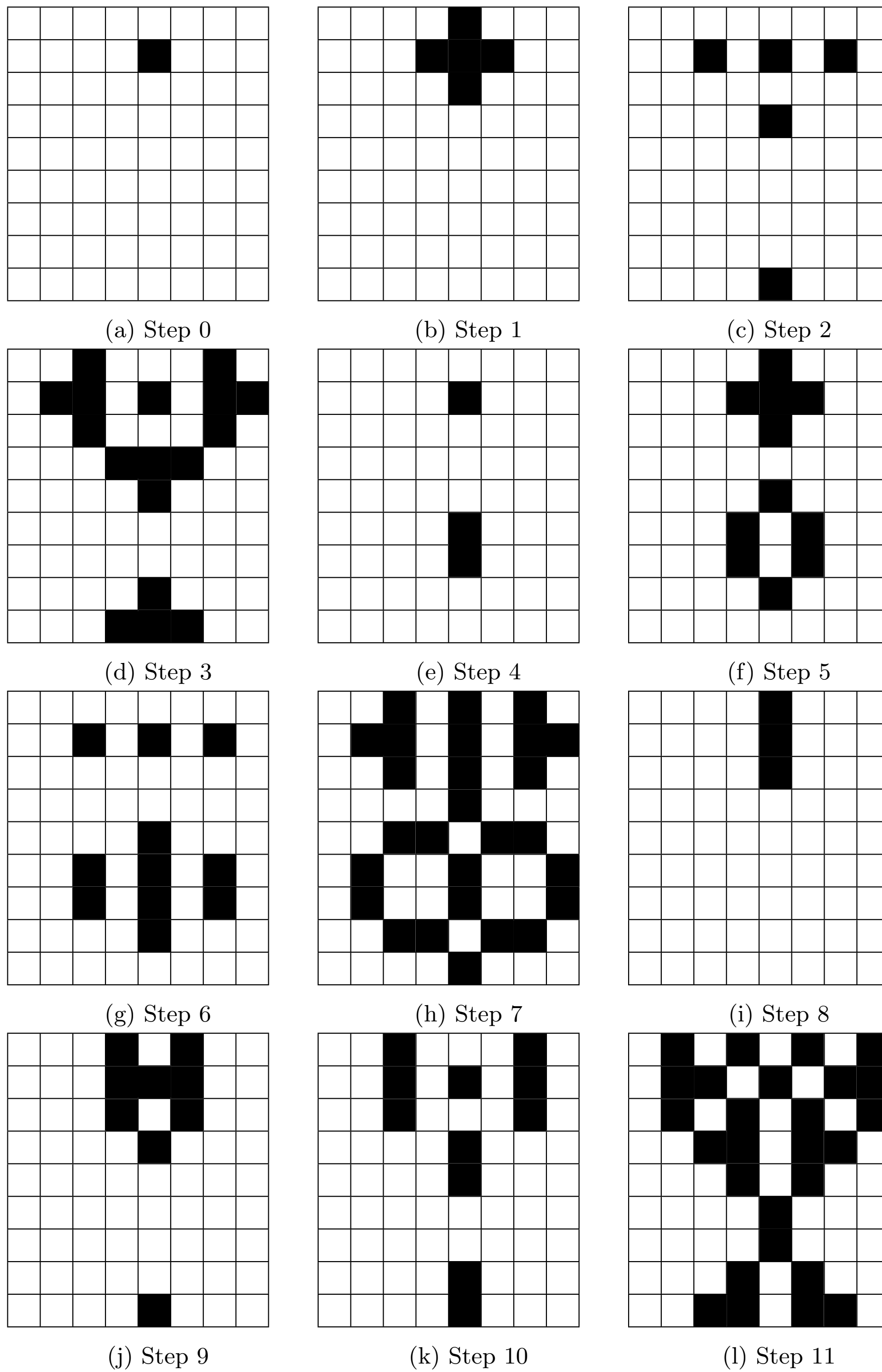

Figure 2.5: Example run of a 2D uniform cyclic CA following the parity rule. Black cells indicate live cells.

The fitness is defined by how good the solution is, its calculation can be anywhere from doing a simple check to running a complex simulation.

The genes are typically represented by a series of zeroes and ones. Breeding is done by combining genes from two individuals selected for their good fitness values. Genetic operations like crossover and mutations are applied when breeding. Crossover is used to combine genes from two individuals, taking some zeroes and ones from one of them, and some from the other. This results in one or more new individuals that may or may not have inherited good attributes from their parents. Mutations are applied to the genes of the new individuals by changing random zeroes to ones or ones to zeroes. This introduces entropy to the search and allows for search outside of the genetic material present in the original population.

Some GAs have the new generation of individuals replace the whole previous generation in the population, while others may choose some of the more fit individuals from the previous generation to live on to keep their good genes in the population.

2.3 Development

Most GAs use a *genotype-phenotype mapping* to translate an individual's genes, the genotype, to a possible solution, the phenotype. In many cases, the genotype directly encodes a set of attributes that make up the whole phenotype. For some problems however, the solutions are so large that evolving them directly using GAs becomes impractical. For such problems, having a smaller genotype be developed into a larger phenotype by some algorithm may be a better solution. The genotype can then encode a set of variables that the development algorithm uses to make the phenotype, it can encode variables that help make up the development algorithm itself, or even both.

In this thesis, development of non-uniform CAs is particularly interesting. Their development is based on cellular development in nature. A set of development rules define how new cells should grow, and how existing cells should change. Both development rules, cell types, and initial state can be encoded by a genotype.

One way to achieve development like this is to use a uniform multistate CA following a set of development rules. Table 2.3 lists a set of such rules for a 3-neighbourhood 1D CA. Each rule has an entry for the development state, called type, for each cell in the neighbourhood. For a rule to apply, or be a *hit*, each entry must match the type of their respective cell in the neighbourhood. The * is a wildcard, or a *do not care*, meaning that cell can have any type for the rule to apply. If several rules are a hit, the last of them is chosen. The result entry of the chosen rule determines the new type of the cell. If no rule is chosen, the cell keeps its type. Table 2.4 shows six development steps for a seven long cyclic CA following the rules from table 2.3. At each step all the cells are checked for all the rules, and updated accordingly.

All types used in development represent a LUT, or state change rules, as shown in table 2.5. All cells in the developed CA are assigned the LUT referred to by their type.

Rule#	West	Centre	East	Result
0	1	*	*	1
1	*	*	1	1
2	1	*	1	2
3	1	2	1	3
4	0	0	0	0

Table 2.3: Examples of 1D development rules. The three middle fields are conditions. The * means ”do not care”.

Cell	0	1	2	3	4	5	6
Step							
0	0	0	1	0	0	0	0
1	0	1	1	1	0	0	0
2	1	1	2	1	1	0	0
3	1	1	3	1	1	1	1
4	2	1	2	1	2	2	2
5	1	1	3	1	1	2	2
6	1	1	2	1	1	1	1

Table 2.4: Development steps for a cyclic 1D CA following the development rules from table 2.3.

Type	LUT
0	00000000
1	10010110
2	10101010
3	01101000

Table 2.5: Type to LUT conversion table. Converts developed types to LUTs used in a CA.

A GA using some aspect of the developed CA's behaviour as a fitness value can be used to evolve the development rules, the type to LUT table, the initial states and types of the CA, or even all them at same time. Sometimes evolving only the development rules is interesting because we know, or have some idea, what cell types are needed to solve the problem at hand. Other times we know less about what cell types are needed, in which case the LUTs can be encoded in the genotype along with the development rules, allowing for evolving them both concurrently.

2.4 Field Programmable Gate Array

An FPGA is an integrated circuit that is reconfigurable, the user can configure it to run his or her hardware designs [4]. They are often used by hardware designers in research, development, and verification before sending the design off for mass production. They are also useful for smaller markets or applications where implementing hardware in silicon is too expensive, and standard microcontrollers are not flexible enough. An FPGA is usually mounted on a printed circuit board (PCB), where it is connected to components as the power supply, light-emitting diodes (LEDs), switches, and other standard input/output (I/O) ports and components.

Figure 2.6 shows the main components and overall architecture of most FPGAs. The input/output blocks (IOBs) are connected to the PCB and its components, and handle all I/O. The interconnect is configurable and connects the different

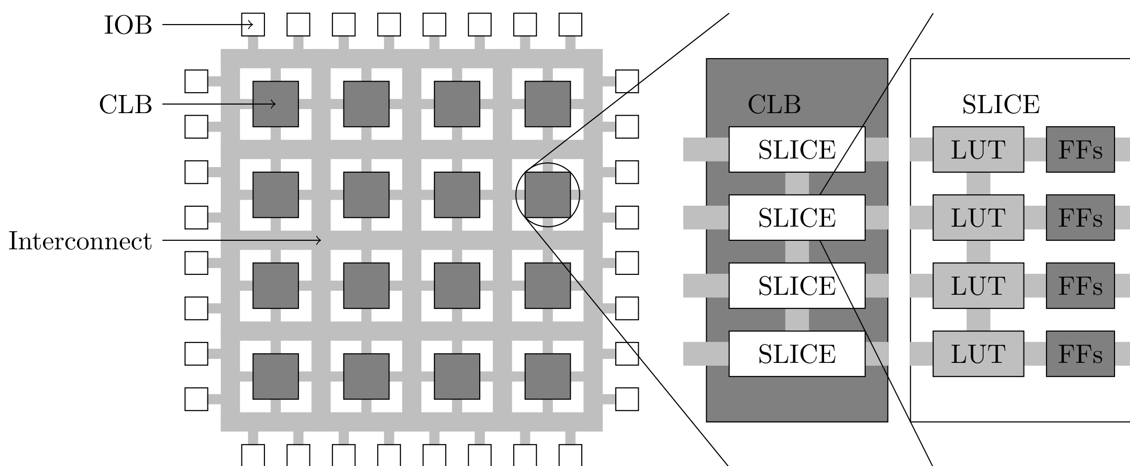

Figure 2.6: General FPGA architecture. CLBs and slices are shown in more detail.

FPGA components to each other. The CLBs are the main components of the FPGA, they implement the custom logic the user designs.

Each CLB usually consists of a few logic slices, where each such slice contains a few LUTs and a few registers. The slices can also have some simple memory called distributed memory as well as some computational logic like full adders. FPGAs often contain some block RAM (BRAM) for storing data as well as some digital signal processing (DSP) slices capable of addition, subtraction, and multiplication.

2.5 Evolvable Hardware

When applying the concepts of evolution and EAs to hardware we get EHW. The idea is to use an EA to evolve a hardware design that can be implemented on a configurable hardware device. FPGAs are highly configurable, so they make good target devices.

One use for EHW is finding novel designs that are difficult to find by humans. An EA explores a much wider range of alternatives than we could consider [19].

Another proposal is to use EHW to let pieces of hardware adapt to unexpected changes in its environment and become more fault tolerant [8]. By evolving in the field it can handle conditions it was not explicitly programmed to tackle.

Chapter 3

Previous Work

This chapter covers the research done by Haddow and Tufte [7, 13] and the work done by Djupdal [3] and Aamodt [1] in their theses, which this thesis builds directly upon.

3.1 An Evolvable Hardware FPGA and Development

FPGAs are widely used in EHW research, however because they are not specifically built for this purpose they are not as fitting as custom hardware could be. For conventional FPGAs, illegal configurations that could damage it are avoided by applying restrictions to the configuration. When evolving a design such restrictions can be difficult to implement, or undesirable. These are some of the reasons that Haddow and Tufte [7] suggest a new type of FPGA specifically designed for use within EHW.

They introduce the *sblock* as the base building block of the EHW FPGA. Sblocks are laid out in a 2D grid, an sblock matrix (SBM). Each sblock can only communicate with their four neighbouring sblocks, limiting the interconnect so that it cannot be misconfigured. The idea is that each sblock can be individually configured and illegal configurations are impossible.

Haddow and Tufte suggested several possibilities for internal sblock logic. This thesis will focus on the one depicted in figure 3.1, which shows an sblock in the SBM. Each sblock consists of a configurable LUT, and an output register (flip-flop (FF)). The LUT's size is 32 bits, and it uses the concatenation of the *north*, *east*, *south*, *west*, and *centre* signals, all coming from neighbouring cells' output signals, as the look-up address.

The SBM is used to implement a CA in this thesis, so the two terms are used interchangeably. The same goes for *sblock* and cell, as they are the base building blocks of the SBM and CA, respectively.

Tufte and Haddow [13] introduce knowledge based development rules. They are like the rules described in section 2.3, but with a differentiation between *change*

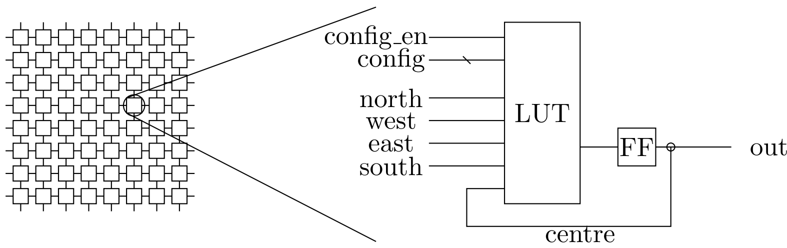

Figure 3.1: An sblock in an SBM. It consists of a configurable LUT and an output register, or FF.

and *growth* rules. Both rules apply only to sblocks that are not *empty*, one that does not have type *0*. When a growth rule is applied, the type of that cell is copied to a neighbouring cell. A type to LUT table like table 2.5 was also introduced. Such a table lets us define sblock types that have specific functions, like *XOR* or *Copy state from North*. Such predefined building blocks introduce some knowledge to the evolution of development rules.

3.2 Sblock Matrix and Development Process in Hardware

In 2003, as part of his master's thesis, Djupdal [3] designed and implemented an EHW design for a BenERA FPGA card. He also made a software driver to communicate with the design over peripheral component interconnect (PCI). The design is based on the work described in the previous section [7, 13], and both an SBM and a development unit is implemented on the FPGA.

3.2.1 Functionality

Figure 3.2 shows a very abstract overview of the system.

The SBM implemented is a 2D non-uniform cyclic two-state CA with a 5 cell neighbourhood. When running the CA, all the cells are updated at the same time in a single cycle, or run step. Several run steps can be executed back to back.

The development unit's functionality is much like the CA development example from section 2.3, but with slightly more complex development rules. The development unit works like a multi-state CA with the same 5 cell neighbourhood as the SBM. It runs in development steps, like the SBM runs in run steps. In each step, all the cells are developed "concurrently", but only two cells are developed at a time, so it takes more cycles to complete one step.

There is support for doing all necessary configuring over PCI. This includes development rules, cell type to LUT conversions, and initial states and types for the SBM cells.

Data can also be read back to the host machine over PCI. This includes the cell

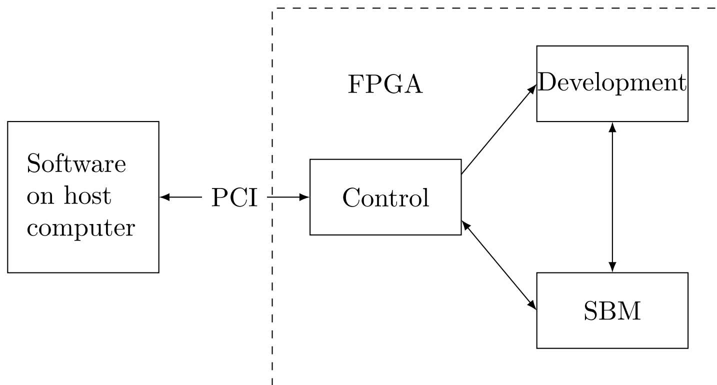

Figure 3.2: Abstract overview of Djupdal's system

type data generated by the development unit and state data from the SBM. Data can be read any time between development steps and run steps. When reading type or state data, data is read for either a single cell at a time, or for the whole CA.

Everything that happens on the FPGA is initiated with an instruction from the host machine. This includes all configurations, data read back, and run/development steps. Instructions can be stored in memory on the FPGA, and later be read and executed. Executing instructions from memory is faster than receiving them over PCI. This is useful when executing a set of instructions in a loop.

3.2.2 Architecture

The overall architecture of the hardware design is shown in figure 3.3.

The SBM is the main component in the design. It is a 2D array of sblocks that is basically a configurable cyclic non-uniform 2D CA with two states for each cell and a five cell neighbourhood as depicted in figure 2.2a.

The BRAM manager consists of two BRAM units, BRAM0 and BRAM1, each of which contains type and state data for each sblock in the SBM. They can swap all their contents with each other, which is particularly useful for development.

The LUT conversion unit is a table as the one in table 2.5. It stores the LUT data the sblocks are to be configured with, and is indexed with the types of the sblocks.

The Config unit reads the type and state data from BRAM1, and indexes the LUTconv unit with the type data to get the LUTs. It then configures the sblocks in the SBM with the LUT data.

The Readback unit reads the state data from the SBM, and stores it back in BRAM1, typically after running the SBM.

The Rule Storage contains development rules, like table 2.3, but with south and north entries and entries for the state for each cell as well. There is also a

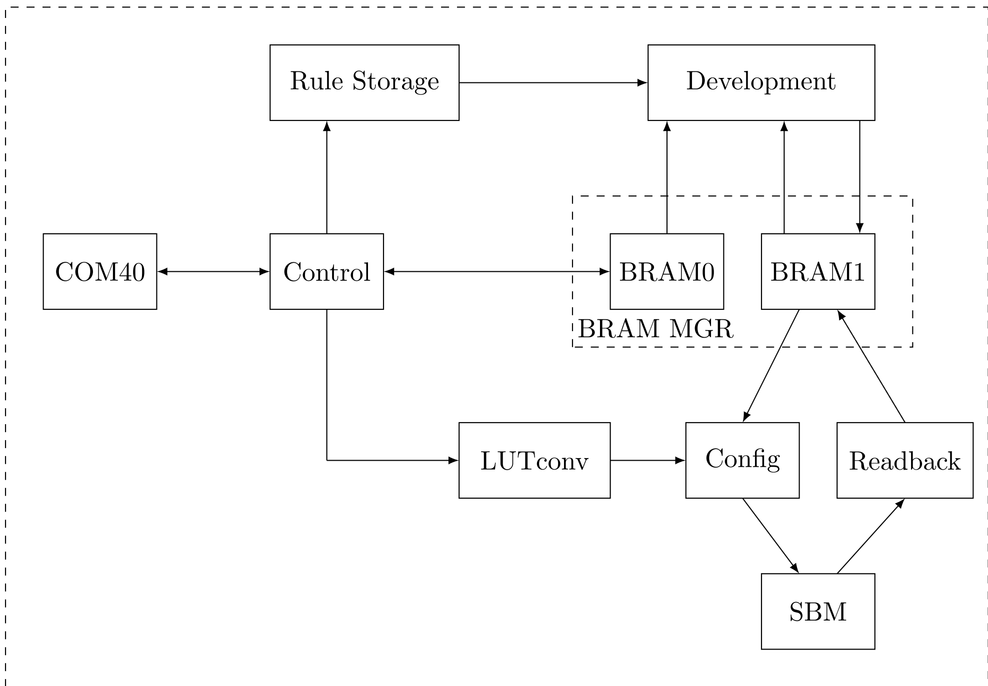

Figure 3.3: Architecture for Djupdal's design

differentiation between change and growth rules. Change rules may only apply to a non-empty cell and change its type to the one defined in the result entry. Growth rules may only apply to an empty cell and change its type to the type of a neighbouring cell defined by another entry in the rule. The rule format is described in appendix B.

The Development unit reads type and state data from BRAM0, checks the development rules it fetches from the rule storage for each sblock, and stores the developed type and state data in BRAM1.

The Control unit communicates with the host computer using the COM40 unit. It receives configuration data, instructions to activate the different units and to read result data from BRAM0 and send it back to the host. It also has a storage for storing instructions that can later be read and executed. Among the instructions is a jump instruction allowing for repeating some instructions until a set amount of developments have been executed.

A typical program flow for the architecture would be:

1. Configure the Rule Storage with evolved development rules.

2. Configure the LUTconv with predefined type to LUT conversions.

3. Configure BRAM0 with initial types and states.

4. Run development, reading from BRAM0 and storing results in BRAM1.

5. Configure the SBM with data from BRAM1 and LUTconv.
6. Run SBM for 150 steps.
7. Readback state data from SBM to BRAM1.
8. Swap data in BRAM0 and BRAM1.
9. Send result type and state data, now stored in BRAM0, back to host.
10. Jump to 4.

3.2.3 Implementation

This section's subsections describe the implementation of the units in the architecture in more detail.

Sblock Matrix

Each sblock is implemented as shown in figure 3.1. When a *run step* for the SBM is executed, the FFs in each cell are simultaneously updated to match their LUT's output. The LUTs' outputs are then updated as the outputs from their neighbours' FFs are used as their input.

The configurable 32-bit LUT in an sblock is implemented using two 16-bit shift register LUTs (SRLs). This means they can be configured in 16 clock cycles by feeding it two bits at a time.

The *readback* unit stores state data from the SBM in BRAM1 as fast as the BRAM allows, 8 states per cycle.

BRAM Manager

The BRAM manager contains two BRAM units that are used to store type and state data, BRAM-A and BRAM-B.

The BRAM manager's interface is made up of two sets of abstract BRAM interfaces, BRAM0 and BRAM1. Initially, these interfaces are linked to the BRAM-A and the BRAM-B units respectively. The BRAM manager can swap the links, so that BRAM1 is linked to BRAM-A and BRAM0 is linked to BRAM-B. The links are swapped back and forth with a simple instruction. From the outside it looks like BRAM0 and BRAM1 simply swapped their contents with each other.

BRAM-A and BRAM-B are identical, they both contain two dual-port BRAM modules for storing type data and two for storing state data. Each word in the BRAM modules contains data for two sblocks. With the two BRAM units with two read/write ports each, the maximum bandwidth becomes 8 types and states, where two and two of them are in the same word.

The type and state data is stored in raster order according to their position in the SBM. The rows of the SBM are stored alternatingly in the two BRAM modules that make up each of BRAM-A and BRAM-B, as shown by the white and grey rows of cells in figure 3.6.

The size of the stored types are 5 bits, which allows for 32 different types. The states are just 1 bit each, *on* or *off*.

Configuration Unit

The LUT conversion table is implemented using a single dual-port BRAM module with 32 words, one for each possible type. The type is used as the address and the LUTs are stored in it.

Figure 3.4 shows how the config unit works. It starts by reading single words of type and state data from BRAM1, two types and states each cycle. The type data is used to get the LUT data from the LUT conversion table, also two LUTs each cycle. The LUTs are stored in a set of registers.

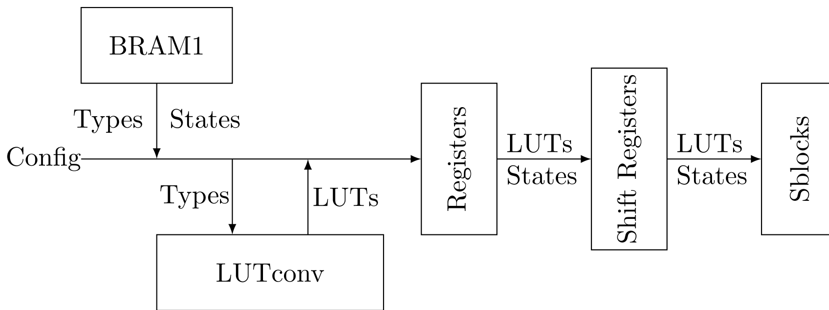

Figure 3.4: Data-flow throughout the configuration of the SBM

After 16 cycles, when 32 LUTs have been stored in the first set of registers, they are all dumped into a set of shift registers. During the next 16 cycles, 32 sblocks in the SBM are configured by shifting the LUTs out of the shift registers and into the sblocks, two bits at a time. At the same time, the first set of registers are loaded up with a new set of LUT data for a new set of 32 sblocks.

Effectively, this process configures 2 sblocks per cycle, and is repeated until all sblocks are configured.

Development Unit

Figure 3.5 shows the data paths through the development unit and its components. Type and state data are read from BRAM0 and development rules are read from the rule storage. Two groups of 8 *rule checkers* and the two *rule select* units do the main work of the development unit. The developed type and state data are stored in BRAM1.

Each rule checker can check one cell for one rule each cycle, and each group of 8 is used to check one cell for 8 rules simultaneously. A rule checker outputs whether the rule is a *hit* or a *miss* as well as the resulting type and state of that rule.

The two *rule select* units each read the outputs of one of the rule checker groups. It then selects the output for the highest priority rule that is a *hit* as its own output type and state, which is then stored in the currently checked cell's

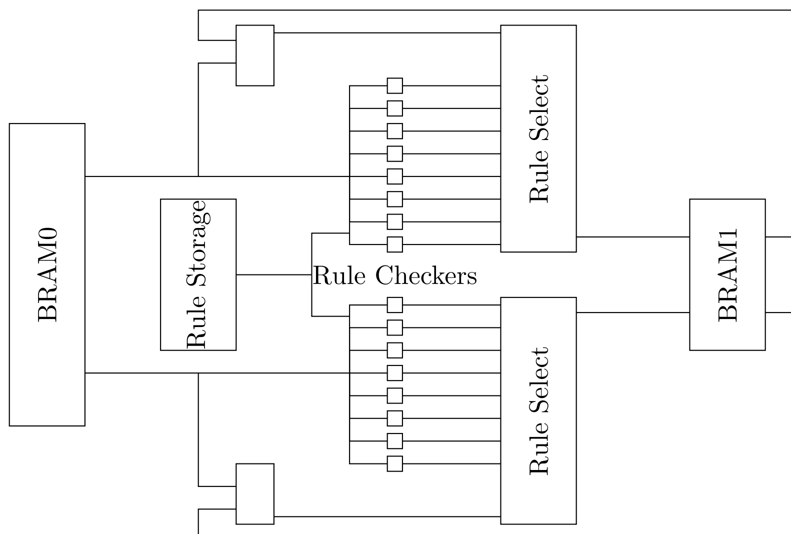

Figure 3.5: Development unit architecture

position in BRAM1. Rules are ordered from low to high priority in the rule storage, with higher priority rules being stored at higher addresses. If none of the 8 checked rules is a hit, rule select outputs the cell's original type and state as read from either BRAM0 or BRAM1.

The development unit starts by loading a set of 8 rules for the rule checkers. It then loads words of type and state data one by one (two entries per word) in raster order. After having checked all the cells for the first set of 8 rules, it loads the next set of 8 rules from the storage and checks each cell for all of those rules, and so on. When no rule from the first set is a hit for a given cell, that cell's original type and state from BRAM0 is selected by rule select and stored in BRAM1. When checking subsequent sets of rules, previous cell data used by the rule select units, in case no rule is a hit, are loaded from BRAM1 instead of BRAM0.

		N	N		
	W"	C'	C'	E	
		S	S		

Figure 3.6: Development: cells' neighbourhood vs BRAM layout

Figure 3.6 shows two cells, *C*, and their neighbours, *N, E, S,* and *W*. Type and state data for all these cells are needed for developing the two centre cells. The two northern cells are stored in the same word in the BRAM, as are the two southern cells, which are also stored in the same BRAM module. Since the BRAM modules have two read ports, both the N and S cells can be read at the same time. The

W, E, and Cs are all stored in the same BRAM module, but within three different words, so they cannot all be read at the same time. However, due to processing the cells in raster order, the development unit can simply store some cell data for eastern cells in some registers and use them as centre cells for the next cycle. The ' in figure 3.6 marks cells that were loaded during the previous cycle, and " marks the ones that were loaded in the cycle before that. This way, only one read port is generally needed for loading the middle row cells. The other read port is used to load the centre cells at the start of each row, and the south cell from the end of the previous row is used as the first west cell on the new row. For the first row, a single extra cycle must be spent to load all the data needed to get started.

The throughput of the development unit is effectively 8 rules for two cells each cycle. So for a 32×32 SBM with 13 development rules, development takes about 1024 cycles plus some setup time.

Control Unit

The control unit consists of several smaller units, *fetch*, *decode*, *hazard*, and *load, send, and store (LSS)*.

The fetch unit fetches instructions from either the host computer via the COM40 module or an internal instruction BRAM. Instructions can be read much more quickly from BRAM than from COM40, so storing instructions that are to be repeated many times, as in the program described in section 3.2.2, is preferred. There is a special instruction for storing instructions in BRAM. When it is received via COM40, all subsequent instructions are stored until the instruction to stop storing is received. The *jump* instruction is used to start executing instructions from a specified address in BRAM.

Figure 3.7 shows an example instruction, specifically the writeType instruction used to write a single type to a single cell in BRAM0. The five least significant bits indicate the opcode unique to the instruction. The next bit indicates whether the instruction is 32 or 64 bit. Bits 8 through 23 encode the x-y coordinates of the cell in the SBM and the type is stored in bits 24 through 28.

unused	type	y	x	unused	0	00001
31-29	28-24	23-16	15-8	7-6	5	4-0

Figure 3.7: writeType instruction from [3]

The decode unit receives the instructions from fetch, decodes them, and sets the appropriate control signals. Along with some instructions comes configuration data for rule storage, LUTconv or BRAM0. The decode unit stores data directly in the rule storage and LUTconv, while the LSS unit takes care of storing data to BRAM0. There are three instructions for storing data in BRAM0, one to store a single type for a cell, one to store a state, and one to set all types and states to a single type and state. The LSS also takes care of reading data from BRAM0 and sending it back to the host via COM40. There are instructions for sending single

types and states, as well as for sending all the types or states. When sending types and states, several are grouped together, sending up to 32 bits of data at a time.

The hazard unit makes sure that no hazards of any kind appears. It keeps the decode unit from issuing new instructions and the fetch unit from accepting new instructions while one is currently being executed.

The complete set of instructions is detailed in appendix A, do however note that this includes the new instructions introduced by [1] as well as the extensions made in this thesis.

3.3 Fitness Function in Hardware

The fitness function of a GA evolving the SBM typically needs to know something about how the states change from step to step. However, sending all the states back to the host for each run step takes a lot of time due to the relatively low efficiency of I/O compared to running of SBM, hundreds of cycles vs one cycle. For this reason Aamodt [1] implements the fitness function in hardware, on the same FPGA as the design from [3].

This section covers his additions to the design. They include the run step function (RSF), the RS BRAM, the fitness function, and two development output BRAMs. They are depicted in figure 3.8, highlighted by the thick lines and grey boxes.

The development output units, *vector BRAM*, and *dev-step BRAM* store some data about the development process that can later be sent to the host by the LSS unit. These units are not further explored in this thesis and are therefore not described in more detail here.

Also added to the design is an instruction for conditional looping, jumpEq. This allows for looping over a set of instructions until a specific number of development steps have been run. There is also a new instruction to reset the development step counter.

3.3.1 Run Step Function

The RSF is a unit that can read some statistics of the SBM, transform it, and store it in the RS BRAM, for each run step of the SBM. In Aamodt's design it implements the summation function. It counts the number of live cells, sblocks whose output state is one. The values are stored in the RS BRAM, one entry for each run step. It uses the same data busses as the readback unit, which is usually used to read 16 states every other cycle when storing data in BRAM1. However, the RSF reads 16 states each cycle. These 16 states are added together to a 5-bit number in one cycle using a small adder tree. An accumulator adds all the 5-bit numbers to a larger number, which is then stored in the RS BRAM.

The drawback of this function is that the efficiency of the SBM is reduced from 1 cycle per run step to *1/16th of the number of sblocks* cycles per run step. Which for a 32×32 configuration is 64 cycles. There is, however, an overall gain compared to having to send all the states back to the host at each run step.

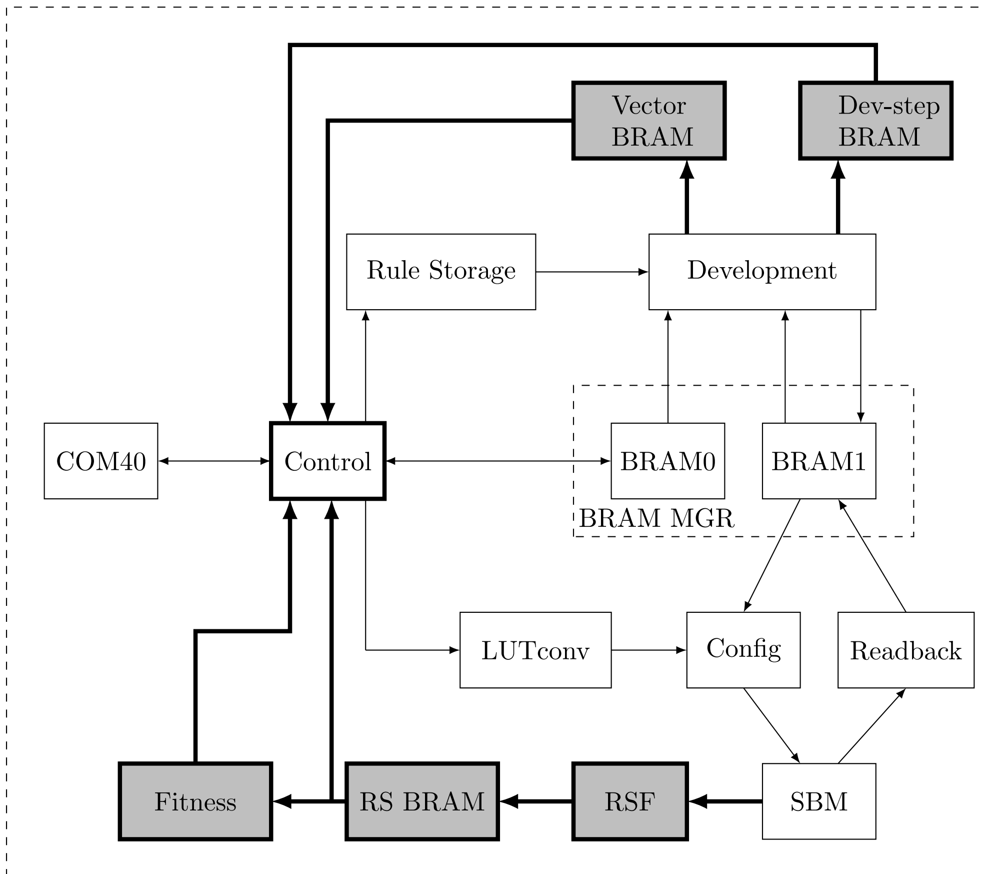

Figure 3.8: Aamodt's additions to the design

The RSF can be redesigned later to capture some other statistics of the SBM, for example the number of dead cells or the number of cells that changed their state from the previous cycle.

The contents of the RS BRAM can be sent back to the host using the LSS unit. It is also read by the fitness function.

3.3.2 Fitness Function

The fitness function uses the data stored in the RS BRAM to calculate a fitness value. In Aamodt's design [1] it simply finds the longest sequence of increasing numbers in the RS BRAM and outputs the length of the sequence as the fitness value. In combination with the RSF implementation, it means that the fitness of a CA is based on the longest sequence of subsequent steps where the number of live cells increases from step to step.

The fitness function, like the RSF, should be redesigned to match the needs of

the user. For example, the fitness could be based on how closely the number of live cells from step to step resembles a given curve.

The output of the fitness function can be sent back to the host using the LSS unit.

Chapter 4

Optimising for New Hardware

As mentioned in the introduction, the present work is based on improving and extending Djupdal's [3] and Aamodt's [1] designs in the anticipation of a newer hardware platform. This chapter starts off with section 4.1, an introduction to the new target hardware along with a comparison to the old hardware. The porting of the design to the new hardware is described in section 4.2, before the last section describes how each unit has been improved.

4.1 The New Hardware

This section compares the new FPGA, the Spartan6-LX150T [16, 17], and the development board it is mounted on with the older VirtexE-1000 [18] and its development board.

Table 4.1 compares some basic numbers for the two FPGAs, where some numbers are more useful in a comparison than others. The number of logic slices is slightly less than doubled in the newer FPGA, however each slice contains twice the amount of LUTs and four times as many FFs. The LUTs in Spartan6 devices are 6-input, while the ones in the VirtexE devices are only 4-input, making for an even greater difference between the two chips.

The number of logic cells (LCs) is a more abstract number that reflects the differences between the basic building blocks of the FPGAs, making it a better way of comparing the amount of customisable logic on the two units. One LC is comparable to one 4-input LUT. The VirtexE slices have two such LUTs as well as some extra logic making them count for 2.25 LCs each. The Spartan6 devices have four 6-input LUTs along with some extra logic in each slice, making them comparable to 6.4 LCs. Hence the great difference in LCs, 5.3 times as many in the Spartan6. The Spartan6 has 12.6 times as much BRAM as the VirtexE, as well as 3.5 times as much distributed random access memory (RAM). With all this extra logic and RAM, a faster design should be a possibility.

	Slices	LUTs	FFs	LC	BRAM	Dist. RAM	DSPs
VirtexE-1000	12 288	24 576	24 576	27 648	384 Kb	384 Kb	N/A
Spartan6-LX150T	23 038	92 152	184 304	147 443	4 824 Kb	1 355 Kb	180

Table 4.1: Resource comparison of the old and the new FPGAs

The Spartan6 has 180 DSP slices that can be used for addition, multiplication, and accumulation. There is little need for these operations in the base design, however they become more useful when looking at what we can do with the CA output in chapter 6.

The old development board uses a separate FPGA to implement PCI logic for communication with a host computer, making the I/O very cheap to implement on the VirtexE. This setup requires the communication between the two FPGAs to run at 40 MHz, which is the reason for the split into two clock domains in Djupdal's design, the rest of the design runs at 80 MHz. Spartan6 devices with a *T* ending their name, as in Spartan6-LX150T, have on chip support for PCI express (PCIe) endpoints. Using this requires the usage of some of the configurable logic on the device, making it more expensive than the PCI on the old hardware. PCIe supports communication at 100 or 125 MHz, which makes its clock a less limiting factor than that of PCI.

4.2 Porting the Design

The first step in optimising the design for the new hardware is to make it work on the new hardware, that is, make implementation on the new target FPGA possible.

In the original design, BRAMs and SRLs were mainly implemented by referencing device specific components that are not available on the Spartan6. Instead of updating their initialisation to match Spartan6 components, they are replaced with complete implementations in the code. This way, the synthesis tool can map them to appropriate device components regardless of what FPGA is used.

The largest task in porting the design is making the I/O work with the new PCIe endpoints that replace the PCI FPGA on the old hardware. This task is not handled in this thesis due to circumstances explained in section 8.2.

After porting the design, comparing the synthesis results for the old device from [1] with the results for the new design gives a good impression of how much more we have to work with now. Table 4.2 shows the results for synthesis of a 8×8 and a 32×32 cell configuration for both devices. The tri-state buffers (TBUFs) on the VirtexE turned out to be the limiting factor, limiting the design to a 32×32 configuration. Modern FPGAs, like Spartan6, no longer have internal TBUFs, so all tri-states in the design are replaced with logic like multiplexers. Do also note that the speed for the VirtexE was limited by the communication unit, which will also be the case for the Spartan6, while the Spartan6 speed shown here is a synthesis estimate. The speed for the Spartan6 would probably be limited to 125MHz by PCIe. What we see in the table is that the percentage of BRAMs used is about 10

Device	Size	Speed	BRAMs	Slices	TBUFs
VirtexE	8×8	80MHz	79%	32%	2%
Spartan6	8×8	170MHz	8%	7%	N/A
VirtexE	32×32	80MHz	79%	49%	10%
Spartan6	32×32	170MHz	8%	11%	N/A

Table 4.2: Comparison of synthesis results for the original design on the VirtexE and the Spartan6

times lower for the Spartan6, and the percentage for the slices is about 4-5 times lower, which is in line with the comparisons in the previous section.

4.3 Parameterisation of the Design

When improving the design one of the key goals was to make it more parameterised, making as much as possible configurable using a set of parameters.

Very few aspects of the original design were configurable. In Djupdal's design, only the width and height of the SBM as well as the number of development rules checked at the same time are configurable. Aamodt's additions to the design are somewhat configurable when it comes to the amount of BRAM used by each new RAM unit, and the efficiency of the fitness function.

Most parts of the design have been made more configurable in this thesis. Some of the most important are number of types and states per word in the BRAM, how many cells are developed concurrently, and how many sblocks are configured at the same time. Each of these greatly affect the efficiency of different parts of the design. Many variables that depend on other configurable variables have also been added. For example the amount of read ports in the LUTconv is very dependent on how many sblocks are to be configured each cycle.

Some aspects of the design have been partially parameterised, meaning that changing the variables would break the design unless some parts are rewritten to match the new values. For example, the amount of BRAM modules used by BRAM-A and BRAM-B is configurable, however changing them does not make much sense without changing many other parts of the design, which is done for the optimisations described in this chapter as well as the extension described in chapter 5.

4.4 Design Improvements

This section's subsections describe how the efficiency of the different units in the design has been improved. Figure 4.1 highlights with thick lines and grey boxes the parts that have been changed.

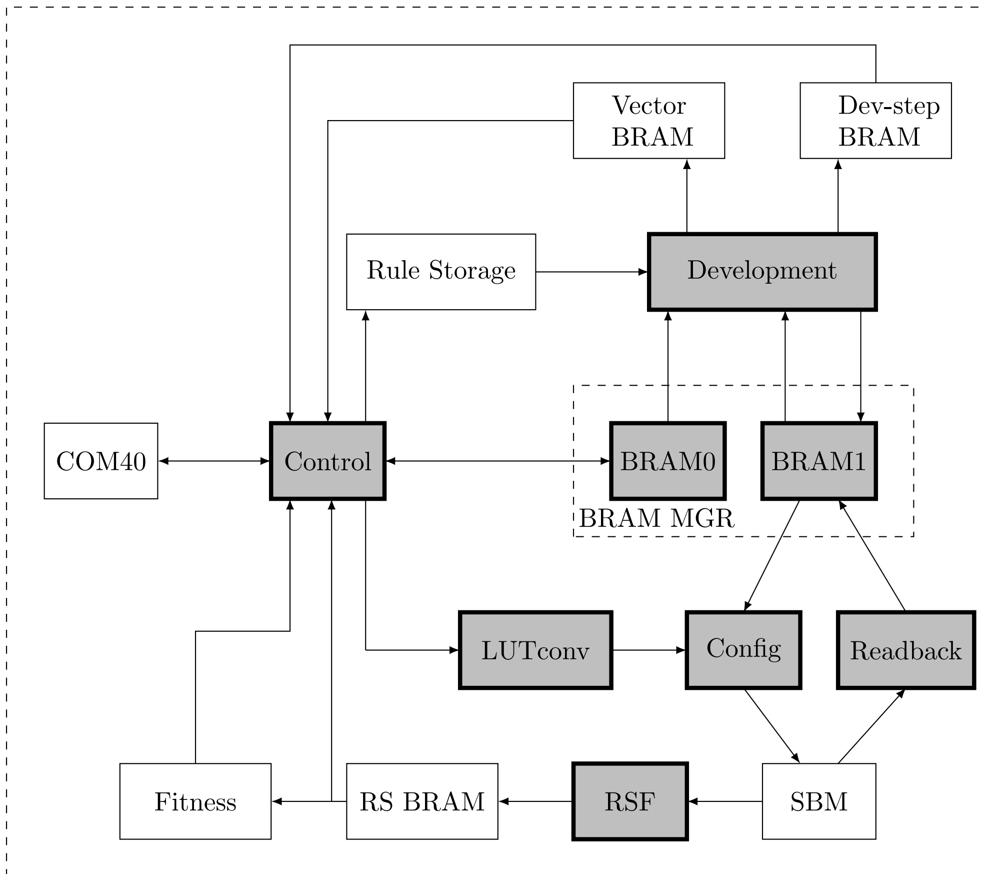

Figure 4.1: Optimised parts in the new design, highlighted by thick lines and grey boxes.

4.4.1 Block RAM

To allow for a higher memory bandwidth, the parameterised BRAM manager with its two BRAM units are reconfigured to use more BRAM modules, with wider data busses. The physical BRAM modules on the Spartan6 have a maximum data bus size of 2×32 bits, two times because it is dual-port, so two words can be accessed at the same time. The new design uses four dual-port BRAMs with four cell types and states in each word, with a maximum cell type size of 8 bits. In comparison, the old design uses two dual-port BRAMs with two cell types and states in each word, and 5 bits per cell type. Ignoring the difference in cell type size, the bandwidth for the new design is 4 times higher than that of the old one. This allows all the units connected to the BRAM manager to work faster. Figure 4.1 shows these units as being *control, development, config,* and *readback*.

It is important that all the units using the BRAM manager agree on what is

0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3

(a) New

0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1

(b) Old

Figure 4.2: BRAM layout for the new and old design. The grids represent SBM, and each cell represents a word of sblock data. The numbers indicate in what BRAM module the sblocks' data is stored.

stored, and where. Figure 4.2a shows a 32×6 part of the CA grid, where four and four cells are grouped into one cell in the table. The numbers indicate which BRAM module those cells' data are stored in. The old design using only two BRAM modules per BRAM unit has a somewhat simpler layout. Figure 4.2b shows this in a 16×6 cut of the CA grid where each number represents a word of two cells. It is important to remember that each BRAM module is dual-port, meaning two words from the same module can be read or written at the same time.

Both the old and the new layouts were chosen to optimise for the development unit's reading of data.

4.4.2 Development

In addition to making the development unit faster, the new memory layout allows for it to be simplified as well. As described in section 3.2.3, the development unit works by looking at sets of cells in raster order, and testing each cell for a set of development rules. The testing of rules requires the type and state of each cell in the tested cell's neighbourhood.

Figure 4.3 shows what cells are developed at the same time (C) along with their neighbours (N, E, S, *and* W), for both the new and the old design. The shades of grey represent the different BRAM modules that the cells' type and state data are stored in. In the old design, two cells are developed at a time. Due to the memory layout, all neighbouring cells' data cannot be read at the same time. By preloading, and keeping some cell data between the sets of cells that are developed, a speed of almost two cells per cycle is still achieved.

By taking advantage of the increased memory bandwidth, the new design can develop 8 cells per cycle. Due to the new memory layout, the data for all the neighbours can be loaded simultaneously, removing the need for preloading data, hereby simplifying the design and saving a few cycles.

The main resource in the development unit is a set of rule testers, each of which can test one cell against one rule each cycle. In the old design, there are 16 such testers, 8 for each cell tested concurrently, which allows for testing against 8 rules at the same time. The development process must go through each cell

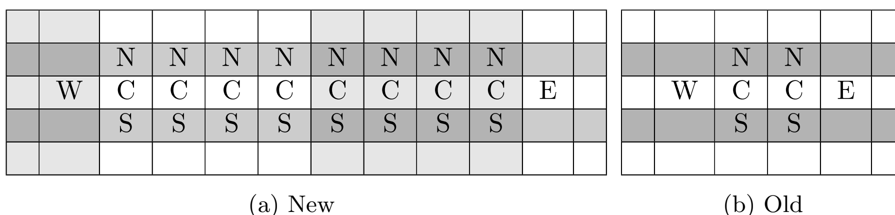

Figure 4.3: Development: cells and their neighbourhoods vs BRAM layout in the new and the old design

once for each set of 8 rules. To increase the performance of the development unit more testing units have to be added. If the new design used only 16, it would be as slow as the old design, even though it works on more cells at the same time. Increasing the amount of rule testers to 256 would potentially increase the speed of development if there are enough rules. Here we have an area/performance trade off with diminishing returns where a halving in time spent equals a doubling of resources used. The amount of testing units is therefore parameterised, so the user can decide how many would be best considering the amount of rules and how much time development takes compared to other processes.

For the new design, the growth and change rules have been combined to a new simpler type of change rule. This makes the rule testers slightly simpler, and less resource demanding. One can still achieve the same growth rules as before by using a few more rules to represent them.

4.4.3 Config

Configuring a cell in the matrix is done by first reading type and state data from the SBM BRAM, then looking the type up in the lut conversion memory which yields a lut configuration that can be fed into the sblock in the SBM. The performance of the configuration unit is limited by SBM BRAM bandwidth and number of LUT conversions that can be done each cycle.

The LUT memory is a single dual port BRAM module in the old design, meaning only two LUT configurations can be read each cycle, while the total SBM BRAM bandwidth is 8 cell types and states. Section 3.2.3 explains the workings of the configuration unit. In short, the performance is effectively two sblocks each cycle, as limited by the LUTconv BRAM.

In the new design, the LUT memory is implemented using several BRAM modules where the same data is stored in each of them, resulting in a memory with one write port and a configurable amount of read ports. The number of read ports is equal to the amount of sblocks we want to configure per cycle. Since there are a lot of free BRAM modules on the Spartan6, the maximum amount of read ports is near limitless compared to the logic expenses of configuring the sblock. The maximum bandwidth to the SBM BRAM is 32 cell types and states.

4.4.4 Run Step Function

The unit that sums up the number of live cells at each run step, the RSF, is the one part of the old design that slows down the running of the SBM. 16 cell states are read at a time, summed up and added to the total. For a 32×32 SBM this takes 64 cycles at each run step. When it is not used, each run step takes just one cycle.

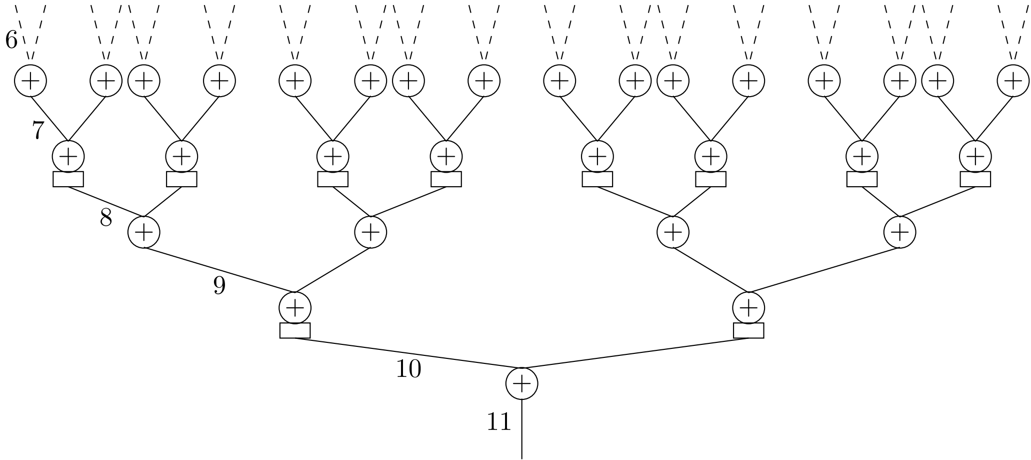

Figure 4.4: Adder tree with clocked registers. It takes 1024 1-bit input signals and outputs the sum of them as an 11-bit number.

The new design employs a large adder tree with pipeline registers to achieve a throughput of one full summation of the matrix per cycle. Figure 4.4 shows such an adder tree, where the circles represent adders and the rectangles represent clocked registers. At the top of the tree (not shown) four and four states from the SBM are input to nodes that add the four states to a 3-bit number. From there, two and two 3-bit numbers are added to a 4-bit number, then two and two of those are added to a 5-bit number, and so on going down the layers of the adder tree. At the bottom is the output, which given a 32×32 SBM is an 11-bit number. At every other layer of the tree there are clocked output registers from the adders. This is to keep the adder tree from forming a critical path in the design, which would lower the potential clock rate of the system.

Such an adder tree requires quite a large amount of resources compared to the accumulator in the old design, but it allows for the CA to be run at one step per cycle with a full addition of the states for each run step.

4.4.5 New Instructions

Formerly, there were only three instructions for writing states and types to BRAM0, one to write a single cell type, one for a single cell state, and one for setting all cells to a specific state and type. Two new instructions are added to the instruction set

in the new design, one for writing a whole word of four types to the memory and one for writing 16 states. Using these instructions makes writing an initial pattern of states to BRAM0 faster, and resetting to the initial state pattern between runs can more easily be done with just 1/16th of the instructions and time used before.

Chapter 5

Three Dimensional SBlock Matrix

This chapter introduces the 3D CA and how it is implemented in the optimised EHW design described in chapter 4. The goal is to extend the SBM into 3D space while keeping as much as possible of the design as it was before. Figure 5.1 highlights the affected units with grey and thick lines.

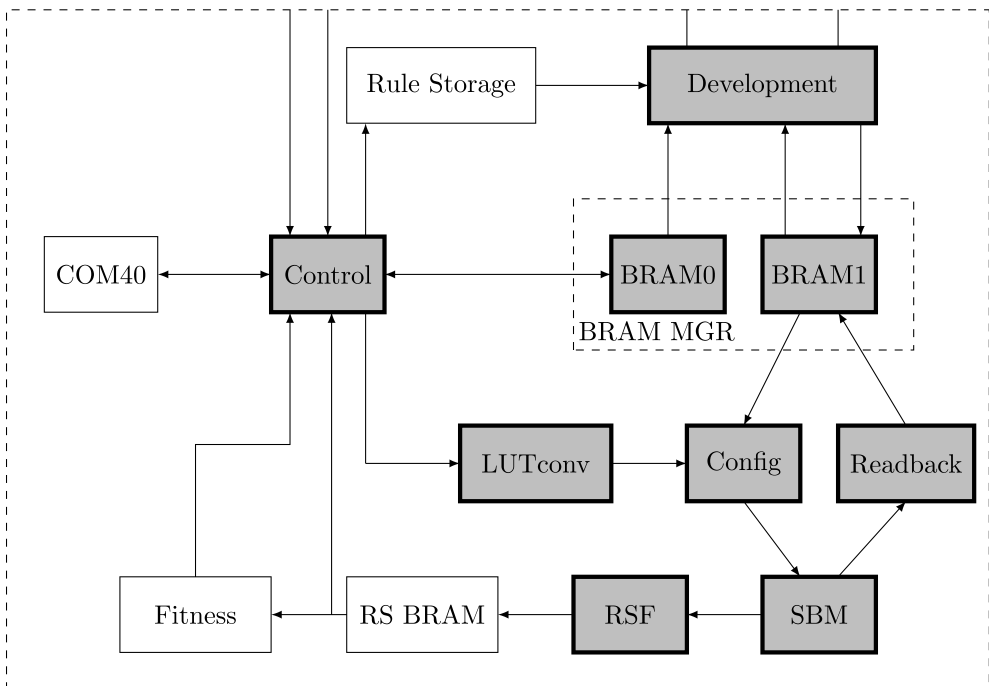

Figure 5.1: Parts changed for the 3D extension, highlighted in grey and thick lines.

5.1 The SBlock Matrix

By adding *layers* to the *rows* and *columns* of the SBM, it is extended to be a cube in 3D space, as seen in figure 2.1c. Each sblock has new inputs *up* and *down* in addition to the standard *north, east, south, west,* and *centre* which creates the neighbourhood shown in figure 2.4. For each additional neighbour to a cell the LUT size is doubled, meaning we now need 128-bit LUTs instead of 32-bit ones.

A configurable LUT is implemented using the distributed RAM capable slices as SRLs. There are 5420 such slices in the Spartan6-LX150T, each of which can be configured as up to four 32-bit SRLs, or eight 16-bit ones. Eight 16-bit SRLs can be combined to the 128-bit LUT needed for a single sblock. This means that the largest possible configuration that is a power of two is a 16×16×16 3D SBM which would use about 76% of the relevant slices.

5.2 BRAM and Development

To make development of an sblock efficient the BRAM manager needs to accommodate for reading all data for a set of cells and all their surrounding neighbours in just one read cycle. To make this achievable for a 3D SBM the amount of BRAM modules used in BRAM-A and BRAM-B is doubled from the four used in the 2D design. The word size is also changed to be half of the size of the SBM in the x-direction, 8 entries per word for a 16 cell wide SBM.

The new layout, shown in a similar manner to those in figure 4.2, is depicted in figure 5.2. This is the BRAM layout for a 8×8×8 SBM. Each number in the tables points to the BRAM module used for storing a word of four cell types and states, and the column number is the x-position of the first cell in that word.

column:	0	4	0	4	0	4	0	4	0	4	0	4	0	4	0	4
row:																
0	0	1	4	5	0	1	4	5	0	1	4	5	0	1	4	5
1	2	3	6	7	2	3	6	7	2	3	6	7	2	3	6	7
2	0	1	4	5	0	1	4	5	0	1	4	5	0	1	4	5
3	2	3	6	7	2	3	6	7	2	3	6	7	2	3	6	7
4	0	1	4	5	0	1	4	5	0	1	4	5	0	1	4	5
5	2	3	6	7	2	3	6	7	2	3	6	7	2	3	6	7
6	0	1	4	5	0	1	4	5	0	1	4	5	0	1	4	5
7	2	3	6	7	2	3	6	7	2	3	6	7	2	3	6	7
layer:	0		1		2		3		4		5		6		7	

Figure 5.2: 3D SBM BRAM layout

All units using the BRAM manager are changed to match the new layout, these include the LSS, config, readback, and development units, all of which can now take advantage of higher bandwidth of the BRAM.

Also depicted in figure 5.2 is a selection of sblocks (dark grey) and all their neighbours (light grey). This selection includes two whole rows of layer 4 in the

SBM and the neighbours above in layer 3, below in layer 5, and to the north and south in layer 4, all of which can be read from the BRAM simultaneously. Due to selecting two complete rows, all western and eastern neighbours are included in the dark grey selection. This is the largest selection of words we can read simultaneously from the BRAM and thus limits development to work on two rows in a layer of the CA at a time, which for a 16×16×16 configuration would be 32 cells.

The development unit works on two rows of a layer at a time, so when decreasing the size of the SBM one can decrease its resource usage by lowering the amount of columns or keep the performance up by lowering the amount of layers or rows instead. One can also, as before, change the performance of the development unit by changing the amount of rules that are checked at a time.

To support 3D development the rule format, which is described in detail in appendix B, is updated to include *up* and *down* entries. The rule testers are extended to check these entries as well.

5.3 Config

The one part of the original design that scales the worst when increasing the LUT size is the configuration unit. Specifically shifting of data into the sblock LUTs is very expensive, and to keep performance up we want to shift more data at the same time than in the 2D configuration with 32-bit LUTs.

The 2D design configures the sblocks by first spending some cycles to read configuration data from a memory and storing them in 32-bit registers. The data in the 32-bit registers are then moved to another set of 32-bit registers before being fed into the sblocks by shifting the registers. While feeding the sblocks, another set of configuration data is loaded into the first set of registers again, so they are ready to be used by the time the first set of sblocks have been configured.

128-bit shift registers would be four times as expensive as the already quite expensive 32-bit ones, so simply changing them to 128-bit ones is not a feasible solution if we want to keep performance up. Instead, the 3D design's config unit can use smaller shift registers by reading only parts of an sblock's configuration data at a time. The LUTconv is configured to store smaller parts of the configuration data in each word, which means that it needs to be read more times per sblock, which in turn means that it has to have more read ports, and more BRAM modules to keep the performance up. Specifically, a halving in shift register size leads to a doubling in BRAM modules used.

5.4 Run Step Function

If using only an adder tree to sum up all the outputs of the SBM, a 16×16×16 configuration leads to an adder tree a bit more than 4 times as large as a 32×32 configuration. For this reason, an accumulator has been reintroduced to the RSF, and the size of the adder tree has been made configurable. The RSF for the 3D

design is thus more similar to the one in [1], but with a potentially larger adder tree.

When using an adder tree large enough to sum all the outputs, each run step still uses a single cycle. For every halving in adder tree input size, logic used by it is more than halved, and the number of cycles spent per run step is doubled.

5.5 3D Instructions

The instruction set has been changed to support the new 3D architecture. Figure 5.3 compares an old and a new instruction. The opcode has been increased to 6 bits, allowing for doubling the amount of instructions. The base instruction size is increased to 64 bit. All instructions referencing specific cells in the CA have been extended to include a z variable in its coordinate data.

unused	type	y	x	unused	0	00001
31-29	28-24	23-16	15-8	7-6	5	4-0

unused	type	z	y	x	00	000001
63-40	39-32	31-24	23-16	15-8	7-6	5-0

Figure 5.3: Comparison of the writeType instruction from [1, 3] (top) and this thesis (bottom)

Since the LUTsize is increased to 128 bits, the instruction that stores the configuration data has been extended to be a 196-bit instruction. Therefore support for up to 256-bit instructions have been added to the instruction set.

The instruction for storing the development rules is extended to have the *up* and *down* fields, making them 22 bits larger.

The same instructions are used for both the 2D and 3D design, except for the LUT data instruction which is a 64-bit instruction in the 2D design and 196-bit in the 3D design. The 2D design simply ignores the 3D-specific fields of the instructions.

The new instruction set is detailed in appendix A.

Chapter 6

Cellular Automaton Output

When evolving a CA a measure of how well the CA is performing, a fitness value, is needed. The RSF module added by Aamodt [1] simply sums the number of live cells in the SBM. The fitness function then uses this data when calculating a fitness value. This chapter explores transforming the output data from the SBM using the discrete Fourier transform. It starts off with an introduction to the discrete Fourier transform (DFT), and why it might be interesting, followed by a description of a hardware implementation.

6.1 Discrete Fourier Transform

The DFT is a transformation of a series of numbers to the frequency domain. The equation below defines the DFT, X, of a series of complex numbers x.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}, \quad k \in [0, N-1]$$

If x contains only real numbers, the latter half of X is simply the first half reversed, so for real x we are only interested in calculating $X_{0\ldots N/2}$. The x_n part of the equation is the input to the DFT, while $e^{-i2\pi kn/N}$ is called the twiddle factor, and its only dependence on the input is N, the length of the series. The twiddle factors, given an input size N, can thus be written as a function $T(k,n) = e^{-i2\pi kn/N}$.

Simulations have been run to try to find out whether different transforms of the output data of a CA produce more interesting data than the raw output data. Several transforms were explored, but the only promising one was the DFT. Many simulations using the same CA but with different starting sets of live cells resulted in the average data plotted in figures 6.1 and 6.2. Figure 6.1 shows the average live cell count for 128 run steps, whereas figure 6.2 shows the average absolute value of the DFTs of the same source data as the first figure. The black line in the figure is the average and the grey area surrounding it is the standard deviation.

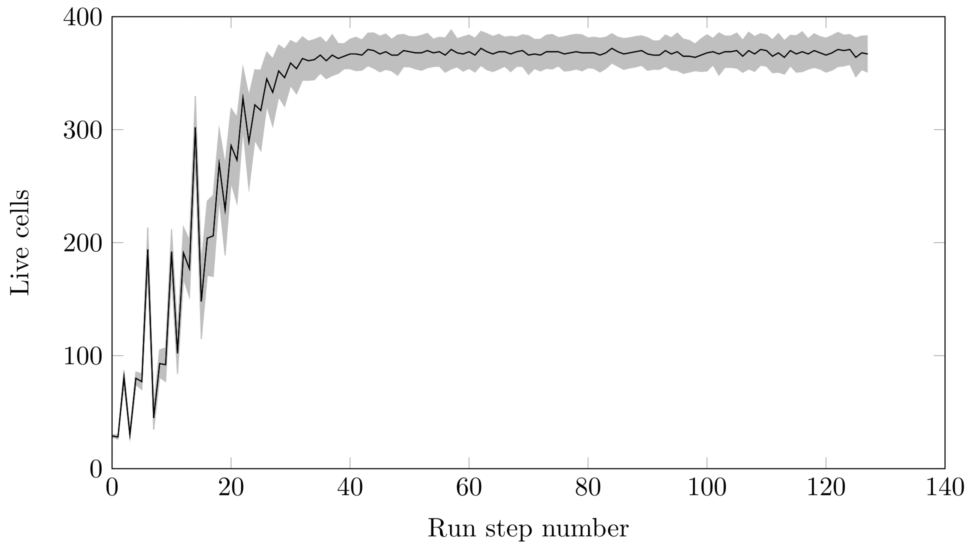

Figure 6.1: Average live cells per step for a specific CA

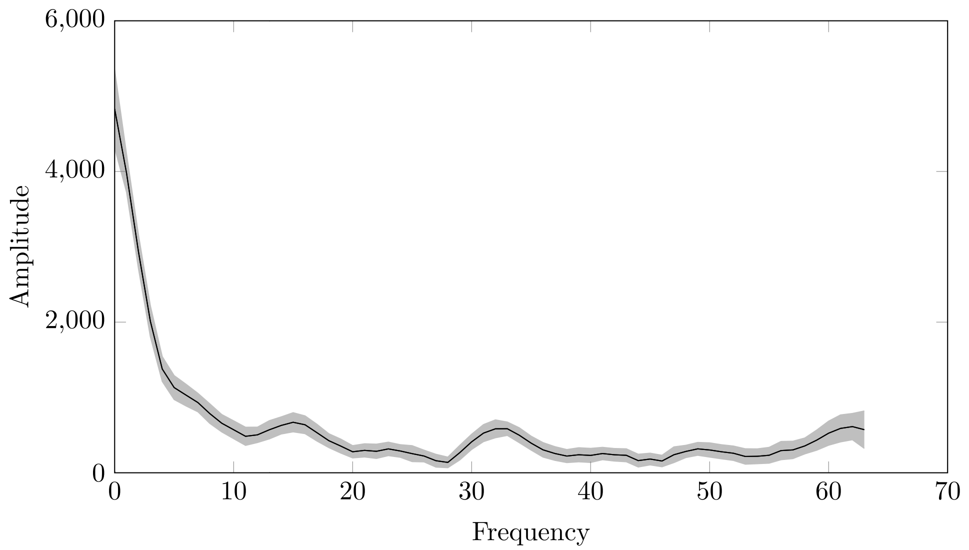

Figure 6.2: Average DFT of live cells per step for the same CA

An interesting aspect of the DFT is that repeating patterns that can be difficult to recognise in the input sequence, as in figure 6.1, can more easily be recognised by the peaks in a DFT. Many CAs are likely to have such repeating patterns in their output, which is one reason that DFTs may be interesting when evolving them.

In 2013 Berg [2] explored the evolution of CAs and found that using DFTs of the output data from the CA in the fitness calculation yielded good results, promoting the point that a hardware implementation could be interesting.

6.2 Implementation

Figure 6.3 shows the DFT unit placed between the RS BRAM and the fitness function. It uses data from the RS BRAM as input, and its output is added to the input for the fitness function.

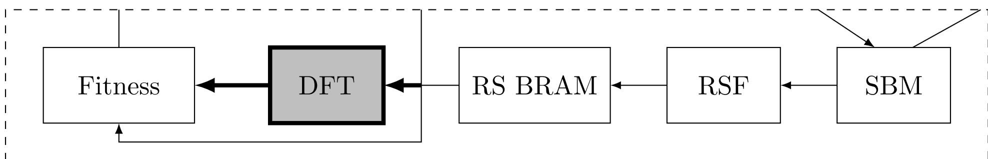

Figure 6.3: The DFT in the architecture

The main resources required for DFT calculations are multiplicators, accumulators, and twiddle factors calculators. Multiplication and accumulation are two of the main features of the DSP slices on the Spartan6 FPGAs, of which there are 180 on the LX150T. This accounts for two of the resources needed. However, calculating twiddle factors is more complex.

The twiddle factor problem is solved by limiting the size of the DFT at synthesis. The twiddle factors needed for a specific DFT size are all calculated before programming the FPGA. They are then stored in a BRAM module on the FPGA in the order they are needed. Table 6.1 shows how the twiddle factors are stored in memory. $T_r(k, n)$ represents the real part of the twiddle factor that should be multiplied with the nth value of the input series in the calculation of the kth value of the DFT output. $T_i(k, n)$ is the same for the imaginary part. Depending on the DFT size and the precision of the twiddle factors, the twiddle memory may require many BRAM modules to be implemented due to the high bandwidth required.

Fast Fourier transform algorithms uses $O(N \log N)$ computations and are normally the go-to choice for DFTs. However, a simpler $O(N^2)$ algorithm is easier to parallelise, especially given a set DFT size. Given a DFT input size of N, we can use N DSPs in parallel to compute the DFT in just $O(N)$ time. Each DSP is used to compute the real or imaginary component of a single output number, X_k.

Figure 6.4 shows a simplification of how the DSPs are used. The multiplicator takes data from the RS BRAM and twiddle factor memory as input. The accumulator is an adder with an output register. The adder takes its own output

Address	Data						
0	$T_r(0,0)$,	$T_i(0,0)$,	$T_r(1,0)$,	$T_i(1,0)$,	...	$T_r(N/2\text{-}1,0)$,	$T_i(N/2\text{-}1,0)$
1	$T_r(0,1)$,	$T_i(0,1)$,	$T_r(1,1)$,	$T_i(1,1)$,	...	$T_r(N/2\text{-}1,1)$,	$T_i(N/2\text{-}1,1)$
2	$T_r(0,2)$,	$T_i(0,2)$,	$T_r(1,2)$,	$T_i(1,2)$,	...	$T_r(N/2\text{-}1,2)$,	$T_i(N/2\text{-}1,2)$
3	$T_r(0,3)$,	$T_i(0,3)$,	$T_r(1,3)$,	$T_i(1,3)$,	...	$T_r(N/2\text{-}1,3)$,	$T_i(N/2\text{-}1,3)$
⋮	⋮	⋮	⋮	⋮		⋮	⋮
N-1	$T_r(0,N\text{-}1)$,	$T_i(0,N\text{-}1)$,	$T_r(1,N\text{-}1)$,	$T_i(1,N\text{-}1)$,	...,	$T_r(N/2\text{-}1,N\text{-}1)$,	$T_i(N/2\text{-}1,N\text{-}1)$

Table 6.1: Twiddle BRAM content

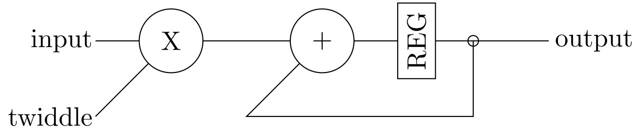

Figure 6.4: Simplified DSP slice

as well as the output from the multiplicator as input. The DSP module can use any consecutive N number portion of the RS BRAM for the input values.

In case we want to have a larger DFT than we have DSPs for, or simply want to use less DSPs due to resource problems, there is support for using less of them. In this case, a smaller set of output data will be computed first, before moving on to the next set, and so on until the whole DFT is calculated. As one would expect, for every halving of DSPs, the time spent is doubled. The twiddle memory is also rearranged to take advantage of the lower bandwidth that is needed.

Appendix A covers the additional instructions introduced for the DFT unit.

Chapter 7

Testing and Results

The first section of this chapter describes how functional correctness testing has been done during this work. The subsequent sections focus more on results from synthesis of the design as well as the efficiency of the different units given different configurations. There is one such section for each main chapter in the thesis.

The process of compiling the hardware design and mapping it to specific components on an FPGA is called synthesis. This is done using Xilinx ISE [15], which is described in appendix D.

7.1 General Correctness Testing

Throughout the thesis project, functional and correctness testing have been done using a simulation test bench combined with a C-program that generates instructions for the design. The testing has been done in a simulation run by using ModelSim [6].

The test bench is a simple wrapper around the top-level of the design. All it does is to set a few PCI signals and run the input clock signal. The c-program generates ModelSim instructions to set all other input PCI signals. The PCI data bus signals are set to instructions from a test program made for the hardware.

The functional test used is taken from [1, 3], and is described in figure 7.1. The output produced by the readTypes and readStates instructions at the end of the simulation is compared to what we know the output should be for the given test.

Throughout the design work the test has been modified to do more specific tests of specific parts of the design. ModelSim can generate waveforms of a set of chosen signals when running a simulation. So, for closer inspections of parts of the design, one can select the relevant signals and see cycle by cycle if they behave correctly.

Appendix C has an overview of all the files used in this project, including the hardware design, test benches and the testing program.

1. init
2. **while** 150 != devsteps:
3. config
4. run(77)
5. readback
6. switch
7. devstep
8. readTypes
9. readStates

Figure 7.1: Functional test program

7.2 Improvements

This section covers synthesis and performance results of the work done in chapter 4 in comparison with the results from [1, 3].

Table 7.1 shows a performance comparison between the design from [1] (old) and the new design. The design from [3] has the same performance as [1], but without the RSF slowing down the SBM running. Because it is negligible for large enough matrices, the table data does not account for setup overhead for the different entries. Apart from the RSF and *rules per set*, most values are 4 times

Design unit	Unit	Old value	New value
BRAM	bus width (types)	8	32
Development	sblocks per cycle	2	8
Development	rules per set	8	8
LUTconv	conversions per cycle	2	8
Config	sblocks per cycle	2	8
Readback	states per cycle	8	32
RSF	states per cycle	16	all
RSF	cycles per run step	64*	1
SBM	speed without RSF	1	1
Write Type	types per instruction	1	4
Write State	states per instruction	1	16

*Given a 32×32 matrix.

Table 7.1: Performance comparison for original and new design

larger for the new design. This means that the performance for most parts of the design is 4 times higher.

Figure 7.2 shows a typical program for design. For a 32×32 matrix the typical program would be about 3 times faster using the new design. This does not take into account the difference of clock speed between the old and new hardware, 80 MHz vs 125 MHz. For typical use, the new design on the new hardware is about

1. init
2. **while** 100 != devsteps:
3. config
4. run(150)
5. readback
6. switch
7. devstep - 8 rules
8. readTypes
9. readStates

Figure 7.2: A typical program

4.7 times faster than the old design on the old hardware. When adding RSF to the equation, the new design would be almost 40 times as fast as the old one.

Aamodt [1] and Djupdal [3] used a test program like the one in figure 7.1, but with 10000 development steps and 50000 run steps per development step, for testing the speed of their designs compared to a simulation. For an 8×8 matrix, Djupdal [3] ended up with 6.3 seconds, Aamodt [1] used 31.3 seconds due to the RSF, and the simulation used 18 minutes. In simulations the test takes 6.3 seconds again with the new design. For this test there is no improvement over [3] due to running the SBM taking about 99% of the time.

SBM size	*Slice LUTs*		*Slice Registers*		*BRAMs*	
	old	new	old	new	old	new
8×8	3884	-	5193	-	27	-
16×8	4151	10167	5288	13054	27	36
32×8	4823	10807	5455	13556	27	37
64×8	5700	12098	5766	14441	27	38
128×8	7371	14810	6362	16267	27	41
256×8	-	20741	-	19833	-	45
16×16	4681	10900	5456	13552	27	37
32×32	7563	14858	6363	16259	27	38
64×64	-	31098	-	26937	-	54

Table 7.2: Comparison of synthesis results for original and new design

Table 7.2 shows the resource usage of the new design compared to the old design when they are both synthesised for the Spartan6 FPGA. The cells without numbers represent designs that could not be synthesised due to certain design limitations. The new design uses about 2.5 times as many LUTs as the old one for a small configuration, and about 2 times as many for the larger configurations. So for a 32×32 matrix we see an approximately 3 times performance gain for an overall doubling in resource usage.

No parts of the old design scales with the size of the SBM. Thus, only the fact

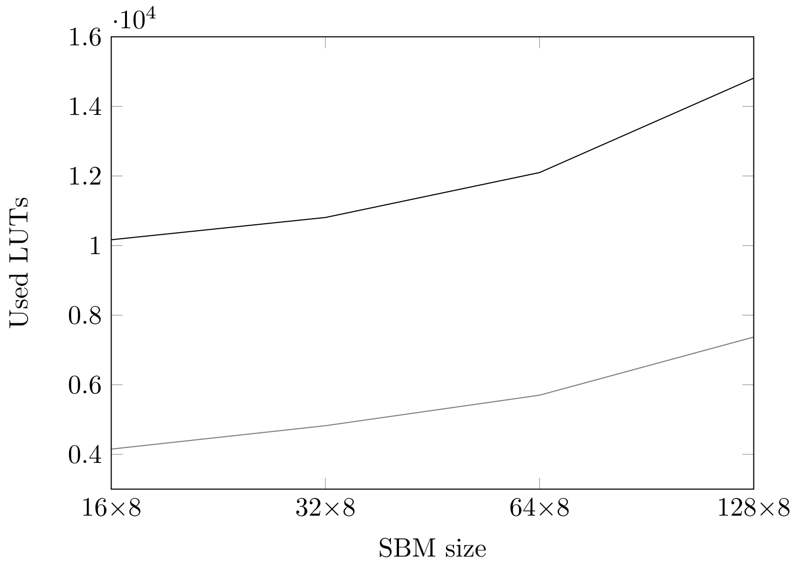

Figure 7.3: FPGA LUT usage given array size for original and new design

that the SBM becomes bigger increases the resource usage. For the new design however, the size of the BRAM units as well as the size of the RSF adder tree change with the size of the matrix. The scaling of the adder tree is reflected in the higher increase in LUT usage for the new design than for the old design. This difference can also be seen in figure 7.3, which plots the LUT usage for the configurations 16×8, 32×8, 64×8, and 128×8 for the old and the new design. The plot for the new design has a somewhat steeper curve.

7.3 DFT

This section covers the testing and results of the DFT unit described in chapter 6. It starts with functional testing before moving on to synthesis results and performance.

The DFT unit has been tested with its own simulation test bench. It implements a BRAM module with example input data, like the RS BRAM in the full design. The test bench starts the DFT unit and waits for it to finish. When it does, the test bench compares the DFT unit's output with the correct DFT. The test bench can be configured to put the input data anywhere in the BRAM. It will tell the DFT unit the first address of the data, and the DFT unit should start reading input from that address.

Table 7.3 shows synthesis results for the design with DFT implemented. Results for configurations that use 32, 64, and 128 DSP slices to compute the DFT of a 128 value input sequence are shown for a variety of SBM sizes.

In general, the more DSP slices used, the faster the DFT unit, at the cost of

DSPs:	32	64	128	32	64	128	32	64	128
SBM size	*Slice LUTs*			*Slice Registers*			*BRAMs*		
16×8	10985	11355	12027	13598	13595	13594	46	54	70
32×8	11974	12236	12994	14151	14141	14140	47	55	71
64×8	13337	13737	14264	15138	15135	15135	47	55	71
128×8	16158	16660	17218	17009	17006	17006	48	56	72
256×8	21940	22533	23005	20646	20643	20643	48	56	72
16×16	12010	12226	12911	14140	14137	14136	47	55	71
32×32	16188	16534	17140	17001	16998	16998	48	56	72
64×64	33050	-	-	27800	-	-	49	-	-

Table 7.3: Synthesis results with DFTs using different amounts of DSPs

more LUTs and BRAM. There is no advantage using more DSPs than the length of the input sequence. The 128 DSP configuration spend 128 cycles computing the DFT, 64 DSPs spend 256 cycles, and 32 DSPs spend 512 cycles.

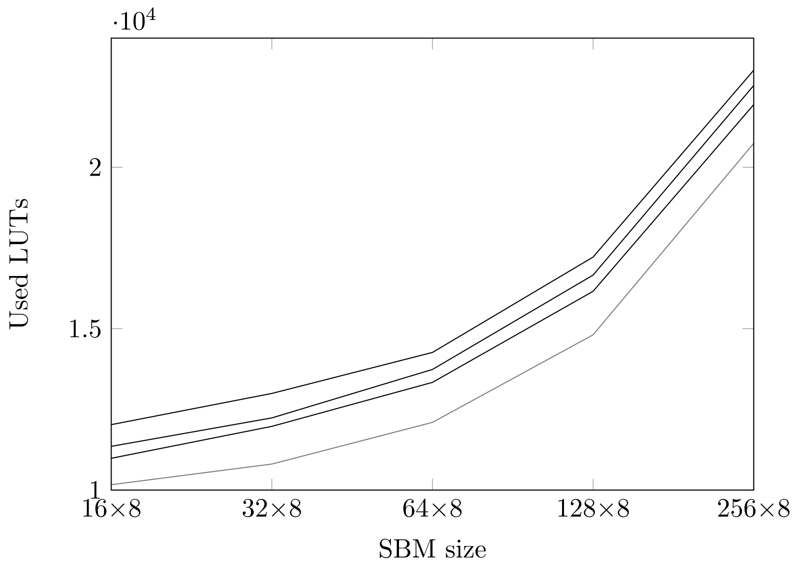

Figure 7.4: FPGA LUT usage given array size for design with and without DFT

A comparison of the three DFT configurations with the results from the previous section, without DFT, is shown in figure 7.4. The 32 DSP configuration (lower black curve) is about halfway between the design without DFT (grey curve) and the 128 DSP configuration (upper black curve). This means that the 128 DSP DFT unit uses almost twice the amount of LUTs as the 32 DSP one. The amount of registers needed however, is almost the same for both configurations.

The higher amount of BRAM modules needed for the larger DFT unit is due to the fact that it needs to read more twiddle factors each cycle, leading to a larger

memory bandwidth requirement. The design is still far away from using all 268 of the BRAM modules on the FPGA.

The place and route part of building the design fails for the 64×64 configuration with 64 or more DSPs. Thus, the configuration with 32 DSPs for that SBM size is at the edge of what is achievable while keeping performance of the rest of the design up. This is the case even though only 35% of available LUTs are used and 15% of the registers are used, which shows that the interconnect resource demand of the design is very high compared to the logic resource demand.

Using the typical program from figure 7.2 again, but with the addition of computing the DFT, about 23% of the time is spent computing the DFT for the 128 DSP configuration. For the 64 DSP configuration this number is 37%, and for the 32 DSP configuration it is 54%. If there is room for the larger DFT configuration on the FPGA it should be prioritised.

7.4 3D SBM

Correctness testing of the 3D design was done using the same functional test as for the 2D design, the test shown in figure 7.1. The development rules and sblock LUTs are extended to 3D rules and LUTs in a way that they still only work in a 2D plane. Sets of rules and LUTs are made for the x-y plane, the x-z plane, and the y-z plane. In this way, development of all cells in relation to each of their neighbours can be tested, and the interconnect between all the sblocks in the SBM can be tested. All testing produced correct results in the end.

The optimistic goal when extending the design to support a 3D SBM was to implement a 16×16×16 SBM *and* scale the other parts of the design to keep the performance up. This quickly turned out to be unfeasible when the synthesis tool claimed the design would use 140% of the available LUTs on the FPGA. The config unit was to blame, using 75% of the available LUTs on its own. After changing the config unit and scaling down other parts of the design, as described in chapter 5, synthesis was possible.

The synthesis results shown in table 7.5 were achieved with the performances per unit shown in table 7.4. The comparison field is a comparison between the

Design Unit	*Value*	*unit*	*Comparison*
BRAM	SBM_WIDTH*8	types/states bus width	4×
Development	SBM_WIDTH/2	sblocks per cycle	1×
Development	4	rules per set	1/2×
Config	SBM_WIDTH/8	sblocks per cycle	1/4×
Readback	SBM_WIDTH*8	states per cycle	4×
RSF	SBM_SIZE/4	states per cycle	1/4×
RSF	4	cycles per run step	4×

Table 7.4: Performances of the units of the 3D design, along with a comparison to the 2D design performance.

performance of a 16 wide 3D SBM and the 2D design performance. The config and run step performances are effectively reduced to a quarter, development's performance is halved, and readback's performance is quadrupled.

Again we use the program from figure 7.2 as a reference. A 16×8×8 configuration has the same amount of cells as a 32×32 configuration. The test program is about three times faster on the 2D design than the 3D design. Thus, the overall performance is reduced to about one third.

SBM size	*Slice LUTs*	*Slice Registers*	*BRAMs*
8×8×4	6529	6011	55
8×8×8	7668	5726	50
8×16×4	8234	6531	55
8×16×16	18003	8466	51
8×64×4	17798	9219	57
16×4×4	9045	8156	65
16×8×8	14097	9613	66
16×8×16	19720	10516	62
16×16×8	19414	10506	62
16×16×16	37313	14956	68

Table 7.5: Synthesis results for the 3D design

Scaling the SBM width scales other parts of the design as well, like the development, config, and readback unit. This is reflected in the synthesis results. The results for a 16×16×8 matrix is almost equal to those for a 16×8×16 matrix. However, an 8×16×16 matrix uses about 1500 less LUTs and about 2000 less registers.

The largest synthesisable 3D design is a 16×16×16 SBM with the performance described above. The tools could however not make the design fit the FPGA after adding the DFT unit. Scaling down to a 16×16×8 SBM almost halves the LUT resources used, lets the design fit on the chip, and leaves some wiggle room for implementing PCIe communication and more interesting fitness functions later.

Chapter 8

Discussion

Correctness testing validates both the 2D and 3D designs in simulation. The functional tests yield the same result as they yielded for Djupdal [3] and Aamodt [1], meaning the functionality implemented by them still works.

No top-level test for testing the DFT unit specifically has been made. It has however been tested with its own test bench, which shows that it produces correct results. No new tests have been specifically designed for the 3D design either, but it has been tested by running the 2D test in all three orientations, so it should work with 3D rules as well.

Performance results for the 2D design show that the speed has increased considerably over [3], and even more over [1]. When compared to [1], the main contributor is the new adder tree in the RSF, which makes run steps 64 times faster on a 32×32 matrix. The increase in performance lets the user run more and/or larger experiments in the same time as a smaller experiment used before. This is very good for running GAs as they tend to be very time consuming.

For the 3D design the performance had to be scaled down considerably to fit the target 16×16×16 SBM on the device, leaving it at about the same level as [3]. Even with the downscaling there is not much room left for implementing required and/or new features as PCIe communication, DFT or interesting fitness functions. For this to be possible, lowering the SBM size to 16×16×8 or making further design optimisation is necessary.

Performance and synthesis results for the DFT unit show that a quadrupling of DSP slices used leads to a quadrupling in performance for that unit, but only a doubling in LUTs used by that unit. For this reason, using as many DSPs as there are numbers in the input sequence is recommended for most 2D designs. However, for the 3D design, LUT resources might be better spent to increase performance of some other part of the design, sacrificing some of the performance in the DFT unit.

In general, synthesis results show that smaller SBM configurations use considerably less resources. These extra resources could be used to further increase the performance of several parts of the design. For example, a 16×16 matrix configuration could in theory develop and configure more sblocks at the same time

than a 32×32 matrix configuration could.

8.1 Implementation

Section 4.3 describes how many parts of the design have been parameterised. Most of these parameterisations are partial, meaning that changing the variables will make the design unsynthesisable unless more hardware code is changed. For example, trying to increase the amount of concurrent cell developments or sblock configurations for the 2D design cannot be done by just changing the parameters, because the relevant units are not fully parameterised. Increasing the performance of those units has however been made simpler due to the partial parameterisation; not as much code needs to be changed as before. For most partially parameterised units, the part that is not yet parameterised has to do with memory access.

The work on the 3D implementation resulted in several optimisations that were not later pulled into the 2D design. For example, the configuration unit was optimised in a way that needed less LUT resources but more BRAMs to achieve the same performance as before. The amount of unused BRAM modules is very high for the 2D design, so implementing the optimisations there as well would not be a problem.

As in [1, 3], only one instruction can be executed at a time. This is to keep the design simple and easy to further work on and debug. By executing several instructions concurrently some units in the design could run at the same time without changing the functional behaviour. For example, the DFT unit could run at the same time as the SBM run steps, as long as it waits for data generated by the RSF to be available before trying to read them. This would make the design more efficient by increased parallelity, but it would also make it more convoluted and prone to errors.

It is not clear that a 3D SBM can produce more interesting or better results than a 2D SBM, or that the DFT is a well suited transform for use in fitness calculations. However, the work done in this thesis should make further research into the subjects more efficient by providing a hardware platform to run experiments on.

8.2 Future Work

This thesis was written in anticipation of a new PCIe FPGA development board. The new hardware did not arrive by the end of this work. As a consequence, the development board specific parts of the hardware design and host computer software have not been implemented in this thesis. The first thing that needs to be done to get this design working in hardware is to implement the PCIe communication in both the FPGA design and the host computer software.

The lack of hardware also means that all functionality testing has been done in relatively slow simulations. With a full hardware implementation with working PCIe communication, more comprehensive top-level testing can be done at full

speed in hardware. Top-level tests designed to specifically test the new features added in this thesis could also be made.

By fully parameterising most parts of the design, the user could more easily change the performance of the different parts to optimise for the typical program that he or she will run on the platform. If the user uses very many development rules, more resources should be devoted to the development unit by reducing the performance of other units in the design.

Optimisations from the 3D design could be pulled into the 2D design. They could be made much more similar or even combined. The instruction format and development rule format have already been unified for the two designs. If combined well, a 2D SBM would simply be configured as a single layer 3D SBM without *up* and *down* in its neighbourhood.

Djupdal [3] and Aamodt [1] also suggest some future work that is still valid:

- Extend the design to support a Turing complete instruction set.
- Change the RSF and fitness function to fit the needs of the user.
- Currently the size of the SBM can only be a power of two in each direction. The design could be changed to support other sizes.

The last point is valid for almost all aspects of the design. Configurations per cycle, cells developed per cycle, DSPs used by the DFT unit, and many more have to be powers of two. Changing this could let the user more finely tune the design performance and functionality.

Bibliography

[1] Kjetil Aamodt. Kunstig utvikling: Utvidelse av fpga-basert sblock-plattform. Master's thesis, Norwegian University of Science and Technology, 2005.

[2] Sivert Berg. Evolution of cellular automata using lindenmayer systems and fourier transforms. Master's thesis, 2013.

[3] Asbjørn Djupdal. Konstruksjon av maskinvare for kjøring av sblokkbaserte eksperimenter. Master's thesis, Norwegian University of Science and Technology, 2003.

[4] Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field programmable gate arrays*, volume 180. Springer, 1992.

[5] David Edward Goldberg et al. *Genetic algorithms in search, optimization, and machine learning*, volume 412.

[6] Mentor Graphics. Modelsim. URL `http://www.mentor.com/products/fpga/model`.

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

[8] Tetsuya Higuchi, Masaya Iwata, Isamu Kajitani, Hitoshi Iba, Yuji Hirao, Tatsumi Furuya, and Bernard Manderick. Evolvable hardware and its application to pattern recognition and fault-tolerant systems. In *Towards evolvable hardware*, pages 118–135. Springer, 1996.

[9] Tetsuya Higuchi, Yong Liu, and Xin Yao. *Evolvable hardware*, volume 10. Springer, 2006.

[10] Paul Layzell. Reducing hardware evolution's dependency on fpgas. In *Microelectronics for Neural, Fuzzy and Bio-Inspired Systems, 1999. MicroNeuro'99. Proceedings of the Seventh International Conference on*, pages 171–178. IEEE, 1999.

[11] Moshe Sipper. *Evolution of parallel cellular machines*, volume 4. Springer Heidelberg, 1997.

[12] Tommaso Toffoli and Norman Margolus. *Cellular automata machines: a new environment for modeling.* MIT press, 1987.

[13] Gunnar Tufte and Pauline C. Haddow. Building knowledge into developmental rules for circuit design. In *Evolvable Systems: From Biology to Hardware*, pages 69–80. Springer, 2003.

[14] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of modern physics*, 55(3):601, 1983.

[15] Xilinx. Xilinx ise, . URL `http://www.xilinx.com/content/xilinx/en/products/design-tools/ise-design-suite/`.

[16] Xilinx. Xilinx ug380 - spartan-6 fpga configuration user guide, . URL `http://www.xilinx.com/support/documentation/user_guides/ug380.pdf`.

[17] Xilinx. Xilinx ds160 - spartan-6 family overview, . URL `http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf`.

[18] Xilinx. Xilinx ds022 - virtextm-e 1.8 v field programmable gate arrays, . URL `http://www.xilinx.com/support/documentation/data_sheets/ds022.pdf`.

[19] Xin Yao and Tetsuya Higuchi. Promises and challenges of evolvable hardware. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 29(1):87–97, 1999.

Appendix A

Instruction Manual

This appendix is an updated version of appendix A in [1].

All instructions have the following format:

operands	size	opcode
n-8	7-6	5-0

- *opcode*; Unique id for each instruction, increased from 5 to 6 bit.
- *size*; Defines the size of the instruction.

size	instruction size
00	64
01	128
10	192
11	256

  Size used to be 1 bit, indicating a 64-bit instruction if set, otherwise 32-bit. Instructions as long as 256 bit are not currently used.

- *operands*; instruction specific data

break

Stops running of instructions from BRAM. Used to end a program and start accepting instructions over PCI.

unused	00	001101
63-8	7-6	5-0

clearBRAM

Sets all sblock data stored in BRAM0 to a given state and type value.

sate	unused	type	unused	00	010011
63	62-40	39-32	31-8	7-6	5-0

- *type*; The type all sblocks are set to have
- *state*; The state all sblocks are set to have

config

Configures the SBM with data from BRAM1 and LUTconv.

unused	00	000111
63-8	7-6	5-0

devstep

Runs a development step. Data is read from BRAM0 and the result is written to BRAM1.

unused	00	001010
63-8	7-6	5-0

doFitness

Runs the fitness function.

configurable	00	011010
63-8	7-6	5-0

- *configurable*; The fitness instruction format is dependent on the implemented fitness function

end

Stops the storing of instructions to BRAM. This instruction is not stored.

unused	00	001111
63-8	7-6	5-0

jump

Start executing instructions from BRAM at a given address.

unused	address	00	001100
63-24	23-8	7-6	5-0

- *address*; Jump to or start executing from this address in the BRAM

jumpEqual

Jumps to a given address in BRAM if the amount of run development steps are equal to a given value.

value	unused	address	00	010110
63-48	47-24	23-8	7-6	5-0

- *address*; The address to jump to
- *value*; The value used in the comparison

nop

Does nothing.

unused	00	000000
63-8	7-6	5-0

readback

Read back all state data from the SBM to BRAM1.

unused	00	001000
63-8	7-6	5-0

readFitness

Send fitness data back to host.

unused	00	011001
63-8	7-6	5-0

readRuleVector

Send to host a number of vectors describing what rules have been activated for the development steps.

unused	n	00	011000
63-24	23-8	7-6	5-0

- n; The number of vectors to send

readState

Send to host a single sblock's state from BRAM0.

unused	z	y	x	00	000101
63-32	31-24	23-16	15-8	7-6	5-0

- x; The sblocks x-position
- y; The sblocks y-position
- z; The sblocks z-position, used only for 3D

readStates

Send to host all sblocks' states from BRAM0.

unused	00	010001
63-8	7-6	5-0

readType

Send to host a single sblocks type from BRAM0.

unused	z	y	x	00	000010
63-32	31-24	23-16	15-8	7-6	5-0

- x; The sblocks x-position
- y; The sblocks y-position
- z; The sblocks z-position, used only for 3D

readTypes

Send to host all sblocks' types from BRAM0.

unused	00	010000
63-8	7-6	5-0

readSums

Send to host a number of results generated by the RSF.

unused	n	00	010100
63-24	23-8	7-6	5-0

- n; The number of values to be sent to host

resetDevCounter

Resets, to zero, the counter that counts the number of development steps that have been run. Relevant for jumpEq.

unused	00	010111
63-8	7-6	5-0

run

Run the SBM for a given number of run steps.

cycles	00	001001
63-8	7-6	5-0

- *cycles*; The number of run steps the SBM should be run for

readUsedRules

Send to host what rules were activated the previous development step.

unused	00	010101
63-8	7-6	5-0

setNumberOfLastRule

Set the amount of rules are to be used in development. All rules up to and including the one stored at the specified position in the rule BRAM are used.

unused	number	00	010010
63-16	15-8	7-6	5-0

- *number*; The highest priority rule in the rule BRAM

startDFT

Start the DFT unit.

unused	address	unused	00	111000
63-48	47-32	31-8	7-6	5-0

- *address*; The first address in the RS BRAM the DFT unit should fetch data from

store

All following instructions are to be stored in the BRAM, until an *end* instruction is received.

unused	address	00	001110
63-24	23-8	7-6	5-0

- *address*; The first address in BRAM the program is stored to

switch

Switches the contents of BRAM0 and BRAM1.

unused	00	000011
63-8	7-6	5-0

writeLUTConv

Writes a LUT to the LUT conversion BRAM. For a 2D design instruction, the LUT is in 95-64, and a 128-bit instruction is enough.

lut	unused	type	unused	10	000110
191-64	63-40	39-32	31-8	7-6	5-0

- *type*; The address at which to store the LUT

- *lut*; The LUTto be stored, for a 2D design it is only 32 bits

writeRule

Writes a development rule to the rule storage.

rule	unused	number	01	001011
127-39	38-16	15-8	7-6	5-0

- *rule*; The rule that is stored. Its format is detailed in appendix B
- *number*; The address in the rule storage the rule is stored. Rules stored at higher addresses have a higher priority.

writeState

Writes a single sblock's state to BRAM0.

state	unused	z	y	x	00	000100
63	62-32	31-24	23-16	15-8	7-6	5-0

- *x*; The sblocks x-position
- *y*; The sblocks y-position
- *z*; The sblocks z-position, used only for 3D
- *state*; The state to be written

writeStates

Writes one word of states to BRAM0 for each BRAM module. For 2D this is 4 words with a total of 16 states. For 3D this is 8 words with a total SBM_WIDTH*4 states.

states	unused	z	y	x	01	100100
N-64	63-32	31-24	23-16	15-8	7-6	5-0

- *x, y and z*; Indicates the position of an sblock, the address of that sblock in BRAM0 has entries in all the modules that make up BRAM0. State data is stored at that address in each of them.
- *states*; The states that are written to BRAM0. The most significant states are written to the most highest numbered BRAM module

writeType

Writes a single sblock's type to BRAM0.

unused	type	z	y	x	00	000001
63-40	39-32	31-24	23-16	15-8	7-6	5-0

- *x*; The sblocks x-position

- *y*; The sblocks y-position
- *z*; The sblocks z-position, used only for 3D
- *type*; The type to be written

writeTypes

Writes one word of types to BRAM0. For 2D this is 4 types. For 3D this is SBM_WIDTH/2 types.

types	unused	z	y	x	01	100001
N-64	63-32	31-24	23-16	15-8	7-6	5-0

- *x, y and z*; Indicates the position of an sblock, the word containing that sblock in BRAM0 is written to
- *types*; The types that are written to BRAM0.

Appendix B

Rule Format

Each rule has the following fields:

valid	type	up	down	north	south	east	west	centre	result
88	87	86-76	75-65	64-54	53-43	42-32	31-21	20-10	9-0

- *valid*; Whether or not the rule is valid. If it is 0, not valid, it is not checked by the development unit.
- *type*; Not used in this thesis' implementations, however could still be reimplemented. Used to differentiate between change (0) and growth (1) rules.
- *up,down,north,south,east,west¢re*; Makes up the rule *condition.* There is an 11 bit condition on each sblock in the neighbourhood. Up and down are only used for 3D rules.
- *result*; What happens if the rule is a hit.

Condition

The condition on each sblock in the neighbourhood is encoded like this:

ignore state	state	ignore type	type
10	9	8	7-0

- *ignore state*; Whether or not to take this sblock's state into account when checking the rule.
- *state*; The state the sblock must have for the rule to apply.
- *ignore type*; Whether or not to take this sblock's type into account when checking the rule.
- *type*; The type the sblock must have for the rule to apply.

Result

The result field is encoded as follows:

no state change	new state	new type
9	8	7-0

- *no state change*; Whether or not this rule changes the state of the sblock.
- *new state*; If this rule does change the state of the sblock, this field contains the new state, otherwise it is ignored.
- *new type*; The sblock's type is changed to this value.

In [1; 3] the result differentiated between growth and change rules, in this thesis they were combined to a more general change rule.

Appendix C

Attached Files

C.1 File Hierarchy

The files used in this thesis project is attached as a .zip file. The folder hierarchy in the .zip file is as follows:

- hardware
  - 2D
  - 3D
  - test_benches
  - synthesis_results
    - 2D
      - 16x8
      - 16x16
      - ...
    - 3D
      - 8x8x4
      - 8x8x8
      - ...
    - ...
- software
  - 2D
  - 3D
  - original
  - output

The *2D* and *3D* folders in the hardware folder contain the hardware code for the 2D and 3D designs respectively. The files are detailed in section C.2.

The *test_benches* folder contain the two simulation test benches used when testing the designs. They are detailed in section C.3

The *synthesis_results* folder contain several folders, each of which contain a series of folders containing synthesis results. See section C.4.

The *software* folder contain testing software written in c, as well as expected output when running tests. Section C.5 covers this.

C.2 Hardware Design Files

The hardware design is written in VHSIC hardware description language (VHDL). There is also a python script that is used to generate the twiddle factor memory.

Files that are new in this thesis are marked with a *. Almost all other files has also been changed during this thesis' work. There are two complete sets of hardware design files, one for 2D and one for 3D. Both of them contain the same files:

- *addr_gen.vhd*: Used to generate BRAM0 and BRAM1 addresses based on the x, y and z coordinates for an sblock.
- *bitcounter8.vhd,bitcounter4.vhd*: bitcounterN uses bitcounter8 and bitcounter4 at the bottom to sum 8 and 8 input signals.
- *bitcounterN.vhd**: Sums N 1-bit input signals. Uses recursion to build an adder tree with pipeline registers.
- *bram_inferrer.vhd**: Implements a general BRAM module. Used to infer BRAM modules on the FPGA.
- *com40.vhd*: Communication module for PCI communication, should be replaced with a PCIe communication module.
- *counter.vhd*: Used throughout the design as a simple counter.
- *decode.vhd*: Responsible for decoding instructions and initiate the execution of them. Has a program counter.
- *decode_and_or.vhd**: Decodes a set of 8 bit signals to 256 bit ones and *or*s the result.
- *dev.vhd*: The development unit.
- *dft.vhd**: The DFT unit.
- *fetch.vhd*: Unit responsible for fetching instructions from PCI and the instruction memory, it also stores instructions in the memory.
- *fitness.vhd*: Wrapper for the fitness function, receives data from the DFT unit and RSF.
- *fitness_funk.vhd*: The fitness function currently implemented.
- *fitness_reg.vhd*: Register for storing fitness calculation results.
- *funct_package.vhd*: contain a few helpful functions.
- *hazard.vhd*: Responsible for stalling other units when some are busy.
- *instrmem.vhd*: Memory used for storing instructions.
- *lss.vhd*: Responsible for execution of all instructions that store or read data from the BRAM0.
- *lutconv.vhd*: Memory used for implementing the type to LUT conversion table.
- *Makefile*: Is used to synthesise the design.

- *package.vhd.in*: Package with component declarations and constants that can be changed to weak the workings of the design. The file *package.vhd* is generated from this file as a first step in the make process.
- *rule_exec.vhd*: Checks an sblock for a development rule. Used by the development unit.
- *rule_select.vhd*: Selects a result from a set of rules checked for an sblock. Used by the development unit.
- *rule_storage.vhd*: Memory for storing development rules.
- *rulevector_mem.vhd*: Memory for storing the rulevector generated at development
- *run_step_funk.vhd*: Implements the RSF, currently summing all the states of the SBM at each run-step.
- *runstep_mem.vhd*: Memory for storing data generated by the RSF.
- *sblock.vhd*: A single sblock.
- *sblock_matrix.ucf*: Contains user constraints for the design, should be changed as it is highly development board dependent.
- *sblock_matrix.vhd*: links many sblocks together to make an sblock matrix.
- *sbm_bram.vhd*: The module used for implementing BRAM-A and BRAM-B in the BRAM manager.
- *sbm_bram_mgr.vhd*: Manages BRAM-A and BRAM-B used for storing the configuration data for the SBM.
- *sbm_pipe.vhd*: Implements the Config unit and other SBM related pipelines.
- *srl_inferer.vhd**: Used to implement shift registers as configurable LUTs in the sblocks.
- *toplevel.vhd*: is the toplevel part of the design. It links many components together and defines the signals that go in and out of the FPGA.
- *twgen.py**: Python script executed by the Makefile to generate *twiddle.vhd* which is a file containing all the twiddle factors needed. The script uses constants from *package.vhd* to determine the twiddle factors and the layout of the file.
- *twmem.vhd**: Constant memory used for keeping twiddle factors for the DFT unit.
- *usedrules_mem.vhd*: Memory for storing data on what development rules were activated for each sblock.
- *word_select.vhd*: A shift register used to activate a set of enable signals one by one.

Synthesising

Before synthesising, make sure the correct constant values are set in *package.vhd.in*.

Synthetisation uses XILINX ISE, and is streamlined using GNU make. The Makefile does some preprocessing using python and m4. To synthesise, navigate to the folder with the design in a terminal and run the following command:

```
$ make
```

This will generate the *package.vhd* and *twiddle.vhd* files before running all the commands necessary to run the whole synthesis process and build a bit file for programming the target FPGA with.

C.3 Test Benches

There are two simulation test benches that are used for simulation testing:

- *toplevel_tb.vhd*: A toplevel test bench for testing the toplevel functionality of the hardware designs. It simulates some of the input signals to the toplevel module. The rest, such as PCI data, must be set using simulation scripts during simulation.
- *tb_dft.vhd*: A test bench for testing the correctness of the DFT unit.

C.4 Synthesis Results

The synthesis_results folder contain synthesis results for the original design, the 2D design with and without DFT and the 3D design with DFT. There are results for using 32, 64, and 128 DSPs in the DFT unit for the 2D design, while the 3D design is only synthesised with 32 DSPs. There is one folder for each configuration, each of which contain several folders with synthesis results for different SBM sizes.

For the 2D designs, these folders have names like 16x16 or 64x8, the first number indicate the width of the SBM and the second number the height. For 3D designs, the number of layers is added to the naming convention, resulting in names like 16x16x8, where 8 is the number of layers.

Each such folder has three files:

- *package.vhd*: The configuration file containing all the constants used in the hardware design, with the values used for this synthesis.
- *tmp.srp*: The first report produced during synthesis, xst. Contains resource requirement and performance estimates.
- *sblock_matrix.par*: Results of the place and route part of synthesis. This file denotes whether or not all constraints were met and whether or not the design was fully implemented.

C.5 Software

The *original* folder contains the test software used in [1]. The files contained in it is:

- *Makefile*: File that compiles the testing program.
- *read_print.c*: C file with functions for reading output data from the FPGA over PCI and printing them with pretty formatting.
- *read_print.h*: Header file for read_print.
- *rules.c*: C file for generating development rules.
- *rules.h*: Header file for rules.
- *sblocklib.c*: C file for the library used to communicate with the FPGA over PCI. It contains functions for generating all the instructions.
- *sblocklib.h*: Header file for sblocklib
- *sblocktest.c*: C file for the testing program. Contains the main method for running tests. Also supports generating simulation scripts that can later be used for testing in a simulation.
- *types.h*: General types and constants. Including the LUTs the sblocks are configured with during testing.

The *2D* folder contains the same test software, but stripped down to only make simulation scripts. Most other code will have to be rewritten for PCIe communication. It is also updated to match the new instruction format and the new instructions have been added and taken advantage of.

The *3D* folder contains the same as the *2D* folder, but with instructions, development rules and sblock LUTs updated to work with the 3D design.

The *output* folder contains expected output for the different test programs for a series of sblock matrix sizes.

Compiling and Running

Before compiling the test program, these variables must be exported or set in the Makefile:

- COORD_SIZE_X: The size of the SBM in the x-direction is defined by $2^{COORD_SIZE_X}$.
- COORD_SIZE_Y: The same for the size in the y-direction.
- COORD_SIZE_Z: The same for the size in the z-direction. Only for the 3D test program.

Compiling the test program is done by running the command in the folder with the code:

```
$ make
```

This requires a unix system with GNU make installed.

The program can then be run using the following command:

```
$ ./sblocktest <test> <simulationscript>
```

- <*test*>: The number of the test to run.
  - *0*: Functional test
  - *1*: Speed test
  - *2*: Rule test
  - *3*: Fitness test
  - *4*: Fitness speed test
- <*simulationscript*>: The name of the file the simulation script should be saved to.

Appendix D

Software

Synthesis of hardware designs have been done on a computer running Ubuntu 12.04.4 LTS (GNU/Linux 3.5.0-47-generic x86_64). Simulations have been done on a windows server loaded with ModelSim.

- *Xilinx ISE version 14.5*: Follows these steps to synthesise a hardware design:
  1. *xst*; translates the code to Xilinx specific code and does a lot of optimisations based on knowledge of the target hardware platform.
  2. *map*; maps the different parts of the design to the different hardware modules that exist on the target hardware.
  3. *par*; place and route, places the modules described by *map* in specific positions on the target hardware and routes the interconnect between them. This process will try many different placements to try and satisfy timing constraints, like the system clock rate, while making routing between all the modules possible.
  4. *bitgen*; produces a configuration file for the target hardware.
- *ModelSim*: Simulation software used to simulate hardware designs for testing purposes before hardware implementation.
- *Python 2*: For running python code.
- *GNU C compiler*: For compiling C-code.
- *GNU Make*: For controlling the synthetisation of the hardware design and compilation of the testing software.