NTNU – Trondheim
Norwegian University of
Science and Technology

Computational Materials: Experimental Platform

Jon Emil Jahren

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Gunnar Tufte, IDI

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

Problem Description

In this research project we try to exploit computational properties of unconventional materials (materials usually not considered as a computational substrate). Such materials may offer computation at extreme low cost and may also enable us to do computation that is hard (or impossible) on a von Neumann stored program machine. Currently we explore possible computational properties of carbon nano tubes.

In 2010 a first version of a platform was made. This system consists of a PCB, including an Atmel microcontroller and a Xilinx FPGA, that acts as an interface between a PC and a micro electrode array. The array interfaces the material under investigation.

In this project the experimental platform will be extended. There are several possible directions. As such there is a possibility for several students pursuing different directions. Possible directions:

a) Extending the software, microcontroller and PC (mostly c-programming).

b) Extending the FPGA interface to the material (VHDL and c-programming).

c) Design of additional interface circuits between the FPGA and the micro electrode array (PCB-design, digital/analogue design).

Abstract

Evolution in materio aim at revealing computational properties in materials. A bottom-up approach combining methods and theories from evolutionary algorithms, complex systems and unconventional computation, is used to explore inherent computational properties of alternative computational materials. This master's thesis build on earlier and ongoing research at NTNU focusing on interfacing materials. Mecobo is a prototype interface platform, including a physical interface, hardware for stimuli/measurements and software, developed at NTNU.

The Mecobo platform is intended to serve as a platform for experimenting with unconventional materials' behavior in a dynamic complex system. This project describes the existing Mecobo platform, modifications to it and the new extension Cellular Automata Central Processing Unit (CA CPU). With the new extension, Mecobo is now capable of modelling dynamic complex systems with a cellular automaton and an unconventional material.

The Cellular Automata Central Processing Unit extension is a parallel processor with 32 cores, but supports up to and including 64 cores, for controlling each material configuration pin. The core is implemented as a stack machine optimized for instruction space. It uses an 8-bit reduced instruction set computing (RISC) instruction set architecture (ISA).

The modified Mecobo platform is tested and confirmed to work with several cellular automata rules and a simple model consisting of a theoretical material with constant response.

Sammendrag

Evolusjon i materio har som mål å utforske komputasjonelle egenskaper i materialer. En bunnen-opp tilnærming som kombinerer metoder og teorier fra evolusjonære algoritmer, komplekse systemer og ukonvensjonell komputasjon, blir brukt for å utforske inneboende komputasjonelle egenskaper i alternative komputasjonelle materialer. Denne masteroppgaven bygger på tidligere og pågående forskning ved NTNU, som fokuserer på interaksjon med materialer. Mecobo er en prototype grensesnitt plattform, med et fysisk grensesnitt, hardware for stimuli/målinger og software, utviklet ved NTNU.

Mecobo plattformen er ment som en plattform for å gjøre forskning på ukonvensjonelle materialers oppførsel i et dynamisk komplekst system. Prosjektet beskriver den eksisterende Mecobo plattformen, endringer på den og den nye utvidelsen Cellular Automata Central Processing Unit (CA CPU). Med den nye utvidelsen, er Mecobo istand til å modellere dynamisk komplekse systemer med en cellular automaton og et ukonvensjonelt materiale.

Cellular Automata Central Processing Unit utvidelsen er en parallell prosessor med 32 kjerner, men den støtter opp til og med 64 kjerner, for å kontrollere hver material konfigurasjons pin. Kjernen er implementert som en stakk maskin, optimalisert for instruksjons størrelse. Den bruker en 8-bit reduced instruction set computing (RISC) instruction set architecture (ISA).

Den endrede Mecobo plattformen er testet og bekreftet å virke med flere cellular automata regler og en enkel modell bestående av et teoretisk materiale med konstant respons.

Acknowledgements

Thanks to my supervisor, Gunnar Tufte, for insightful and interesting ideas, discussions and criticism. And thanks to my brother Ole Henrik Jahren for his input and discussions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Evolution in Materio is a relatively new field, but the ideas date back to the 1950's and the field of cybernetics. One of the pioneers of evolution in materio is Gordon Pask, a British cybernetics researcher. Who made the famous "ear" which could distinguish two frequencies from another [1]. The primary focus in the field evolution in materio is finding and exploiting the underlying material, more common today is evolvable hardware (EHW) [9] which is largely dominated by research into evolving circuits based on known and common electronic components. In evolution in materio there is more focus on finding new unconventional materials to use in computation, like Harding and Miller's tone discriminator evolved in liquid crystal [10] or using a substance of gel and nanotubes, i.e. like in the Mecobo platform.

While the current semi-conductor industry is heavily invested into doped silicon and transistors, and struggling with area, heat and power constraints. It is desirable to find a new method or material which does not have these constraints or are less affected by them. Maybe even an inherently robust material could be found, which is one of the key properties wanted in an adaptive system.

The system described in this thesis is Mecobo. Mecobo is a platform for running experiments on different materials. It consists of software, a microcontroller, a FPGA and a material bay which interfaces with the material.

The task was to design and implement an extension which enabled the ability to experiment with evolvable material, and integrate it into the already existing Mecobo platform. The result is Cellular Automata Central Processing Unit(CA CPU) and modifications to the existing Mecobo platform.

The Mecobo platform is a platform for interfacing with a test material, the interfacing can be done by setting each input to the material manually, or by using the CA CPU(cellular automaton central processing unit) which is a parallel processor module which can be used to model a CA in conjunction with the test material, which makes it a hybrid CA materio model. The CA will guide the evolution of the material by changing the configuration or input to the material based on the material's response.

The Mecobo platform and ideas of possible research directions, are based on theory of evolution in materio and self-organization, cellular automata, genetic algorithms and complex behavior in general.

Similar to what Langton did [6] for CAs, one goal is to search for materials which can support information transmission, storage and modification to see if we can exploit the material to exhibit complex behavior and ultimately support universal computation.

Can the complete hybrid CA material system modeled on Mecobo exhibit complex behavior equivalent of a class 4 CA and able to do computations? Or perhaps the CA can evolve the material such that the material by its own can exhibit complex behavior and support universal computation?

It is also interesting to see if it is possible to find and evolve a material which can add complexity to a CA's behavior or perhaps even inhibit complex behavior, in this way the material acts as a complexity modifier.

The Mecobo platform is designed to explore these ideas and to possibly find new materials with interesting properties and behavior. Mecobo enables the user to experiment with different materials, to see how they behave. The Mecobo platform is very programmable, the CA running on the FPGA is user-defined, or a GA can evolve the CA and initial state. One can also experiment with the material directly without the use of the CA CPU module by configuring the pins to the material directly using a GA or manually. The material bay also supports different voltages(be sure that the FPGA's IO ports can handle the current and voltage).

Chapter 2 gives an introduction to the ideas, models and some of the theory behind the design of the Mecobo platform and the CA CPU extension. Chapter 3 gives an overview of the Mecobo system and general idea behind it. Chapter 4 is a description of the system and some implementation details. Chapter 5 describes each of the initial programs run on Mecobo and their results. Chapter 6 discusses the platform and results, and describes some of the technical difficulties, as well as future improvements and new ideas.

Chapter 2

Background

This chapter describes some of the necessary theory which is needed to understand how the system is suppose to operate. In [8] Lykkebø introduces some of the background and motivation of the Mecobo platform. The extension of Mecobo introduces some new elements, among them cellular automaton and self-organization to get a better understanding of how the new extension fits into the theory of evolution in materio. The conjunction of cellular automata and self-organizing material does not have a formal model tied to it yet, there are shared principles and ideas that gives valuable insight when trying to merge them.

2.1 Self-organization

One of the pioneers of self-organization which introduced self-organization to the sciences as we know them today is William Ross Ashby, he created one of the first self-organizing devices, the Homeostat in 1948, later described in [11]. Which could adapt itself to the environment. He stated that "every isolated determinate dynamic system obeying unchanging laws will develop "organisms" that are adapted to their "environments"". Which in essence means that all dynamic systems tend to evolve towards equilibrium or an attractor. Others like Gordon Pask's ear and Heinz von Förster's principle "order from noise" [12] followed after. Order from noise is the principle that the faster the system goes through its state space the faster it ends up in an attractor or equilibrium.

Some of the main motivation of evolution in materio comes from Gordon

Figure 2.1: A thread forming and splitting (Taken from Gordon Pask [1])

Pask's experiments which showed a self-organizing system based on metallic ions and electric current. Through Pask's experiments he were able to create an ear by using methods presented in [1], which could distinguish between two signals with different frequency. The main part in the system is the ferrous sulphate solution in which the metallic ions form threads which lead current. And a device able to distribute current through the electrodes creating connections. The connections may grow stronger or weaker, more or less current flowing through them. The device maximizes the current through the electrodes, but limits the overall current in the network.

The connections formed by threads can be seen in figure 2.1, the thread forms because of the voltage differences of the electrodes in the solution. Where x,y and S are electrodes with different voltage levels. The system self-organizes and the thread is building itself from S towards both x and y based on the current flowing.

The voltages is applied through the resistor network shown in figure 2.2. The slightly wavelike object in the background of figure 2.2 being the ferrous sulphate solution and infront the resistor network, part of the device distributing the current. The controlling machine which evolves the system can be seen in the top.

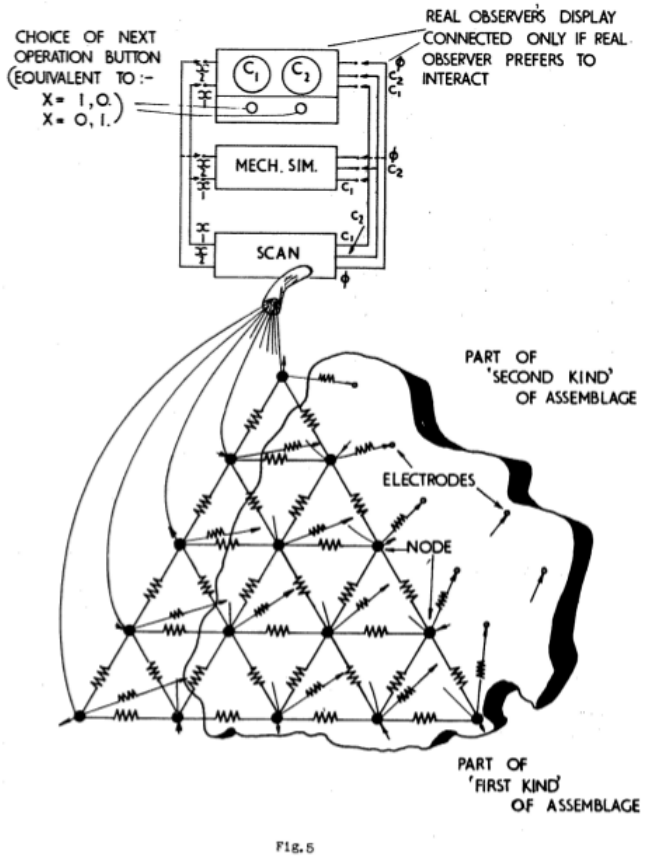


Figure 2.2: Gordon Pask's assemblage, showing the electrodes and the underlying solution (Taken from Gordon Pask [1])

2.2 Evolvable Hardware

Adrian Thompson presented in [2], 3 hypotheses and showed how evolution could exploit the underlying material. Thompson evolved among other things frequency discriminator similarly to what Pask managed to create, and was able to observe that the evolution process was able to exploit some "hidden" functionality in the FPGA. Thompson's hypotheses:

1. conventional design methods work withing constrained regions.
2. evolutionary algorithms can explore some of the regions beyond the scope of conventional methods.
3. evolutionary designs can in practice produce design that are beyond the scope of conventional methods.

The hidden properties in the FPGA was found by checking each cells contribution to the emergent behavior of the circuit shown in figure 2.3. The figure shows the circuit layout of the logical blocks in the FPGA, the grey blocks where identified as not connected and still they affect behavior of the circuit. Which means that they found a physical property of the FPGA not present in the theoretical model, and was able to exploit it.

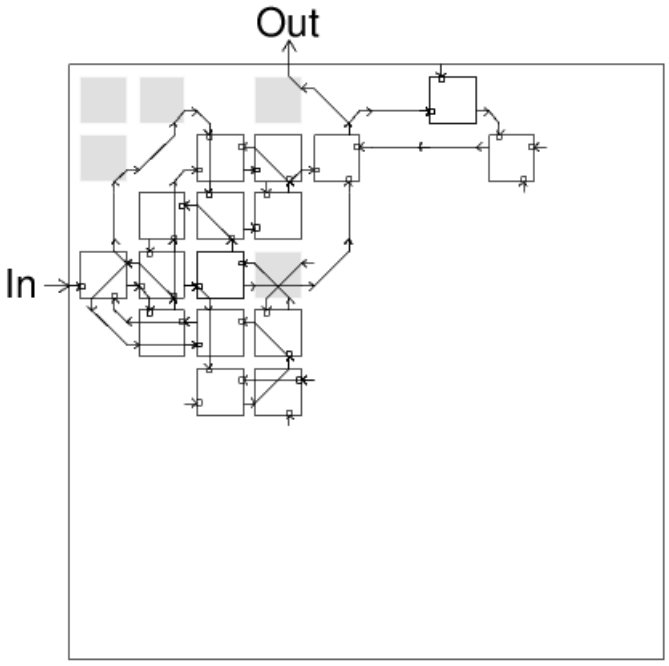


Fig. 19. The functional part of the circuit. Cells not drawn here can be clamped to constant values without affecting the circuit's behaviour.

Figure 2.3: Final circuit of one of Thompson's FPGA experiments. Gray cells are cells that affect the functionality even though their output is not used. (Taken from Thompson [2])

Similarly to what Thompson and Pask has done, Linden's antennas also shows interesting evolution of hardware and how evolution can exploit the underlying material or structure. The antenna presented by Lohn, Linden, Hornby, Kraus, Rodríguez-arroyo and Seufert in [3] perhaps being the most interesting since it got sent out to space. One of Linden's evolved antennas is shown in figure 2.4. The figure shows how the evolution has created an unconventional antenna structure with several of what seems like unstructured branches.

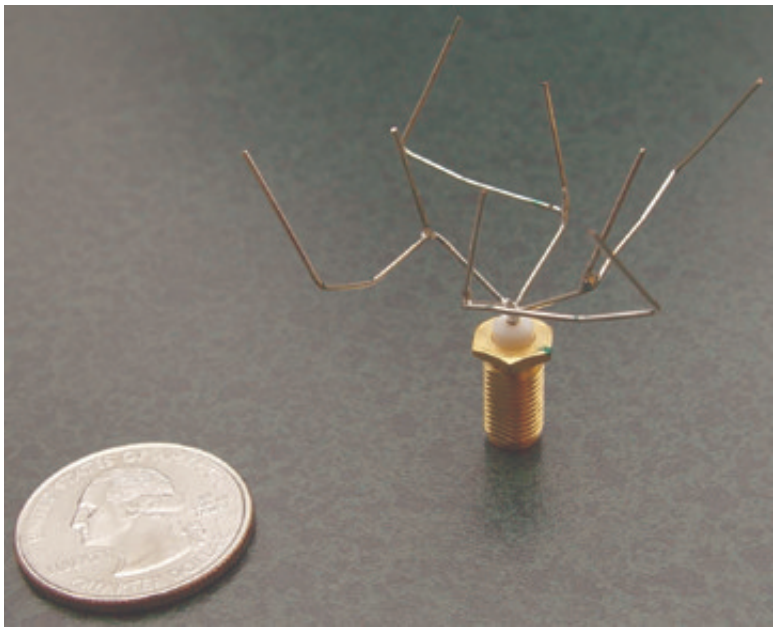


Figure 2.4: Photograph of the antenna ST5-3-10 (Taken from [3])

Thompson's or Linden's research might not be perfectly described as Evolution in Materio, it does still provide insights into how we can use evolution to find different solutions outside of our current methodologies' scope. Evolution in Materio can be described as Evolvable Hardware (EHW), but what distinguishes Evolution in Materio from "normal" EHW is the focus on use of different materials for computation, like Harding and Miller in [10] which evolved a tone discriminator in liquid crystal.

Miller and Downing in [13] describes a system called a "Configurable Analog Processor" or "CAP", which has clear similarities with the extension in our project. Miller and Downing surveys several different directions of existing technologies in Evolution in Materio among them evoultion of liquid crystal and gives interesting ideas for the future. Most systems which is described as Evolution in Materio have a similar structure as the CAP presented.

Our understanding of materials comes from several different sciences, two of the most obvious would be physics and chemistry. The knowledge we possess is encoded in many different models which we use to describe different phenomena of nature. Old models gets replaced or refined when we observe phenomenas which is not described by the models. The models we use have limited degrees of freedom and depends on abstraction level of the system it describes. Atom models and how they have changed through time is a good

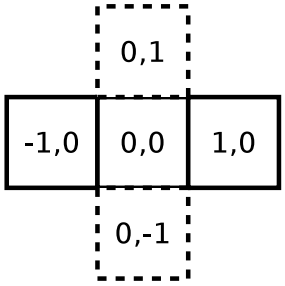


Figure 2.5: Von Neumann neighbourhood showed in two dimensions. 1-dimensional Von Neumann neighbourhood is without the stippled cells (only horizontal axis), while 2-dimensional includes all cells(vertical axis included).

example, from a cubic model with electrons placed on the corners to the electron cloud model used today. Another is how the earth rotates around the sun. On a high abstraction level the result is a system which earth has an elliptic trajectory around the sun, which is fairly easy to model mathematically. However looking at a lower level of abstraction, to model the system we would potentially have to take into account the entire universe and how it all fits together to model the system perfectly, which is not possible.

In other words, what the models we use today describe, is small subsets of the entire search space and we are limited by our own knowledge about how these system work and should work. This is what evolvable hardware is all about and more specifically evolution in materio. It tries to free the design and creation process of a system, from the constraints enforced by our models, to exploit any underlying functionality present.

2.3 Cellular Automata

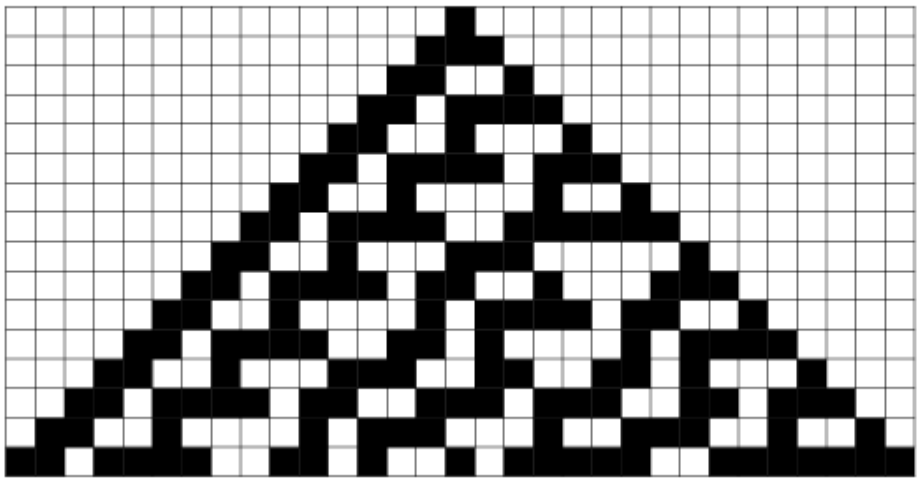
Cellular Automata (CA) are mathematical models for complex natural systems. These systems can contain large numbers of simple components with local interactions. They are discrete dynamic systems with simple construction, but can exhibit complex self-organizing behavior [14].

Cellular automata consists of a grid of cells can be 1-dimensional or of a larger dimension, it's easiest to look at them in 1 or 2 dimensions, because how they are visualized. The cells can be in a finite number of states. The easiest is either "0" or "1", false or true. Cells have a defined neighbourhood, often in a 2-dimensional grid the neighbourhood is a Von Neumann neighbourhood. Simplified to left or right neighbour when using 1 dimension, as shown in

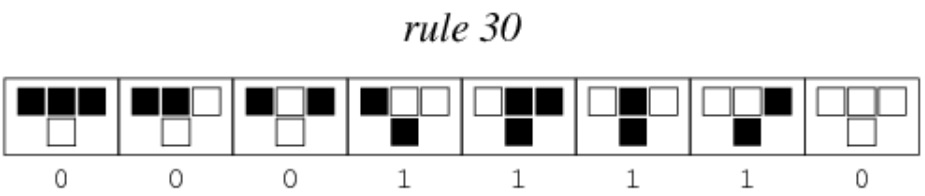
figure 2.5. The figure shows a Von Neumann neighbourhood with range 1 in both dimensions, however larger ranges can be used.

Each cell also has a function which decides the new state when updating the cells based on their own value and their neighbourhood. Figure 2.6a shows a typical space-time plot of a CA, the time axis is vertical starting with $t = 0$, the initial state and the horizontal is how the cells are laid out in space.

Cellular Automata evolved from work by John Von Neumann among others. However Von Neumann's interest was mainly building a self-replicating machine not the CA environment itself. Von Neumann created the self-replicating automata in a kinetic model which can be described more or less as a robot. Stanislaw Ulam had worked on a cellular model and state spaces [15] and suggested to Von Neumann a more theoretical approach to the problem, which was free of the physical constraints of the kinetic model. Von Neumann created a self-replicating machine called "Von Neumann universal constructor" based on Ulam's suggestions, which was published later by Arthur W. Burks [16]. This machine used 29 different states and was able to recreate itself by reading its own cell states in a CA environment.



(a)



(b)

Figure 2.6: Rule 30, figure (a) shows the chaotic behavior of rule 30 (b) shows the Wolfram code of rule 30 (Taken from [4])

Many years later, Stephen Wolfram started studying cellular automata using computer simulations. In 1983 Wolfram published a statistical analysis of all the elementary cellular automata (ECA) [14] and devised the number system for them known today as the "Wolfram code". The code is simply the output of the cells in the next state encoded as a binary number, where the respective input increases from right to left as can be seen for rule 30 in figure 2.6b. So for rule 30 this will be $00011110_2 = 30_{10}$.

In 1984, Wolfram published [5] a classification system for the cellular automata. In which he describes 4 classes each with different characteristics and present evidence that all one-dimensional cellular automata fall into one of these classes. These classes are:

1. Class I - almost all initial states evolve into a homogeneous state. Any information in the initial state is destroyed.
2. Class II - almost all initial states evolve into simple structures, which

are either stable or periodic with typically small periods. Perturbation in initial state tend to stay local. Information in the initial state tend to be "filtered" out.

3. Class III - almost all initial states evolve into aperiodic patterns, chaotic. Perturbation in initial state tend to spread indefinitely. Any structures are destroyed quickly.

4. Class IV - almost all initial states evolve into complex structures, in most cases these structures are seen to "die". In some cases stable or periodic structures persist.

The Class 4 type of cellular automaton is special, because it is shown that many of these have characteristics that indicates they are capable of universal computation. That is, such a system can compute everything which is computable i.e. create a computer from only CAs. This was proven by Von Neumann in [16] and later by Codd [17], Smith [18] and Conway and co-workers [19], Fredkin and Toffoli [20]. Wolfram's conjecture about ECAs being to simple to support universal computation was disproved around year 2000 when Mathew Cook published a proof [21] that ECA rule 110 is computationally universal.

The CA in figure 2.7 and 2.8 show several different CA with very different behavior, these are all with k=2 and r=2. Which means 2 different states "on" or "off" ($k = 2$) and neighbourhood of 4 including the cell's own old state ($r = 2$, 5 in total, 2 on either side plus old state). Both figures show the same CA rules, but different initial conditions. Notice that figure 2.8 differs from 2.7 significantly in the emergent behavior, yet still each cell does exactly the same thing. The first row exhibits chaotic behavior, second row more complex structures are formed, the third goes into a quiescent state quickly and the last row exhibit the most complex behavior.

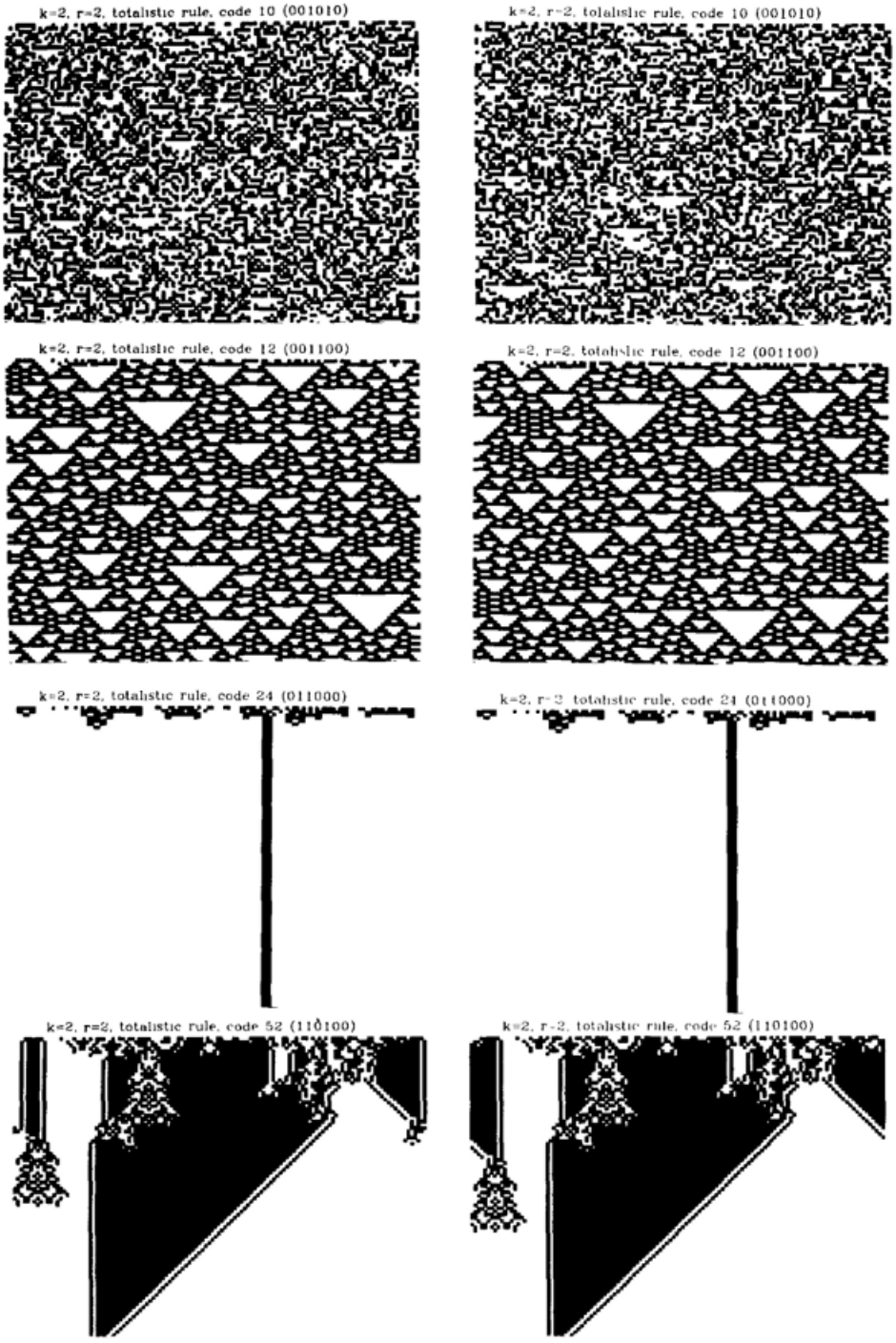


Figure 2.7: Wolfram CA with k=2 and r=2 (Taken from [5])

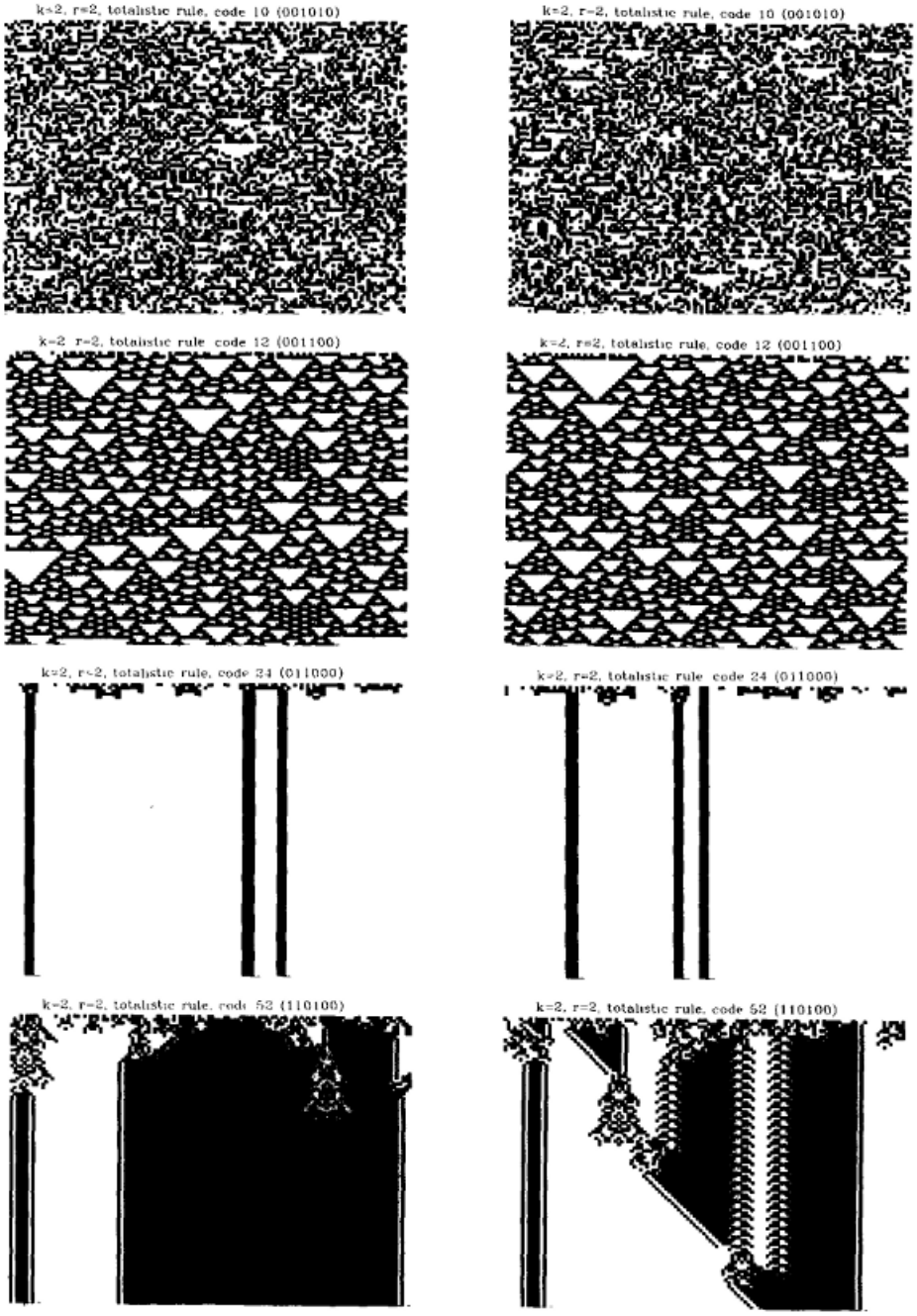


Figure 2.8: Wolfram CA with k=2 and r=2, with different initial condition (Taken from [5])

Langton explores in [6] the conditions under which cellular automata can support information transmission, storage, and modification as needed by computation. Langton's paper generally supports Wolfram's findings and shows an explanation for the classes and the relationship between them. By using the λ parameter, defined as equation 2.1.

$$\lambda = \frac{K^N - n}{K^N} \tag{2.1}$$

Where K is the number of cell states, N is neighbours and n is the number of transitions to the quiescent state. Using this λ parameter Langton searched the CA space by generating different CA by stepping through $0 \leq \lambda \leq 1.0 - \frac{1}{K}$. The results in figure 2.12 and 2.12 show how the CA behaves differently at very low λ values while in the vicinity of $\lambda = 0.5$ the behaviors exhibits similarly behaviours as computations. And at high values they exhibit more chaotic behavior.

Here transients are the CAs trajectory through state space moving towards an attractor, or the behavior the CA settles in. Langton observed that in the phase transition between ordered and chaotic the transient lengths grow rapidly, which is known as "critical slowing down". This is shown in figure 2.9, which plots transient length as a function of λ. The figure shows that the time before the CA settles in an attractor increases dramatically around the edge of chaos.

Langton also observed that for low and high values of λ the size of the CA had little effect, but when for $\lambda = 0.5$ the transient length grew exponentially, transient length is plotted as a function of CA size in figure 2.10.

Another observation was that, in the vicinity of the phase transition it supported both static and propagating structures. Evidence of the can be seen in CA figure 2.11 and 2.12.

Langton uses Shannon's entropy H to measure the information in the system. As defined in equation 2.2.

$$H(A) = -\sum_{i=1}^{K} p_i \log p_i \tag{2.2}$$

Langton uses the definition of mutual information in equation 2.5, as a meassurement of the degree of correlation between the state of cell A and B.

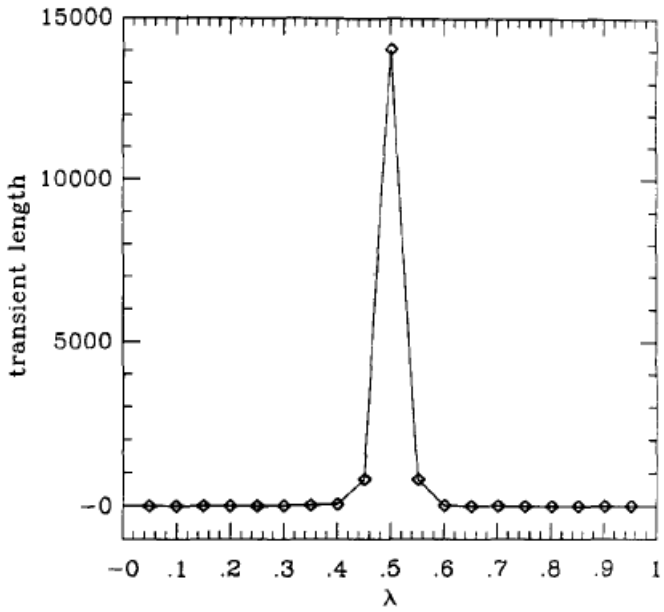


Fig. 3. Average transient length as a function of λ in an array of 128 cells.

Figure 2.9: Rapid transient length growth in the vicinity of the phase transition between ordered and disordered dynamics (Taken from [6])

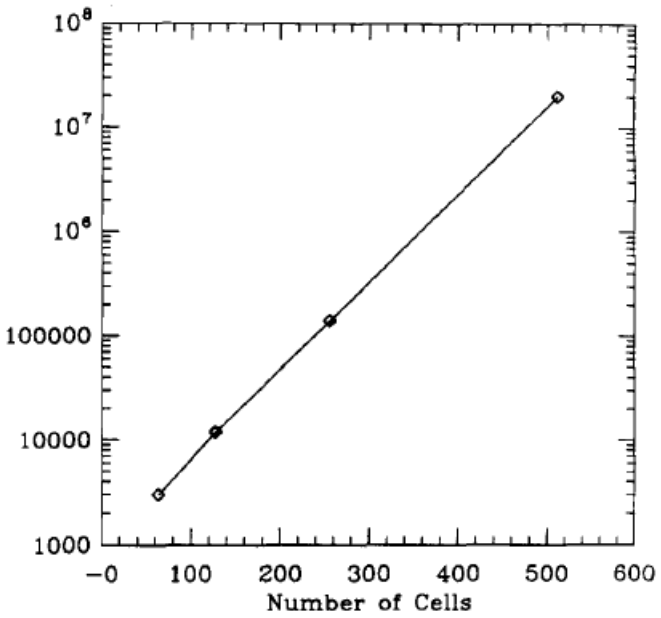


Fig. 4. Growth of average transients as a function of array size for $\lambda = 0.50$.

Figure 2.10: Growth of the transients as a function of CA size (Taken from [6])

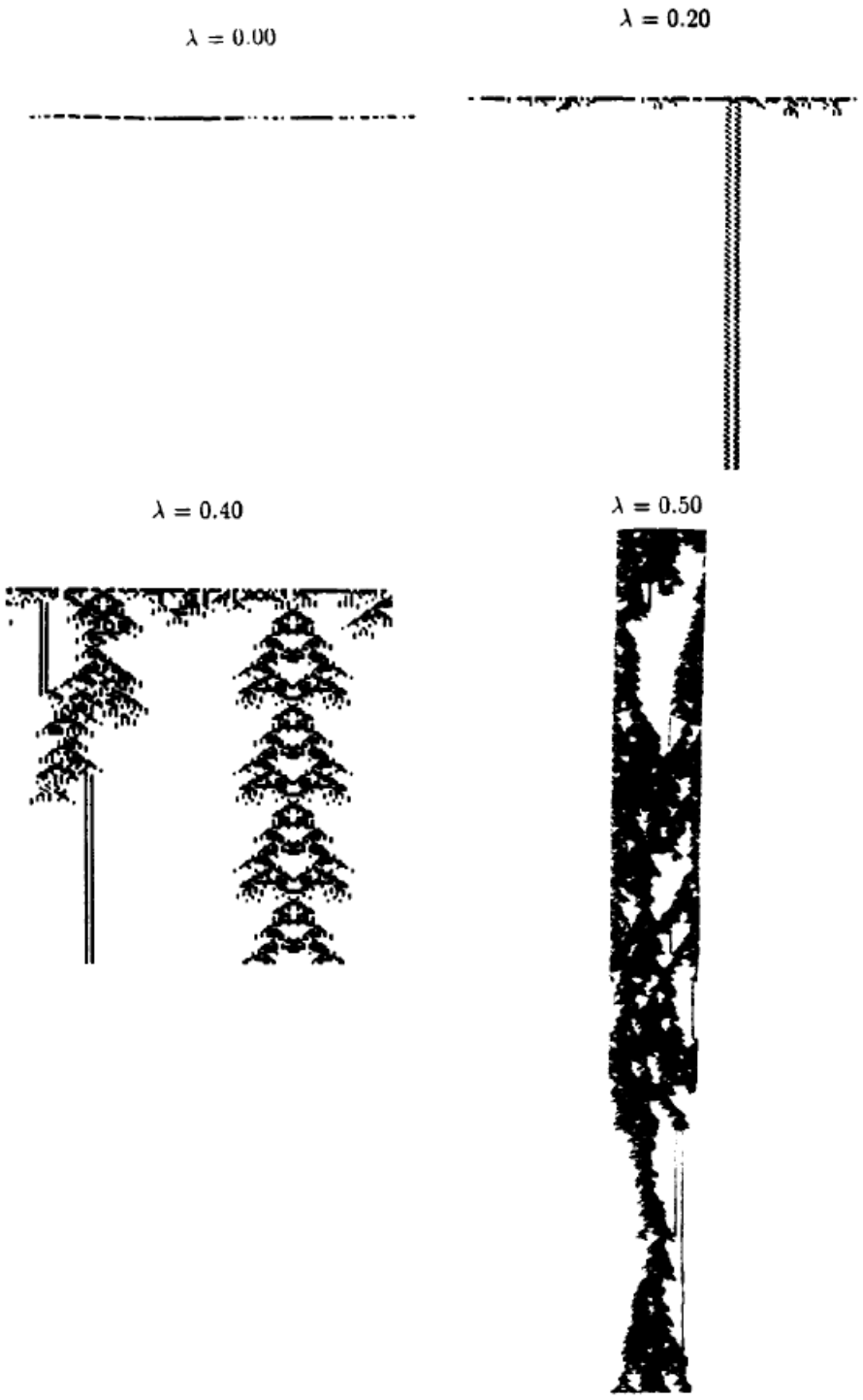


Figure 2.11: Langton's CA with K=4 and N=5 (Taken from [6])

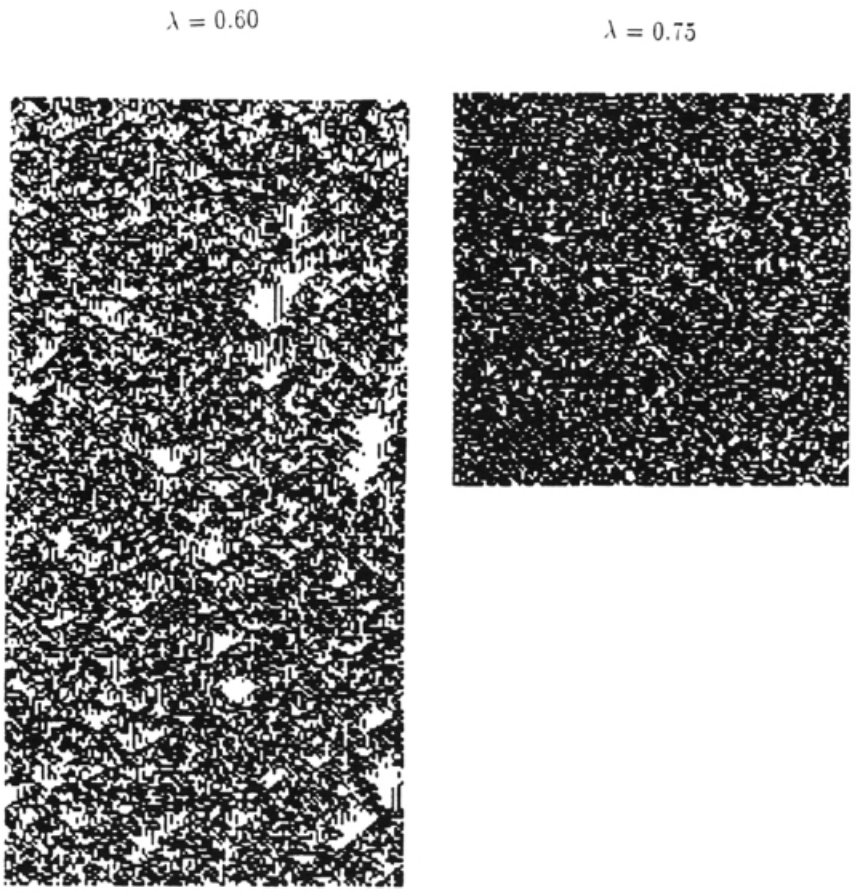


Figure 2.12: Langton's CA with K=4 and N=5 continued (Taken from [6])

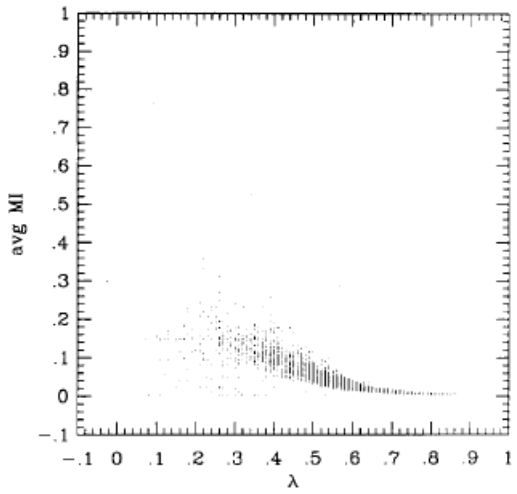


Fig. 11. Average mutual information between a cell and itself at the next time step.

Figure 2.13: Shows how at higher λ a cells behavior becomes more chaotic (Taken from [6])

Where $H(A)$ and $H(B)$ are individual entropies and $H(A, B)$ is the joint entropy defined as equation 2.3. Where a and b are particular values of A and B, $P(a, b)$ is the probability of these values occurring simultaneously.

$$H(A, B) = -\sum_{a}\sum_{b} P(a, b)\log_2[P(a, b)] \tag{2.3}$$

$$P(a, b)\log_2[P(a, b)] = 0, \text{ if } P(a, b) = 0 \tag{2.4}$$

$$I(A; B) = H(A) + H(B) - H(A, B) \tag{2.5}$$

The mutual information plots 2.13 and 2.14 shows that the complex behavior which a system needs to support computation is when there is just enough correlation between cells, but not too much. Too much correlation implies a very static behavior. The jump in the average mutual information indicates the transition to chaotic behavior. And as λ increases even further the system act more and more random, as is indicated by the decreasing average mutual information.

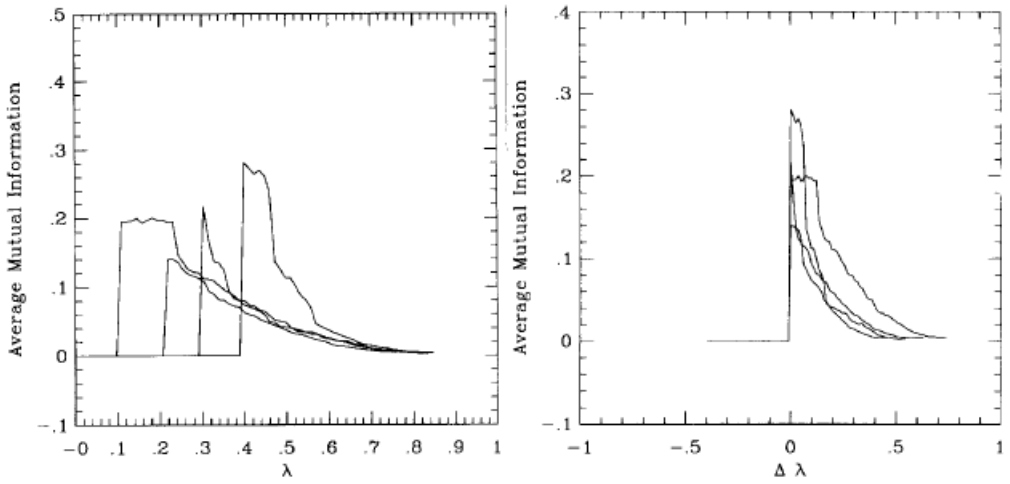


Fig. 12. Average mutual information versus λ and Δλ. The mutual information in this case is for a single time step at a single cell.

Figure 2.14: Shows how the average mutual information spikes around the phased change event or edge of chaos (Taken from [6])

To see the Wolfram classes in the context of the λ and mutual information. The higher λ the more complex behavior the CA exhibits, up to a certain point. Past what is referred to as the edge of chaos, the system is too chaotic or random which in means that the entropy is decreasing again. 2.15.

However as also Langton pointed out; "Finally, it must be pointed out that although the examples presented illustrate the general behavior of the dynamics as a function of λ, the story is not quite as simple as we have presented it here.", Melanie Mitchell, James P. Crutchfield, and Peter T. Hraber revisited the original experiment of Langton [22], with evidence suggesting that the "average" statistical properties of the CAs are not correctly captured by the λ. As others have pointed out later, like Miller and Page [23], there are multiple edges of chaos.

What does that mean for our complex system? Based on earlier research of CAs it is proven that the class 4 CAs are the ones that can support universal computation, [16, 17, 18, 19, 20, 21]. However in our case the CA guides or evolves the material, depending on how the CA rules are set up in our platform it might be possible that the complete complex behavior (CA and material) combined can be the equivalent of a class 4 CA and able to do computations, or that the CA is able to evolve the material to some state that the material alone can perform some complex class 4 computation.

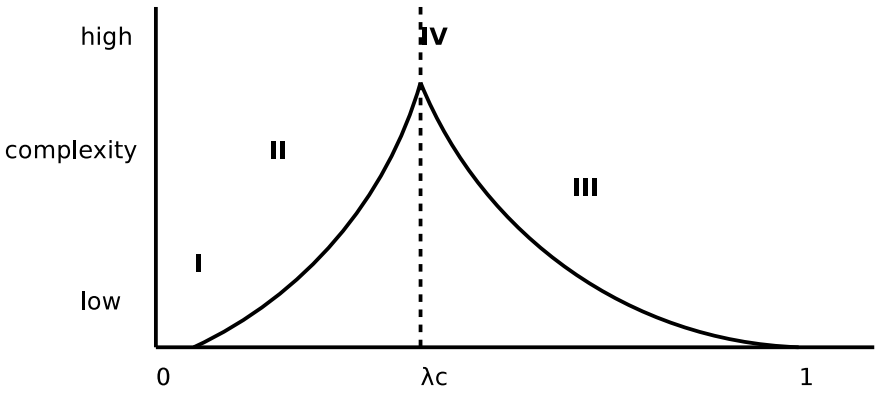


Figure 2.15: Location of the Wolfram classes in λ space (replicated figure from [6])

When that is said, it is not possible to rule out that other classes might be able to produce interesting results as well. Even if the class 4 CAs are proven to support universal computation, this might not be needed or even wanted in all applications.

When dealing with cellular architectures and even parallel architectures in super computers today, a big task and problem is to program it correctly and as efficient as possible. For "normal" parallel Von Neumann like architectures there is a good basis and understanding of how to program them, and many tools and paradigms exist to help the programmer. However for CAs or other cellular architectures the task is not that simple.

The reason for this is that it is the emergent behavior of the system which is the result of the computation. Moshe Sipper categorises the new programming paradigms for cellular programming [24] in two main categories. Cellular programming requires distinction between a local and a global task, where a local task is what each cell does and a global task is what the cells does collectively known as emergent behavior.

- Direct programming
  In this category the programmer must specify the system completely, what all the individual cells should do. Configuring a FPGA could be seen as a loose analog to this, were the logic cells must be configured individually to perform a certain task in the global picture. Synthesis tools can be used for this, but the problem is that the programmer has to define the global task of the system in a descriptive language and the synthesis the converts it into a complete cellular specification. As per now it is no general synthesis algorithm which can convert from

any language to a cellular specification to solve a global task.

- Adaptive methods
  These methods are usually the goto method for a lot of research involving problems with a huge search space where it is hard or unknown how to program or configure the system efficiently or even correctly. These methods let the programmer partly specify the system, while it is used in an adaptive process like learning, evolution or self-organization to find the right functionality. Genetic algorithms used to evolve a CA is a good example as an adaptive method. The adaptive methods abstracts the systems task so that it mostly look at what the emergent behavior is, not what each cell does in particular, the local task is abstracted away.

2.4 Genetic Algorithms

Genetic algorithms became popular in the 1970s through John H. Holland's, his colleagues' and students' work at the University of Michigan [25, 26, 27]. A genetic algorithm is a search heuristic used in optimizing and search problems. It works by imitating natural evolution and the selection process. There are many variations and impossible to describe them all, but they all share the common core.

GAs have an initial population of individuals. Each generation, the fitness is calculated of each individual. The best individuals are chosen and they produce offspring, some of the population dies. And it repeats the process with the new population of individuals. A generic GA is listed in 2.1. Where n is the number of generations to run and m is the population size.

Listing 2.1: A generic genetic algorithm in pseudo c code

```
1  void
2  generic_ga(size_t n, size_t m)
3  {
4    size_t i,j;
5    population *p = generate_population(m);
6    for(i = 0; i < n; i++) {
7      population *p_new;
8
9      calculate_fitnesses(p);
10
11     for(j = 0; j < m; j++) {
12       parents *pr = choose_parents(p);
13       individual *c = create_child(pr);
14       mutate_individual(c);
15       add_individual(p_new, c);
16     }
17     p = p_new;
18   }
19 }
```

Mitchell in [7], shows a simple GA and how to use the GA to evolve cellular automata to perform a density classification task. The CA evolved should settle in an all 0 state or all 1 state, depending on how many 0s and 1s there are in the initial state. The programs which the cells run is encoded in the individuals, and when offspring is made, crossovers and mutations happen

and the programs are modified.

The density classification task, is also called a majority voting. The majority voting function can be defined as in equation 2.6. Performing this calculation on a Von Neumann type architecture or by thought is trivial. But in a cellular automata this is not a trivial task. No cell in the CA knows the complete count of either 0s or 1s in the system.

$$Majority(p_1, \ldots, p_n) = \left\lfloor \frac{1}{2} + \frac{\sum_{i=1}^{n} p_i - 1/2}{n} \right\rfloor \tag{2.6}$$

What is interesting to note from the Mitchell's results 2.16 (and others like it) is that using a simple GA, the CA evolved was able to perform the task quite well. The task performed is the equivalent of "a lot" of computation in a Von Neumann type architecture, count all zeroes and ones, then compare them (subtract one from the other), based on the result set majority to either 0 or 1.

On the other hand, because there is actually a lot of computation involved to actually evolve the CA. Imagine if this could be achieved at very low level, type bacteria or in a nano or micro-scale material. While the evolution of such a system might be costly the resulting system could be very power-efficient and scalable.

GAs are often used on search and optimization problems, because it would cover the search space better than with an exhaustive search. And an exhaustive search would take too much time in most cases. The rules of the CA is encoded in the individuals of the GA. And each of these individuals fitness is calculated from how well the CA does the density classification task.

It is important to understand that it is not trivial to find rules which makes the CA do the task which the programmer wants it to do (Sipper's categories of programming techniques [24]). In a normal processor it's straight forward to program the complex behavior. However in a CA the complex behavior is emergent, and each rule must be carefully chosen to support this emergent computation. The CA is a self-organizing system in which the complete behavior is emergent from local interaction between the cells.

Using a GA to evolve the rules for the CA is the equivalent(but not similar) of a programmer which creates a program for a computer based on a Von Neumann architecture. In the case of the Von Neumann architecture the programmer already knows how to program it, while in the CA case there is

currently no programming paradigm which is generally compatible with this type of cellular computing.

2.5 GA, CA, Evolution in materio, in Mecobo

Each of the methods in use in Mecobo has its own job. A GA can be used to search for a set of CA rules or programs to run on the Mecobo platform. This CA uses the CA rules and inputs/outputs from/to the material under research. The co-operation between the CA and material can be viewed in at least two different ways. The first is that the CA and material are part of the same "CA" and self-organizes together. The other is that the CA rules are evolved by the GA, and the material is evolved by the CA.

The CA is changing the next configuration of the material based on its own state and the output of material. How the user are going to interpret the results are up to the user, since there is many different ways to model this co-operation on the Mecobo.

It is unknown if and how the theory of CAs will be reflected in the results of this system, since the behavior depends largely on the material under research. However when looking at Pask's results, it sounds plausible that the more complex the behaviour of the CA, the more complex the structure in the material will become, depending on the material of course. If the material does not retain any information between configurations, the evolution the CA "guides" the material through, will have no permanent effect, and the behavior of the material will only be defined by the current configuration.

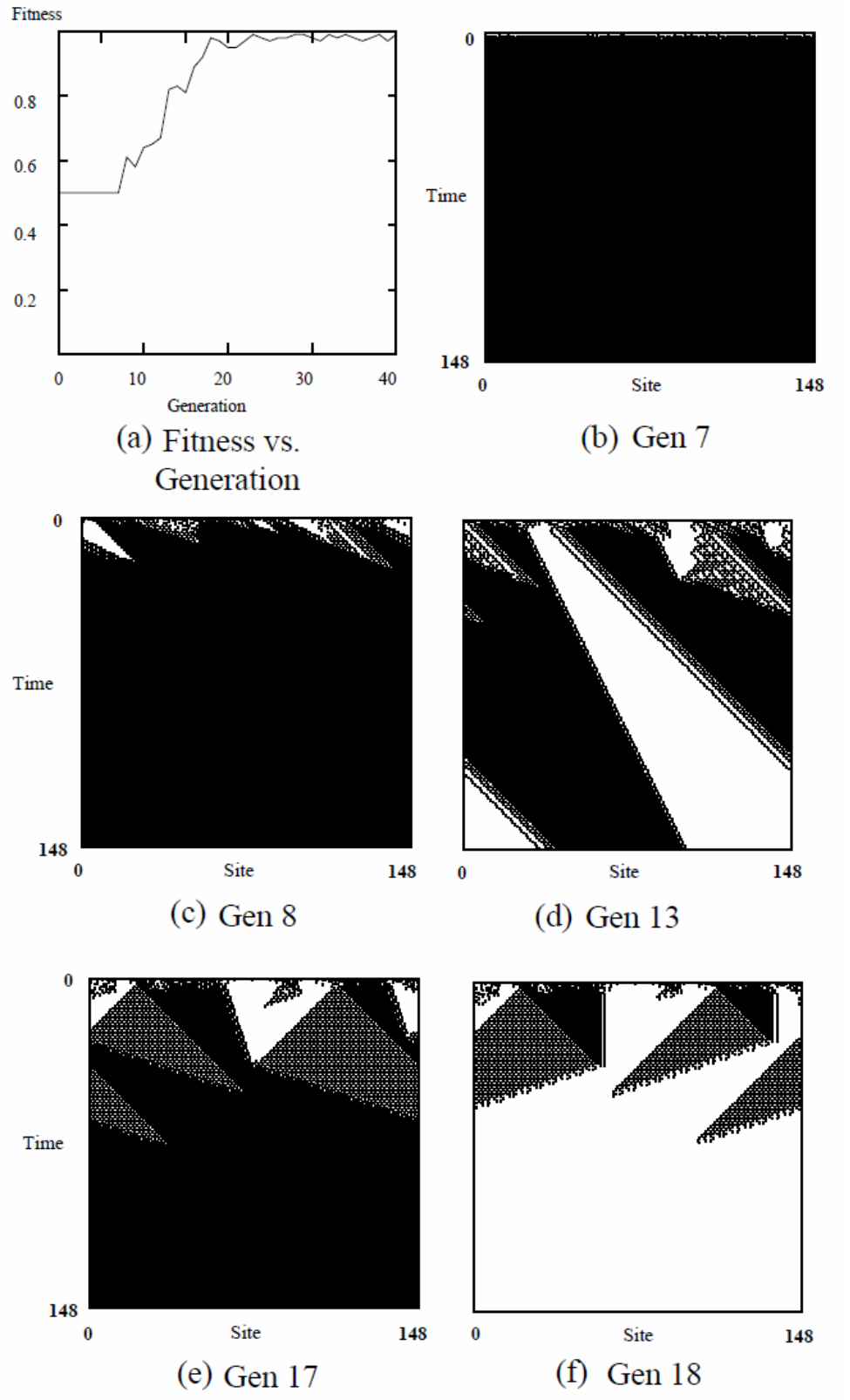


Figure 2.16: (a) fitness each generation (b)-(f) space-time plots of the behavior of highest-fitness individual (Taken from [7])

Chapter 3

Overview

The Mecobo platform is an experimental platform capable of modelling CAs and experimenting with a material and capturing its behavior both alone and in conjunction with a CA. It consists of a software framework running on a host computer, a microcontroller, an FPGA and a material bay interfacing the material.

Currently there is two main usages of the Mecobo platform. The first is it can act as a interface to a test material which the user can configure from a computer connected to Mecobo through USB. The second is it can run a CA model in conjunction with the material, in which the CA changes the initial configuration of the material based on the response from the material automatically. The CA model and how Mecobo updates the configuration is programmed by the user.

The CA model and initial configuration is supplied by the user from the host computer through the microcontroller which loads the CA model and initial configuration into FPGA memory and then starts the evolution. The user then waits for the evolution to finish and gets back 256 samples of the evolution from the Mecobo platform.

The most significant functional difference in the new version of Mecobo is that the platform now has the ability to evolve the material under research as part of a complex system. A system which uses the Mecobo platform consists of 3 parts shown in figure 3.1.

1. Host computer. Where the user evolves the cellular automaton used.

2. Mecobo platform. Consists of a microcontroller and a FPGA.

3. Material bay. The interface to the material seen in Figure 3.3.

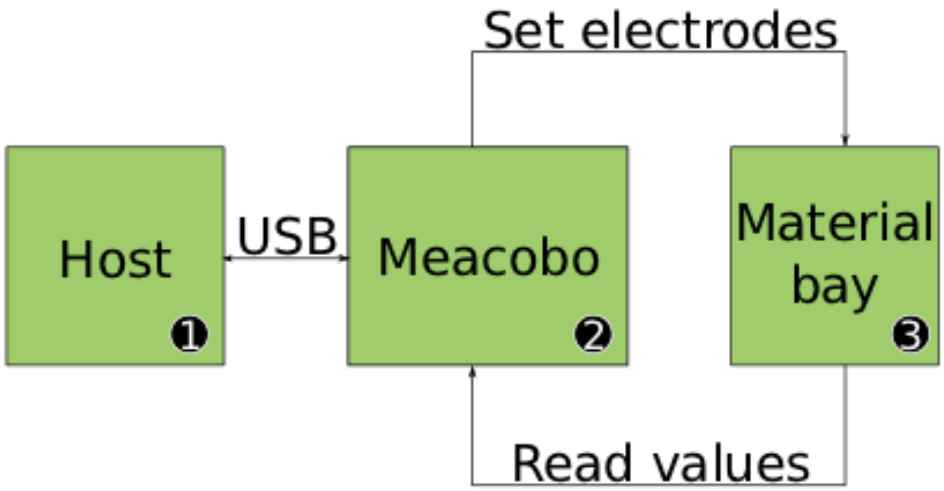


Figure 3.1: The host computer controls the material bay through Mecobo. (Taken from [8])

The old system (shown in figure 3.2) was limited to only setting and reading values on the Mecobo platform, the rest of the evolution had to be done on the host and it had no support for recognizing any self-organising behavior of the material other than in terms of the evolution on the host. The configuration was written from the microcontroller into the memory, and when the action is started by writing into the command address, the "usr_module" reads out the configuration and sets the pins through the "pin_control" module. This design suffered from a flaw in which either the microcontroller or command control state machine in the FPGA would potentially be starved, since both use the "memory" module through the same state machine in the memory controller.

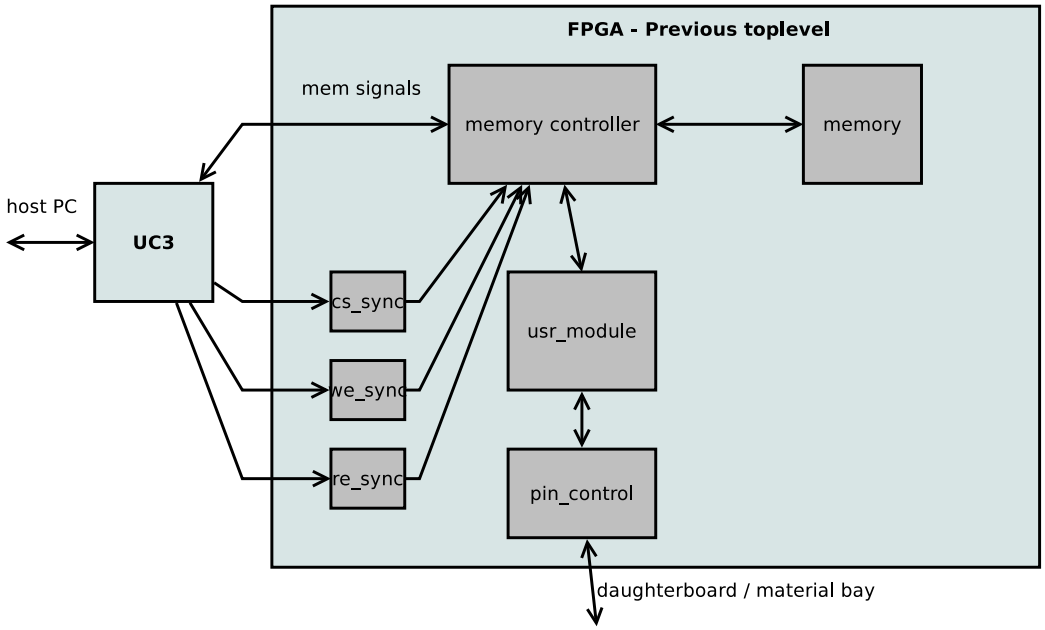


Figure 3.2: The old Mecobo FPGA toplevel, before the addition of the new extension

With the new extension the material is now part of a complex system, it is connected to a cellular automaton. So the material and the CA organizes together and exhibits a unified behavior. This behavior is not easy to couple to the theory behind CAs, because it is not know what sort of connection is formed inside the material under research. Only the future will reveal whether this kind of combined behavior has something to it or not. The new toplevel is designed as shown in figure 4.3. In the new design a true dual-port memory [28] is used to solve the starvation issue. The memory controller is now rewritten and called "uc_interface", and there are three different kinds of memory, one for CA state samples, one for configuration and instruction memory for each core inside the CA CPU.

The CA state memory is unwritable from the microcontroller's side to avoid potentially damaging the samples.

The command control in the "cmd_control" has the same core functionality as the old "usr_module". That is, it reads the command address waiting for a command when it gets a command, it dispatches the command and loads registers, configure pins through the "pin_config_control" module and runs the CA CPU depending on the command supplies. The pin controller is also

rewritten and now sets all the pins in the same clock cycle, unlike the old design which incrementally updated 8 pins at the time.

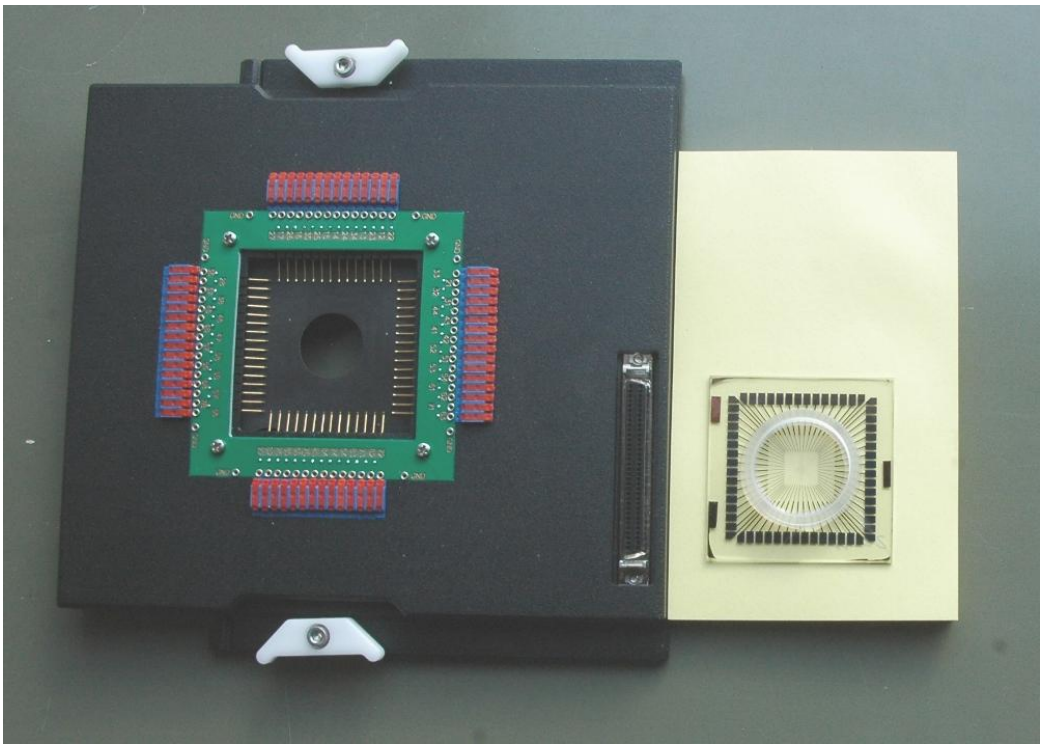


Figure 3.3: The MEA Amplifier to the left, without the replaceable electrode array inserted, shown on the right. (Taken from [8])

3.1 The Idea

The thought behind this combined complex system of CA and the material lies in the fact that evolving the material alone is a very static approach. Only one configuration can be applied at the time so the material will either exhibit interesting results or not, perhaps based on what configuration has been applied to the material before. Since we already know from earlier research that many CAs do indeed have interesting behaviour [5, 6, 21], we want to see if we can exploit the natural emergent behaviour of CAs in conjunction with an unconventional material to see if we can evolve a system which exhibit equal or more complex behavior than the CA or material would alone. This new extension of Mecobo is designed with this as the main goal.

In this hybrid CA and materio approach the material can be seen as an analog part of the CA itself, which might have the right properties to exhibit some sort of behavior other than a chaotic or seemingly random behavior.

3.2 CA models

The configuration data referred to in this section, is the 128-bit register, which holds both value bits and mode bits for each pin. Where the mode bit is set to input(that is, not driven from the FPGA) the value of this pin is actually the output of the material. When the mode bit is set to output, the pin is driven from the FPGA.

The mathematical model which describes how the hybrid CA material model is updated is based on Wolfram's CA equation 3.1 (from [5]). Where k is the number of states each cell has, r is the range to either side, t is time-step and i is pin number.

$$a_i^t = \boldsymbol{f}\left[\sum_{j=-r}^{j=r} \alpha_j a_{i+j}^{t-1}\right] \tag{3.1}$$

$$\alpha_j = k^{r-j} \tag{3.2}$$

First is the definitions of the pins available to the FPGA and material. These are defined as in equations 3.3-3.8. Where set A is the set of all the pins and FPGA is the set of pins driven from the FPGA and MAT is the pins driven from the material (inputs to the FPGA). And n is the total amount of pins.

$$\mathrm{A} = \{b_0, \ldots, b_{n-1}\} \tag{3.3}$$

$$\mathrm{FPGA} \subseteq \mathrm{A} \tag{3.4}$$

$$\mathrm{MAT} \subseteq \mathrm{A} \tag{3.5}$$

$$\mathrm{FPGA} \cap \mathrm{MAT} = \emptyset \tag{3.6}$$

$$\mathrm{A} \setminus \mathrm{FPGA} = \mathrm{MAT} \tag{3.7}$$

$$\mathrm{A} \setminus \mathrm{MAT} = \mathrm{FPGA} \tag{3.8}$$

The layout of the pins and connection is visualized in figure 3.4, where the D indicates a driven pin from the FPGA and the others are inputs from the

material. The pins which is not marked with D is updates straight from the material in time-step t. Before the FPGA pins in the next time-step $t+1$ is updated with cell function f as can be seen in equation 3.9. M is the material function or response of the material.

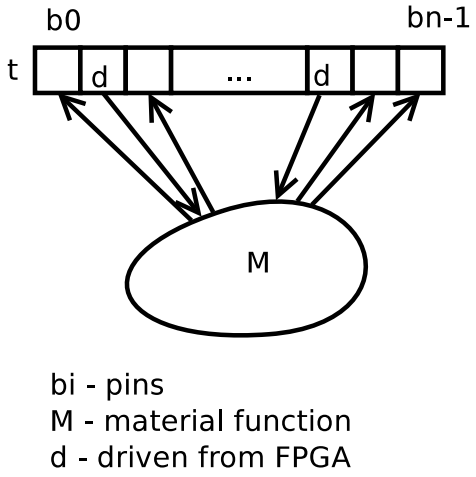


Figure 3.4: Mathematical model of the hybrid CA material system

Equation 3.9 shows the updating process for both pins driven from the FPGA and inputs to the FPGA from the material where $k = 2$. The model is simplified, but captures the essence of the updating process and the general idea of how the system is working. Bits from the material pins is updated inside the time-step t based on the configuration (the bits to the FPGA driven pins) of time-step t by the material function M. M_i means that it is a input pin(since $i \in$ MAT) corresponding to the pin i from the material. Note that when $i \notin$ MAT, the function $M_i = b_i^t$, since the pin is driven from the FPGA not caring about the response of the material.

$$b_i^t = \begin{cases} M_i(b_0^t, \ldots, b_{n-1}^t) & \text{if } i \in \text{ MAT} \\ \boldsymbol{f}\left[\sum_{j=-r}^{r} 2^{r-j} b_{i+j}^{t-1}\right] & \text{if } i \in \text{ FPGA.} \end{cases} \tag{3.9}$$

There is a wide variety of ways to to program the platform, it can model a lot of different Cellular Automata. The different types of nodes or cells have different programs running on the cores. While the programs are not defined only from CA models, some of its details are encoded in the models.

The experiments run later described in chapter 5 are all uniform programs and differs slightly from models described in this section. All the cells or

nodes in the experiments are of the same type, equal amount of "in" instructions and the same boolean function executed, but different neighbourhood cells.

It is not clear from the models, that the values coming from the material are actually stored in the configuration data register, but it is shown earlier in figure 3.4. In the case of figure 3.5, there is 4 configuration pins on the material(output pins), and 4 input pins from the material. Which pins are input pins and which are output pins are arbitrary, the configuration can potentially differ from different application and further testing can perhaps shed some light on what works best for each problem type.

So the CA CPU does not know or care whether it is taking in a value from the material or a value from neighbouring nodes or itself. It is the same "in" instruction which fetches both values. This has to be manually selected so that each core actually runs the right program and takes the right values as input.

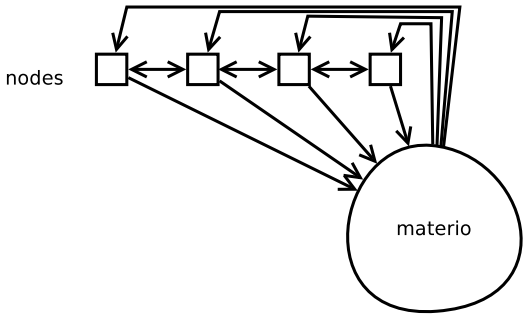


Figure 3.5: Cellular Automaton 1

The model in figure 3.5 has 4 nodes, each of these nodes are programs running in its own core in the CA CPU. There are two different types of nodes in the figure , the node on the far left and on the far right, node 0 and 3 respectively, are of the same type. The two in the middle are of the same type as well.

Node 0 has in this case 3 inputs. From node 1, from the material and not shown from its own old state. Node 3 has also 3 inputs, from node 2, the material and from its own state. Node 1 and 2 has 4 inputs, from their neighbours, their self and from the material.

While the 4 inputs from the material are also stored in the configuration data, it is clear from the model that they are not considered a part of the

CA state, in some applications it might be useful to encode the materials' output in the same CA state as the configuration values that produced it. That is, both the configuration values to the material and the output values from the material encoded in the CA state.

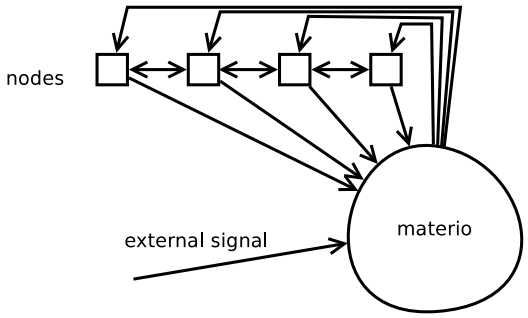


Figure 3.6: Cellular Automaton 2

What shown in figure 3.6, is essentially the same as figure 3.5. Except that there is also an additional external signal. This signal will have a pin-configuration set to input, since it is driven from outside of the FPGA, in addition to the previous input signals from the material.

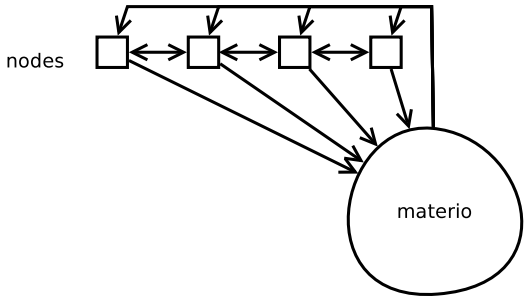


Figure 3.7: Cellular Automaton 3

Figure 3.7 shows an example, where we have 1 pin configured as input and 4 pins as output configuration. Each of the programs takes the output of the material and does some calculations with it and the previous state of the CA to produce the next configuration.

3.3 Evolution and behavior

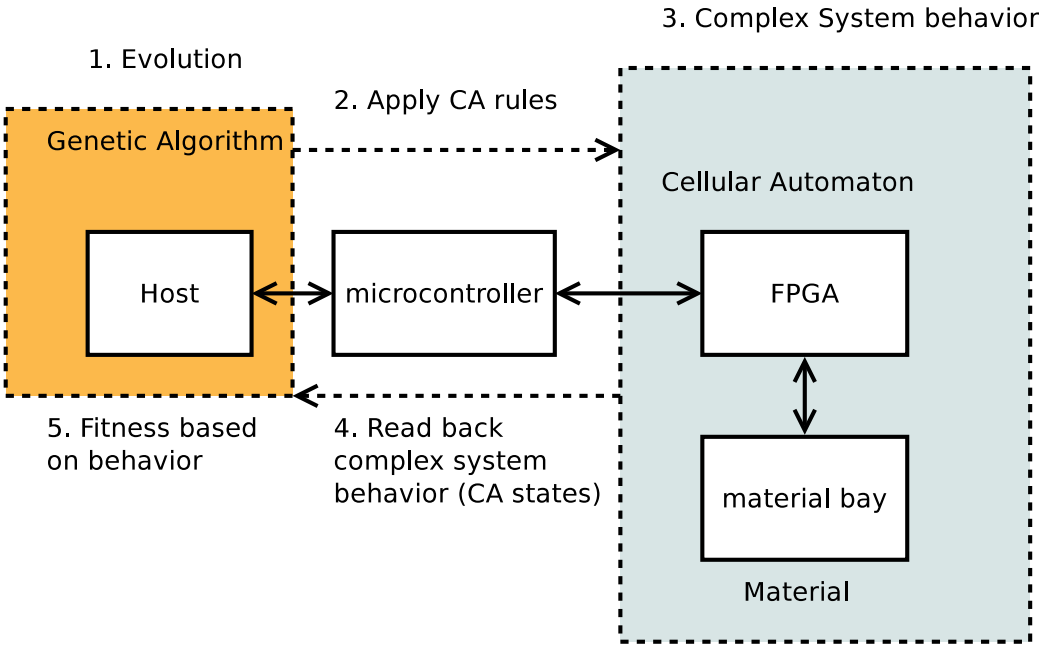


Figure 3.8: This is the complete system with the new extension and how it is suppose to be used. The solid lines are the physical system, while the stippled lines denote the abstract system and how it is suppose to be used.

The figure 3.8 shows how the system is suppose to be used and how it maps on the physical hardware. The main abstract parts is the genetic algorithm which is run on the host computer, the complex system which is the cellular automaton's and the marial's behavior together. The physical parts are the host computer, the microcontroller, the FPGA and the material bay.

The complex system is for the programmer to define, some examples are given in the previous section 3.2, but Mecobo is capable of modelling several different systems other than what is presented there.

Mecobo can be used in the following way:

- Step 1 use a GA to evolve the rules for the CA.
- Then in step 2, apply the rules/program the CA CPU.

- In step 3, start the complex system and let the CA and material self-organize in a fixed amount of steps. Each of these steps gets recorded and saved in the CA memory on the FPGA.
- In step 4, read back the state history of the complex system (the CA and material).
- Step 5 involved calculating fitness over the complex system's behavior and continuing with the GA.

3.4 Hardware and software

The host computer should be running any GNU/Linux distribution which has support for serial over USB communication. The libEMB [8] is extended with new functions to support programming the CA CPU, setting initial state and running the complex system and reading back the results. The control path can be seen in figure 3.1.

There is a new API to read and write to the FPGA's memory, this API can now handle both byte and half-word accesses to the FPGA's memory. The updated address space can be seen in figure 3.9. The FPGA uses its internal memory as little-endian memory, however the AVR32UC3 expects its memory to be big-endian [29], to overcome these troubles the interface to the UC3 on the FPGA emulates a big-endian memory while internally it uses little-endian addressing.

The extension of the system was initially designed to be as non-intrusive as possible. To retain compatibility with any old programs written for the platform. However after suffering difficulties trying to implement the new addition and interfacing with the memory, the layout and design was changed and the system reimplemented with new memories and command control. Figure 4.3 shows how the new toplevel looks like. The CA CPU is the main new addition to the system. It is a programmable 64-core CPU, its purpose is to modify the configuration of the material in test based on previous configuration and the response from the material. The CA memory module is the memory where the CA CPU stores each new configuration or state it generates.

Initially the idea was to design something simpler and more rigid, which could be programmed without much effort on the users part and still be able to change the configuration of the material based on the output from the

material in test. However, since there was no concrete application to use as a reference, and the applications of computation in materials could potentially span multiple research directions across different sciences. The only logical thing to do was to design a more general solution which did not have to be redesigned for each new application or research direction.

The FPGA address space is currently just enough to fit in 256 samples from the CA CPU before it has to transfer them. This limit is a hard-limit, limited by the fact that only 12 pins of the external bus interface (EBI) is used on the UC3 (see the AVR32UC3A datasheet [30] for details). The new address space layout can be seen in figure 3.9 and is described later in chapter 4. Note that some unchanged details are omitted for clarity.

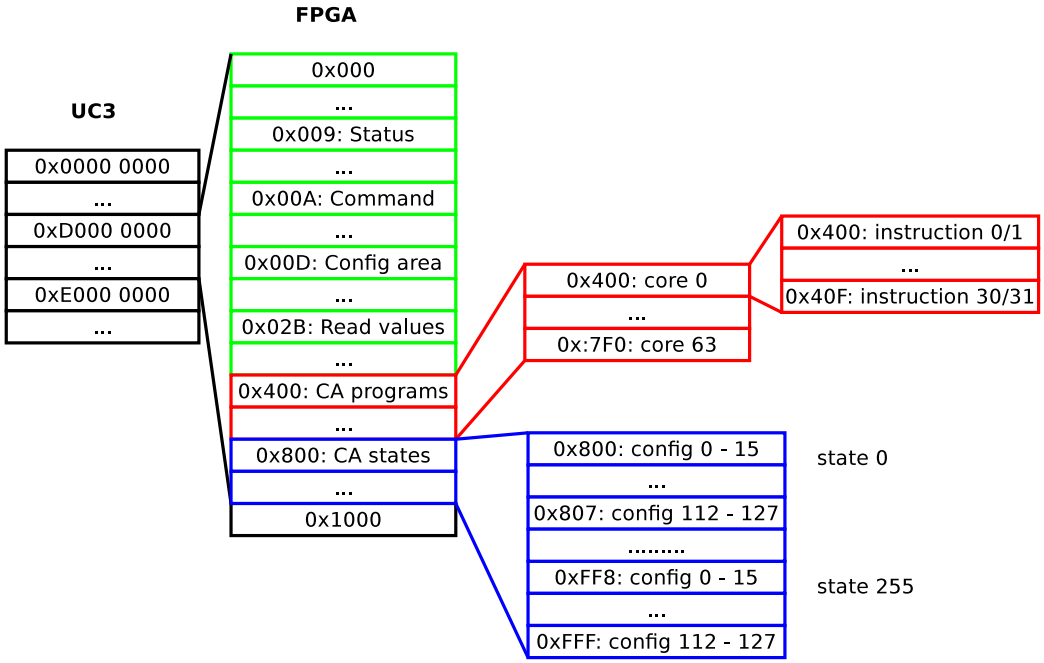


Figure 3.9: Address space layout

Chapter 4

Design and Implementation details

This chapter describes the current Mecobo platform with emphasis on the modifications. It is divided in three parts, the FPGA, the microcontroller and the host software.

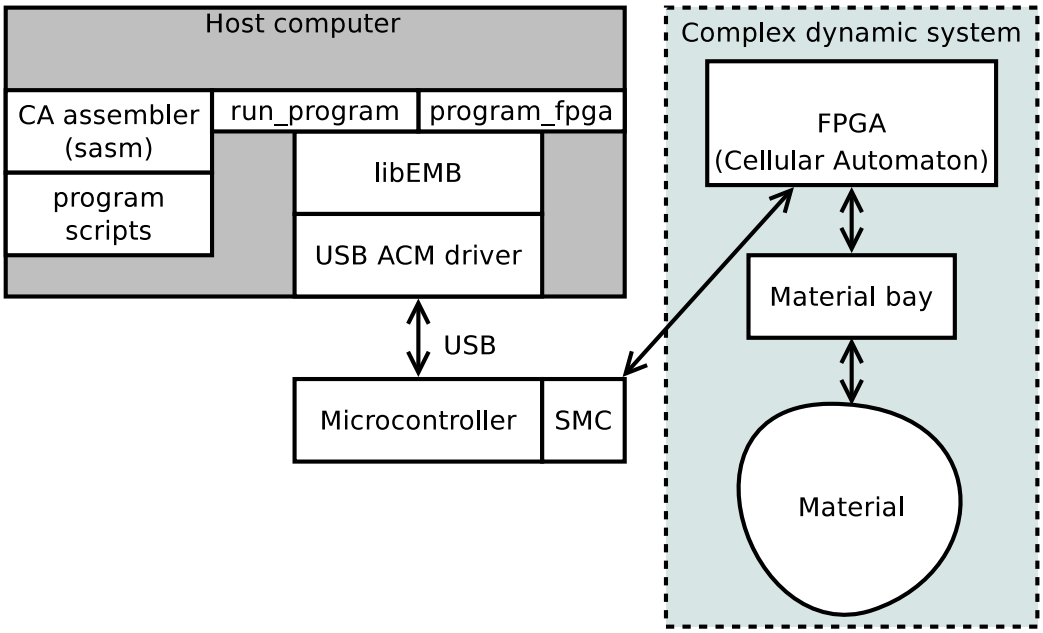


Figure 4.1: The software and hardware stack of the new functionality.

The software and hardware stack is shown in figure 4.1. In the FPGA section all the significant modules are explained and shown visually, however note that the complete CA CPU reference is in the appendix A. The microcontroller section focuses on the changes to the communication between the microcontroller and the FPGA and hardware setup. The host software section describes some of the new functionality as well as introducing some of the software used to run the experiments.

The material configuration used to configure the material is defined as 128 bits. Where even numbered bits is mode bits and odd numbered bits is value bits, as shown in figure 4.2. Throughout the thesis, this configuration may be referred to as a CA state or a material configuration, or simply configuration if the material is the context. This material configuration includes both FPGA driven values and values sampled from the material as well as the mode bits for each pin. When the CA state is stored, all of the 128-bit is stored unless stated otherwise. The different pin configurations can be seen in table 4.1 The value bit(bit 1) where the pin is material driven is the sampled response from the material.

Bit 1	0	Description
0	0	FPGA driven 0
1	0	FPGA driven 1
0	1	Material driven 0
1	1	Material driven 1

Table 4.1: Table of possible pin configurations

If bit 2 and 3 is both set to 1, this means that pin 1 is an externally driven pin, and 1 is sampled from the material. This is also shown earlier in chapter 3.2, more specifically equation 3.9 where the value bits associated with an externally driven pin is updated within the current time step.

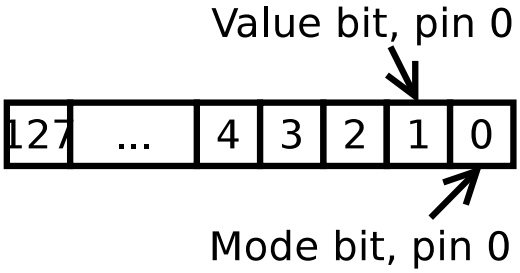


Figure 4.2: Material configuration bits (also referred to as a CA state or configuration)

As explained later in chapter 5 all the bits are not always used, more specifically in the state-space figure, where only the FPGA driven pins are encoded in the values presented.

4.1 FPGA

Due to how the extension was implemented and the old design, it was too difficult to just add the extension in the previous design. The previous FPGA firmware had certain flaws in the memory handling and command control which is the reason that those parts were re-written along with a new true dual port configuration memory [28] implementation. All the modules was rewritten except the very lowest level of the pin controller, which is the tri-state buffer for the material IO (input output) and the synchronizers for sampling the material.

The new FPGA firmware now consists of the toplevel in *toplevel.vhd*, CA CPU toplevel in *ca_cpu.vhd*, microcontroller interface in *uc_interface.vhd*, CA state memory in *ca_mem.vhd*, configuration memory in *config_mem.vhd*, command control in *cmd_control.vhd*, synchronizers in *synchronizer.vhdl*, pin controller in *pin_config_control.vhd* and low level pin control in *pin.vhdl*.

The new toplevel in figure 4.3 shows the communication between the modules, each module will be described in the following sections.

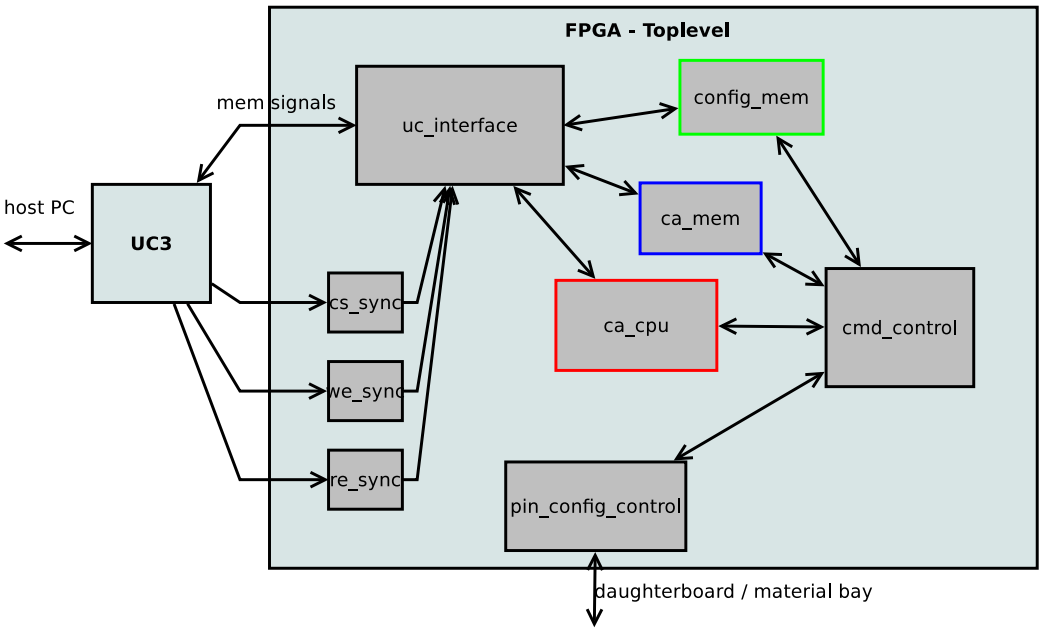


Figure 4.3: The new FPGA Toplevel

Each core in the CA CPU in *cpu.vhd* consists of a fetch unit in *fetch.vhd*,

instruction memory in *mem.vhd*, decode unit in *decode.vhd*, execute unit in *execute.vhd*(this consists of two smaller modules, the ALU (arithmetic logical unit) *alu.vhd* and configuration IO mux *io_reg.vhd*) and writeback stage in *writeback.vhd*.

4.1.1 Microcontroller interface

The way the microcontroller talks to the FPGA is by memory mapping all of the FPGA memory. The address space layout can be seen in figure 3.9. This interface's main part is the muxing of the write enable signal and memory buses to the 3 different memories in the FPGA. Based on the address on the memory address bus from the microcontroller the control FSM sets signals such that the write enable signal is sent to the right memory and the right data out is registered to be sent on the tri-state memory bus to the microcontroller. The microcontroller interface's block diagram can be seen in figure 4.4.

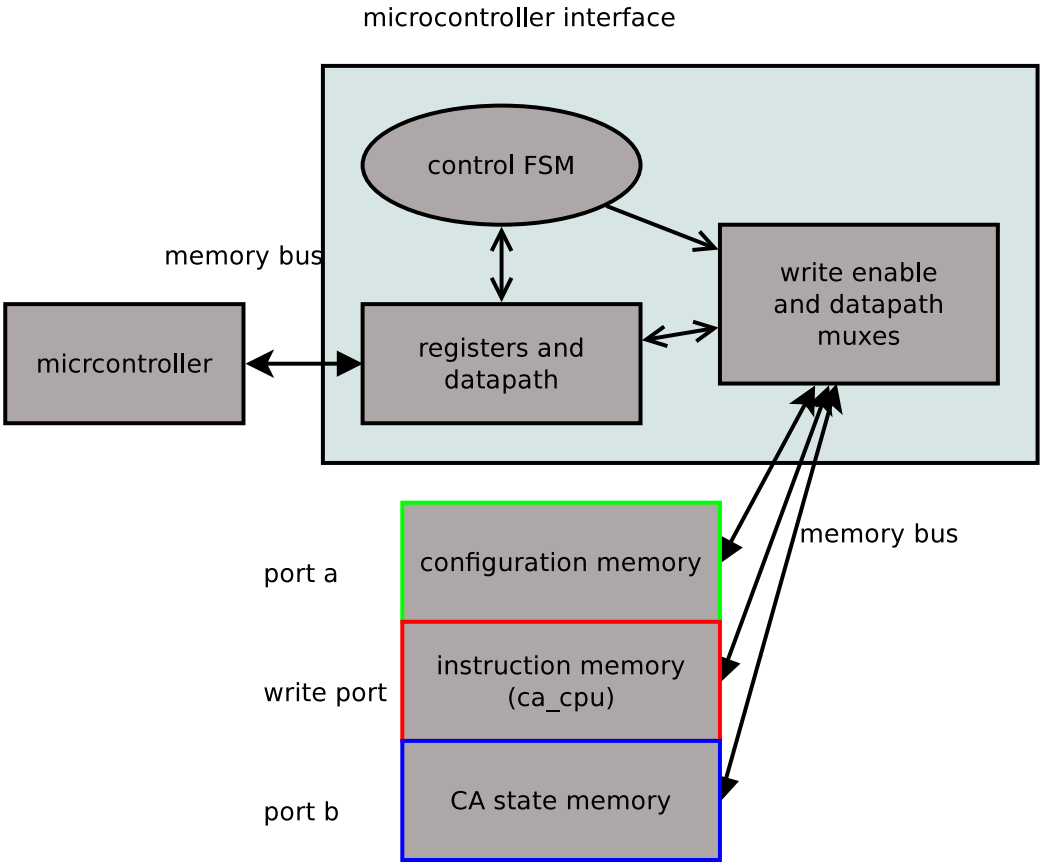


Figure 4.4: Microcontroller interface

The state machine can be seen in figure 4.5. The state machine shows both the read and write functionality, the chip select and write/read enable signals coming from the microcontroller are active low. The write enable signal passes through an edge detection unit, not visible in any figures to avoid writing data to the memory several times. This is needed because the FPGA clock frequency is higher than the microcontroller's clock frequency. The microcontroller "ACKs" the write/read operation by deasserting the chip select or read/write enable signal. Depending on how the microcontroller's SMC unit is configured.

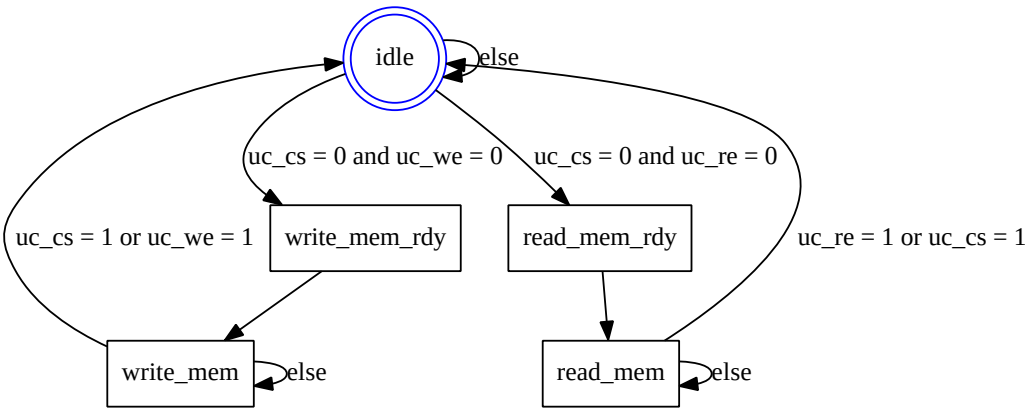


Figure 4.5: Microcontroller interface state diagram

The current implementation of the interface suffers no transfer errors like the old implementation (due to changes in both the microcontroller and FPGA at the same time, this could have been a self-introduced artifact after the FPGA changes was done) with the microcontroller SMC's read and write cycles configured as shown in figure 4.11 and 4.12. However the timings are not completely optimized on the FPGA side, it is possible to save a cycle both on read and write operations, but the current timings are tested and works.

Note that since the microcontroller in use is a AVR32, it is a big-endian architecture. And since all the modules in the FPGA is using the memory as little endian, the bytes is flipped on reads and writes so that the microcontroller writes the correct order of the bytes in the each half word(16-bits).

4.1.2 Command controller

The command controller is what controls the majority of the modules, it consists of the main state machine of the system. Its job is to read the command register(or memory address) in the configuration memory and dispatch the command. As can be seen in figure 4.7 it starts in the idle state and goes through the read_cmd_reg state where it sets up the address bus to the configuration memory and when it enters dispatch_cmd the data read is ready to be dispatched.

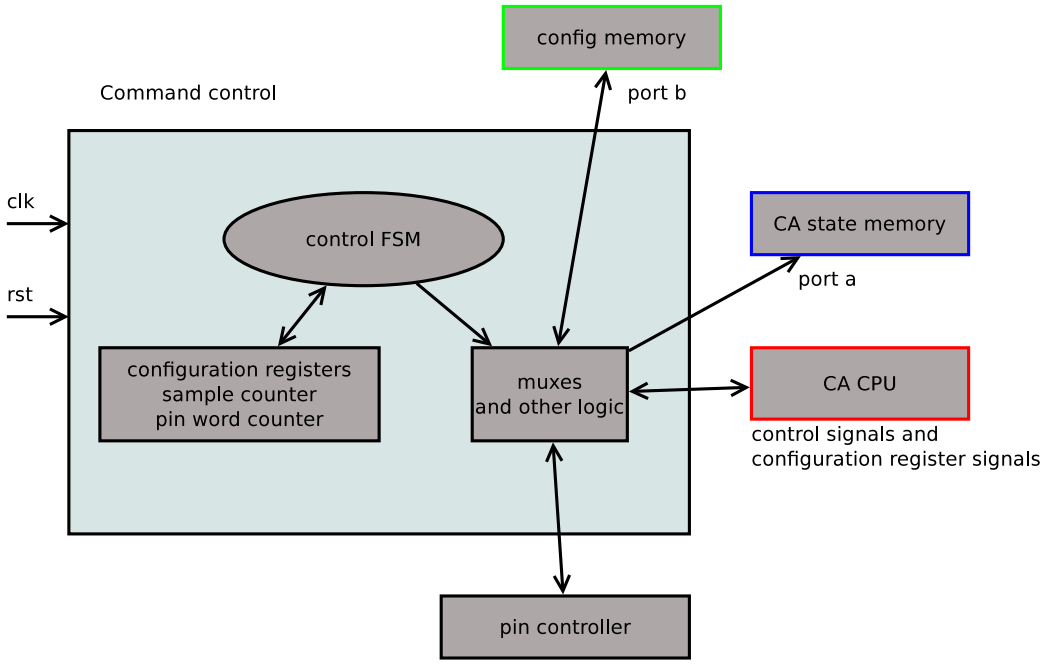


Figure 4.6: Command control block diagram

The connections between the command controller and other module can be seen in figure 4.6. It is connected to the config memory on port b, the CA state memory on port a, it controls the CA CPU and pin controller.

Currently there is 4 commands available CMD_CONF_EVO, CMD_RUN_EVO, CMD_CONF_PINS and CMD_READ_PINS.

- CMD_CONF_EVO is a new command which uses the CA CPU and material in conjunction as a complex dynamic system. The data in the plots in the results sections in chapter 5 is sampled using this command. This command reads the initial configuration from the command address in the configuration memory and uses the pin controller to set the material's configuration, after the response is sampled the FPGA driven pins' and material driven pins' values (also called a state) is written to the state memory, then the CA CPU runs its programs to create the new pin configuration. This repeats itself until 256 samples is taken.
- CMD_RUN_EVO is much alike the CMD_CONF_EVO command however it does not use the initial configuration, it continues from the last state.

- CMD_CONF_PINS is the old way of experimenting with the material, it reads the initial configuration from the configuration memory then configures the material through the pin controller.

- CMD_READ_PINS reads back the sampled data from the pin controller. This should be used after a CMD_CONF_PINS command is executed.

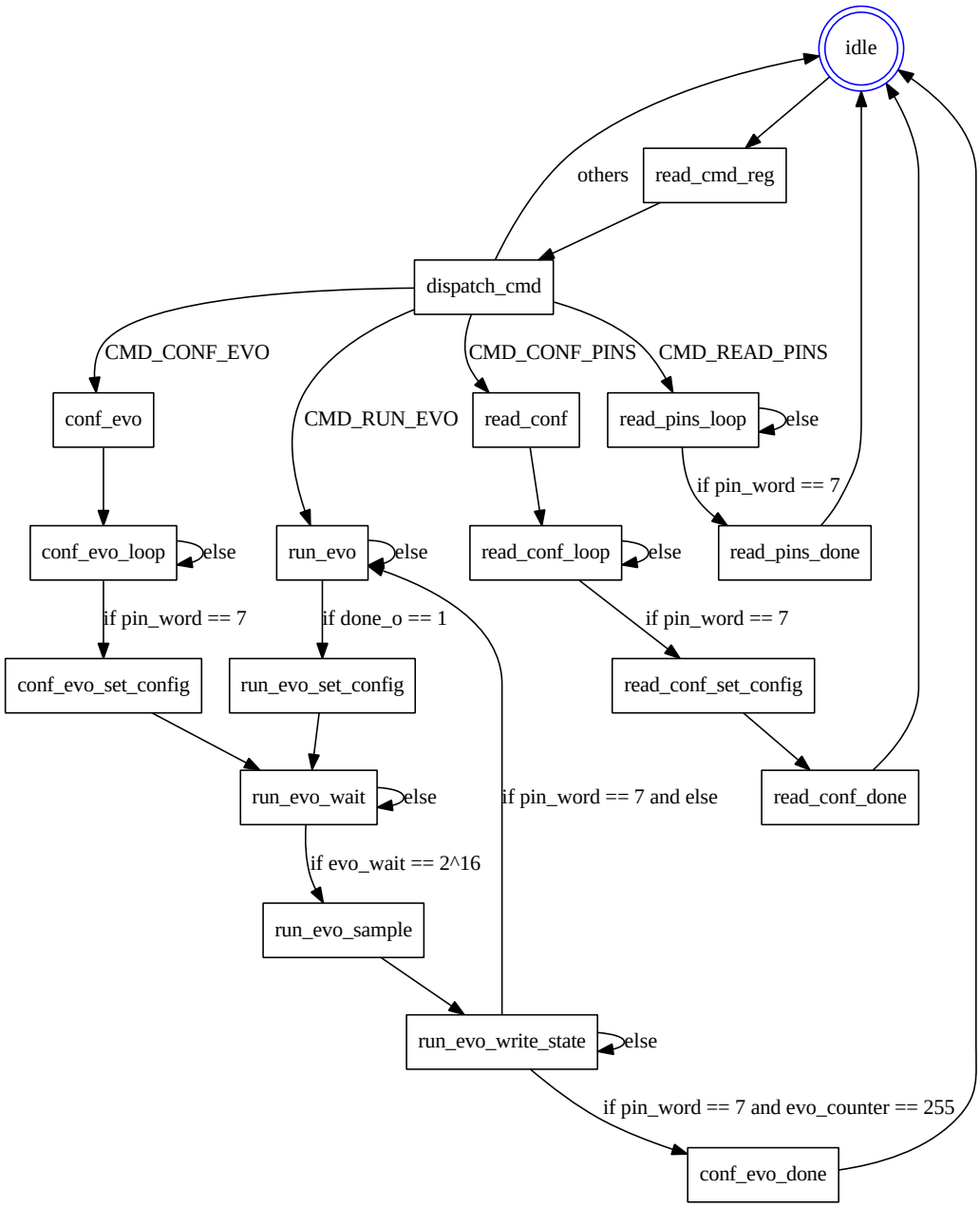


Figure 4.7: Command control state diagram

4.1.3 Pin controller

The pin controller abstracts the configuration of the material. It has a low level control block for each pin, that is 64 low level controllers. Each of these controllers have 2 bits assigned to it as seen in figure 4.8. The output of pin controller is 128 registered bits, where the mode bits are never changed, while the pin value bits is from sampled data in the low level pin controllers.

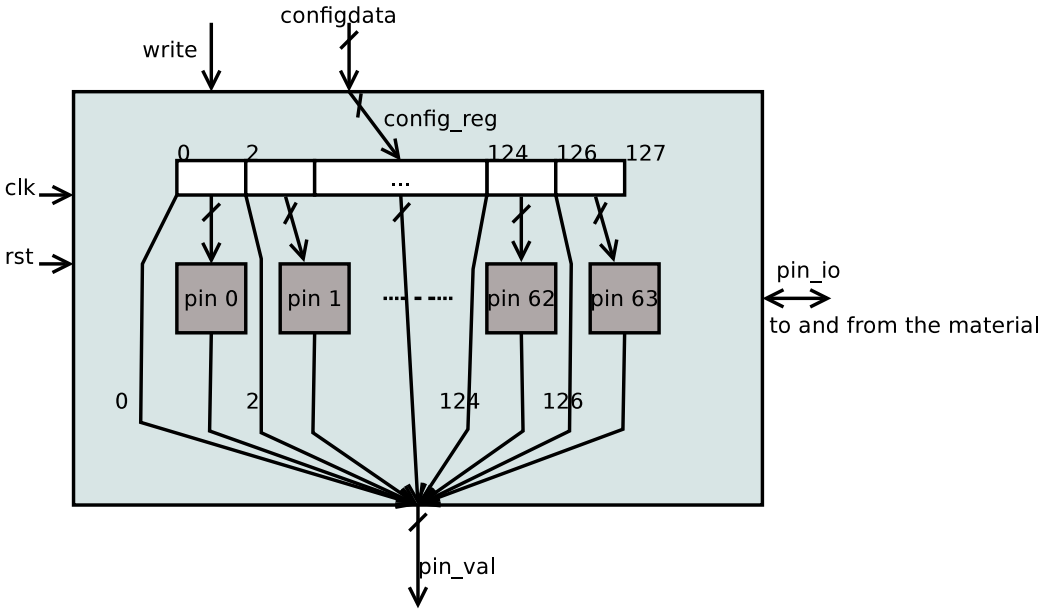


Figure 4.8: Pin control

The low level controllers can be seen in figure 4.9. It consists of a tri-state buffer and two two-register synchronizers, the synchronizers protect the system from metastability, that is when a signal is sampled in a transition between high or low it is arbitrary what the signal will settle on, it could become high or low. By using several chained registers to protect the system from using a metastable signal, assuming by the time the signal has gone through all the registers it has settled on either high or low. Usually two or three registers are used, increasing the numbers beyond that has very small gains. This is unchanged from the old Mecobo platform.

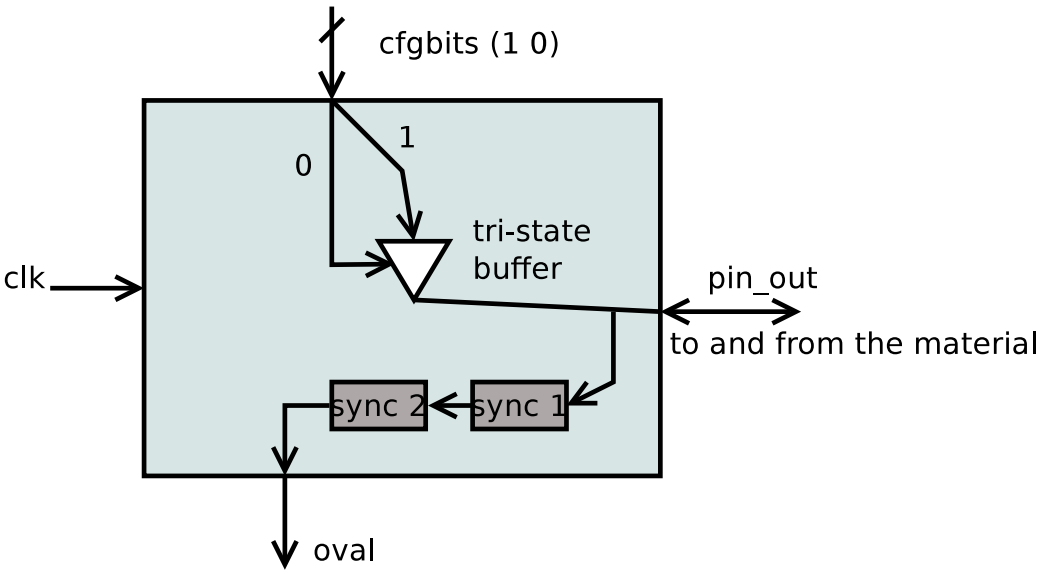


Figure 4.9: Pin

4.1.4 CA CPU

The CA CPU is the new extension which takes the programs supplied from the host computer (the CA rules) , and calculates a new state of the complex system based on the old state and the CA rules supplied. This new state is then used to configure the pin controller and the value bits of the pins which is configured to be driven from the material is updated and then the complete state is written to CA state memory. The update process is described in section 3.2 and more specifically equation 3.9.

The overview of the CA CPU can be seen in figure 4.10. Inside the CA CPU there is an array of cores or small CPUs, with 1-bit datapath which together calculates the new state. Each core controls one bit in the state. An idea is to extend this even further by adding functionality so the CA rules or programs can reconfigure the mode bits in the CA state. The memory signals comes from the microcontroller interface module, the configuration is supplied from the command control module. The command control module controls the CA CPU with the core reset signal.

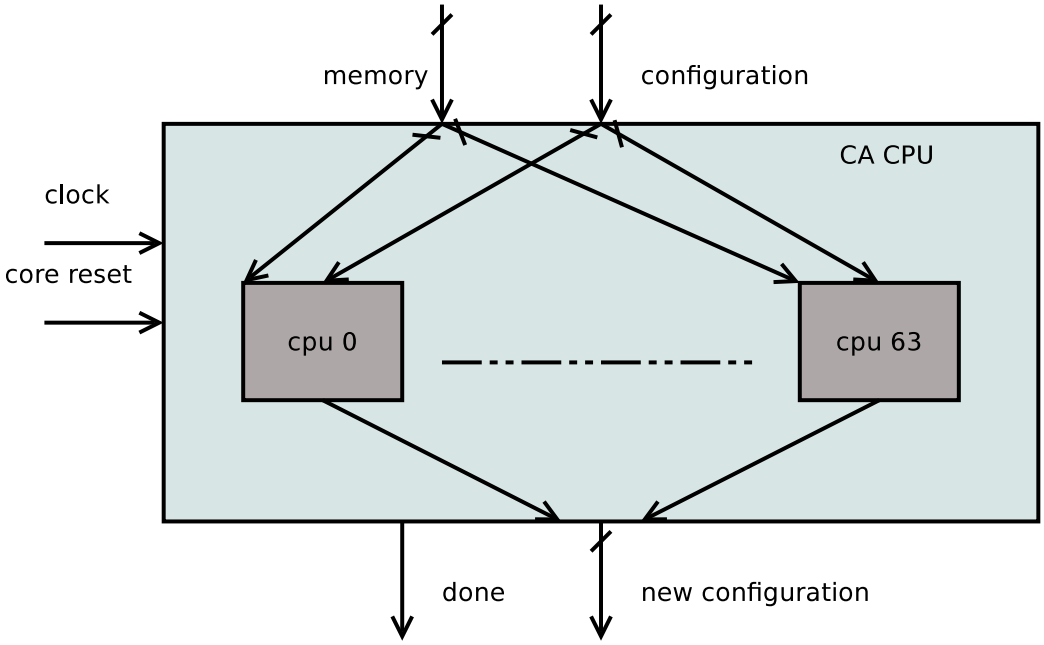


Figure 4.10: CA CPU

In appendix A there is a complete description of the CA CPU. Section A.3 describe in detail the modules and the pipeline. The implementation details and ISA are described in section A.5. Simulation is described in section A.10.

4.2 Microcontroller

Most of the base code is not changed from the last version of the Mecobo platform. It still uses the Atmel driver framework, which resides in the *src/uc3_fw* directory.

The hardware is configured and initialized in *src/uc/setup_hw.c*. The power manager is configured with 3 different clock domains. The CPU clock, Peripheral Bus A(PBA) and the USB domain. The CPU clock is initialized to 16MHz while PBA is 4MHz. The USB clock is handled by the Atmel driver framework and configured to be 48MHz.

The communication between the host computer and the microcontroller is over a virtual com port which is much easier to use than a USB driver. The microcontroller identifies itself as a USB CDC device, more specifically an ACM modem, which has a serial interface. The communication then works as a FIFO buffer for both ends. Pushing 1 byte at the time. The implementation of this is in *src/uc/datalink.c*.

The communication with the FPGA is by utilizing the External Bus Interface(EBI) [30, p. 145] and the Static Memory Controller(SMC) [30, p. 366]. The FPGA is connected to the lines associated with chip select line CS1. So the FPGAs memory mapped address space starts on address 0xD0000000. The EBI is configured in *src/uc/conf_ebi.h* and the SMC is configured in *src/uc/fpga_smc.h*. The SMC timings are changed and optimized. The timings for a read cycle can be seen in figure 4.11, where CLK_SMC is the clock, A is the address bus, NRD is the read enable signal, NCS is the chip select signal and D is the data bus. The write cycle is in figure 4.12, where NWE is the write enable signal and the other lines are the same.

The read cycle is set up as NRD controlled [30, p. 374] which means the SMC expects the data to be read when NRD goes high. For the write cycle, it is similar, it is NWE controlled [30, p. 378] which means the cycle is controlled by NWE and it expects the data to be written when NWE goes high, the data bus is held until NCS goes high again for good measure.

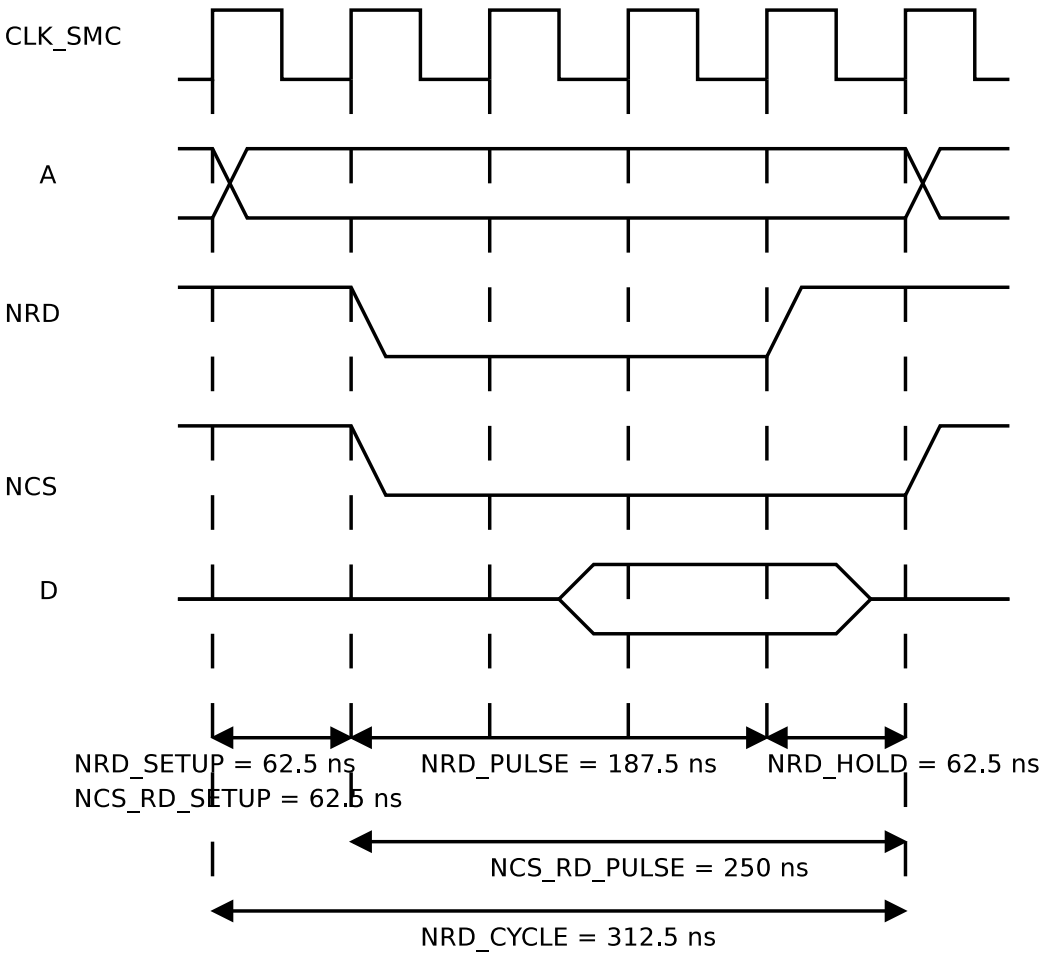


Figure 4.11: Read cycle of the microcontroller

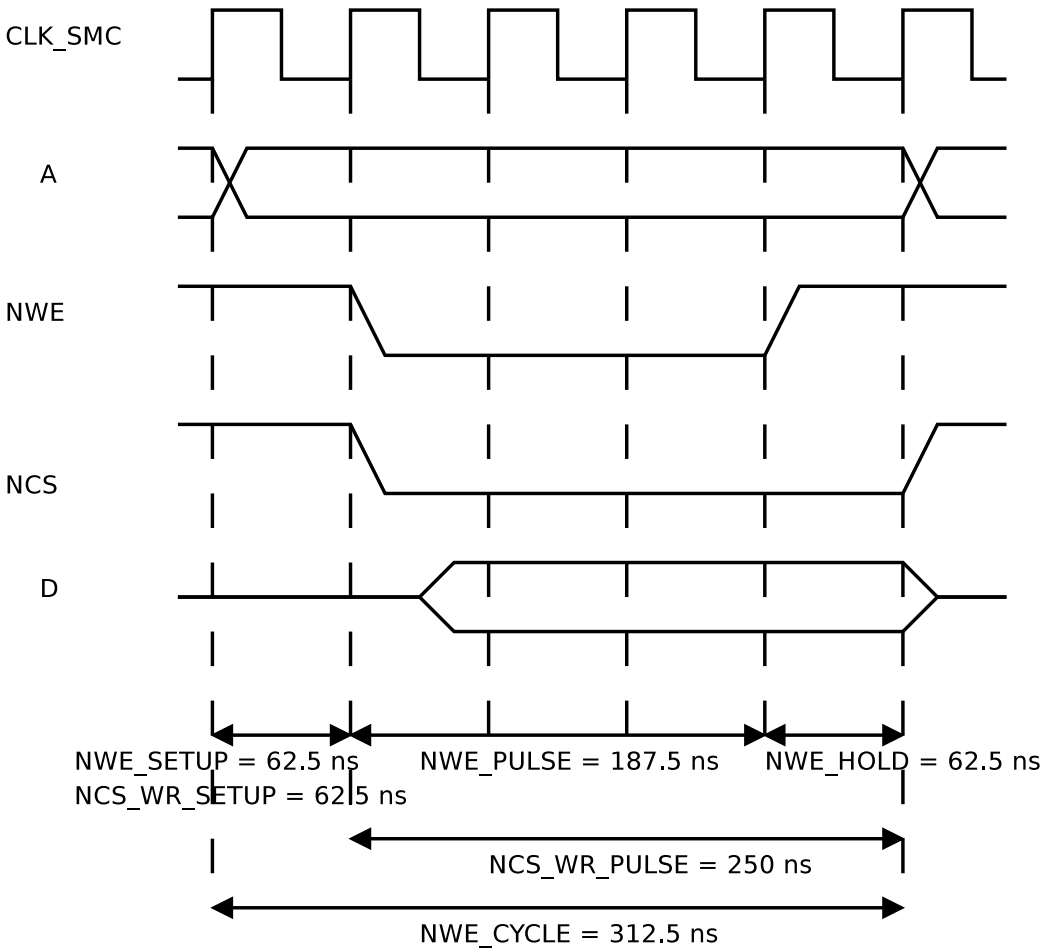


Figure 4.12: Write cycle of the microcontroller

The microcontroller's firmware is now using a new memory API for accessing the FPGA. This new API is defined in *src/uc/fpga_mem.h* shown in listing 4.1, the rest is implemented in *src/uc/fpga_mem.c*.

This new API forces the accesses to be half-word(16-bit) aligned accesses, so it does not depend on the compiler optimizations whether it will work or not. Note that the all the functions but "fpga_mem_readh" and "fpga_mem_writeh" uses byte offsets and not half-word offsets for address. So when using functions that require byte offset like "fpga_mem_copyto" multiply the offset by 2.

Two utility scripts are included, one for the build process *src/uc/build.sh* and one for debugging purposes *src/uc/avr32debug.sh*. The build script is a hack,

Listing 4.1: FPGA memory API

```
1  #ifndef FPGA_MEM_H_
2  #define FPGA_MEM_H_
3  #include <stdint.h>
4  #include "command.h"
5  uint8_t fpga_mem_readb(uint16_t byte_offset);
6  void fpga_mem_writeb(uint16_t byte_offset, uint8_t data);
7  void fpga_mem_copyfrom(void *dest, uint16_t byte_offset, ↩
       uint16_t nbytes);
8  void fpga_mem_copyto(uint16_t byte_offset, void *src, ↩
       uint16_t nbytes);
9
10 static inline uint16_t
11 fpga_mem_readh(uint16_t offset)
12 {
13   volatile uint16_t *ptr = (uint16_t *)FPGA_ADDR;
14   return ptr[offset];
15 }
16
17 static inline void
18 fpga_mem_writeh(uint16_t offset, uint16_t data)
19 {
20   volatile uint16_t *ptr = (uint16_t *)FPGA_ADDR;
21   ptr[offset] = data;
22 }
23 #endif
```

because after programming the microcontroller on the Mecobo platform it would not reset correctly so what it does is, programming it then reset and then jump to the address of the main function.

The debug script is for starting up the debugging of the microcontroller on the hardware. It uses avr32-gdb-proxy and the remote functionality of gdb to debug the microcontroller over JTAG.

4.3 Host software

The communication with the microcontroller and data structures for CA programs and behavior data is bundled in the EMB library, libEMB. libEMB is in the directory *src/libemb*. The new additions in libEMB includes *ca_program.c/h*, *ca_state.c/h* and the function evo_config implemented in *emb.c/h*, these additions are data structures for holding state data, assembled programs and functions for sending and receiving data to and from the microcontroller.

The function evo_config works as follows:

1. Allocate space for the packet to be sent and CA state data
2. Build the packet from the CA program structures
3. Create the CMD_CONF_EVO packet
4. Send the packet and free its data structures
5. Wait for the OK response
6. Receive the packet with the CA state data
7. Return the data structure of the CA states data

CA programs is implemented in python scripts in the *src/host/* directory. The programs are converted to runnable programs with the assembler written in python (*src/host/sasm.py*). Listing 4.2 is an example of such a script which sasm takes as input. This particular script is for the ECA rule 90.

The function "ne(core_num)" is the neighbourhood function. It returns a tuple containing the neighbourhood of the given cell or core number. The "prog" list is the list of instructions or program to be assembled. The parameter to the "asm_in" object is used as lookup in the neighbourhood tuple. So "asm_in(0)" will assemble into an instruction which takes the cell's 0th neighbour as input, which is defined by the function "ne".

The initial_config variable is the initial material configuration. In this case there is only a single bit set, which is an odd numbered bit, that is a value bit. Even numbered bits are the mode bits, where 1 indicates the pin is externally driven and 0 indicates that it is FPGA driven.

Some commands that the Mecobo platform support are not tested as well after the changes and is thus left out of the table. Table 4.2 shows the commands of the previously described functionality. The two new commands added for the new extension is CMD_CONF_EVO and CMD_RUN_EVO.

Listing 4.2: Rule 90 program script

```
1  def ne(core_num):
2    if core_num == 0:
3      return (core_num+1, 31)
4    elif core_num == 31:
5      return (0, core_num-1)
6    else:
7      return (core_num+1,core_num-1)
8  prog=[asm_in(0),\
9    asm_in(1),\
10   asm_xor(),\
11   asm_out()]
12 initial_config='\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\↩
     x00\x00\x00\x00\x00\x00'
```

The program *src/libemb/program_fpga.c* uses libEMB to send a FPGA bit file in a CMD_CONF_FPGA packet.

To assemble a program, run the command:
./sasm.py uniform-1
Where "uniform-1" is the basename of the program to assemble, there must be a program script named *uniform-1.py* for this to work.

Assembled programs can be run by using the run_program program in *src/libemb/run_program.c*. It takes in one argument which is the basename of the program.
Issuing the command:
run_program uniform-1
It will read in all of the assembled program (each of the cores has its own corresponding program in a file, named *src/host/uniform-1-NN.bin*, where NN is a number from 00 to 31), as well as the initial configuration in *src/host/uniform-1.init* and sends it to the microcontroller in a CMD_CONF_EVO packet.

When the FPGA is done the microcontroller will read out all the CA states from the CA state memory and send it back to the host computer and write them to a file.

libEMB	value	Description
CMD_SEND_PATTERN	0x1	Set the pins to the material.
CMD_READ_PATTERN	0x2	Read the pin values to the material.
CMD_CONF_FPGA	0x3	Upload configuration bit file to the FPGA.
CMD_RESPONSE	0x4	Response from microcontroller.
CMD_CONF_EVO	0xB	Configure and run the CA CPU and material with supplied initial state.
CMD_RUN_EVO	0xC	Run the CA CPU and material with from the current state.

Table 4.2: Commands supported by libEMB

Chapter 5

Experiments

This chapter describes the initial experiments run on the new Mecobo platform and the results. Each experiment has its own section.

Each experiment consists of a CA model or program which the cores in the CA CPU runs to updated the material configuration for each round. The programs, or mathematical functions executed by each core is generated by setting up each rule's function table as minterms and performing boolean simplification. Some of the programs in this chapter could possibly be simplified even more. All programs must end with an "out" instruction.

The program along with the neighbourhood function must be defined in the program's python file. The assembler converts the program into runnable programs for each core.

To generate the instructions for each core the programs must be assembled. From command line:
cd src/host
./sasm.py rule_NN.py
This generates sets of instructions for the programs, one for each core in the CA CPU and a file with the initial configuration. The files generated is *rule_NN.init* which is the initial configuration and the core programs is in the files *rule_NN-nn.bin*.

The example program script given in listing 4.2 is the rule 90 program script.

Results
Each experiment has two different types of result figures associated with it, one type is the space time plot shown for rule 90 in figure 5.2, the other is a

state space plot shown for rule 90 in figure 5.4.

The space time figure is similar to what is seen earlier in figure 2.6a, the first line is the initial configuration, each diamond represent a cell. The red cells are cells connected to FPGA driven pins, the blue cells are driven externally or from the material. Figure 5.14 is the only experiment run which uses externally driven cells.

The state space plots shown i.e. for rule 90 in figure 5.4, shows the trajectory of the behavior through state space. This particular figure shows that it settles in an attractor. However from the plots it might be hard or impossible to actually see what state-value it settles in. The state-value can also be seen from the space time figures like 5.2 when it starts repeating itself. In the state space plots the FPGA driven pins are the only ones encoded in the state value. However for some problems and materials it will most likely be interesting to see the state space of the material driven pins as well, but for these initial experiments it would contain little information.

5.1 Rule 90

This program is generated from the script *src/host/rule_90.py*. Rule 90 is one of the simpler ones, each cell executes the function $f(L, R) = L \oplus R$, where L is the left neighbour and R is the right neighbour. The instruction tree is shown in figure 5.1, notice the "in" instructions for each neighbour, since this is a stack machine all operands has to be fetched and pushed onto the stack before executing an operation(xor in this case). The two "in" instructions are executed first, then the "xor" and finally "out" which indicated the core is done.

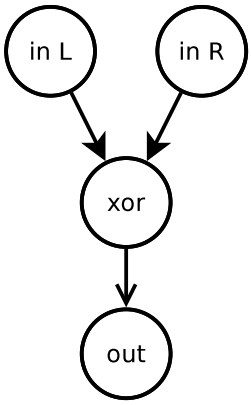


Figure 5.1: The rule 90 program visualized

5.1.1 Results

Figure 5.2 and 5.3 shows the general behavior of the rule 90 modeled on the Mecobo platform. It is very easy to visually confirm that this is in fact the right behavior of the rule 90 program. This experiments also shows how the circular neighbourhood function works, the missing piece on the right side of the structure is appearing on the left side, before it settels in an all 0 attractor.

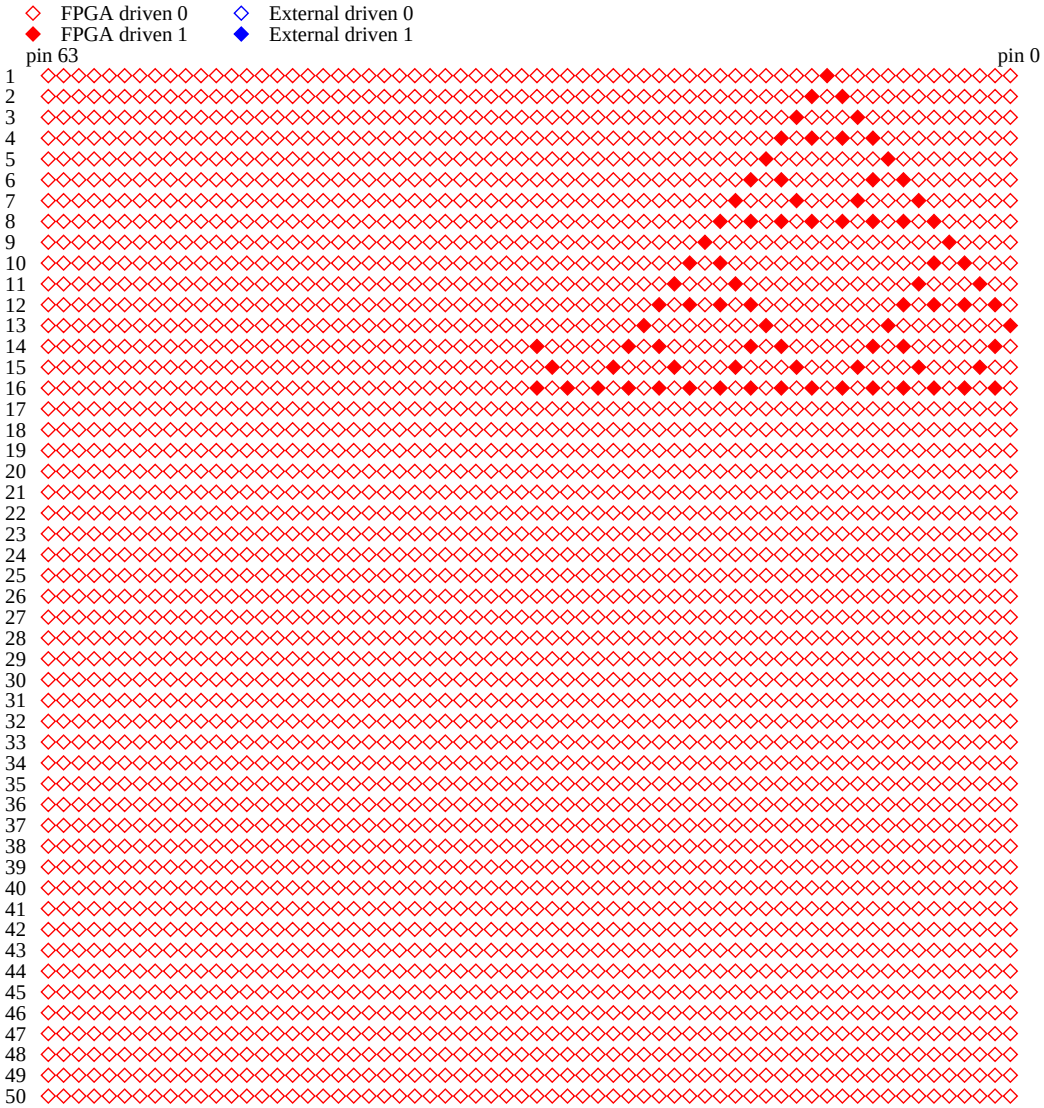


Figure 5.2: Results from rule 90

The figure 5.3 is redundant for rule 90, because it reaches an attractor long before the 51 time step, but it is shown for completeness. The CA does not exit the attractor.

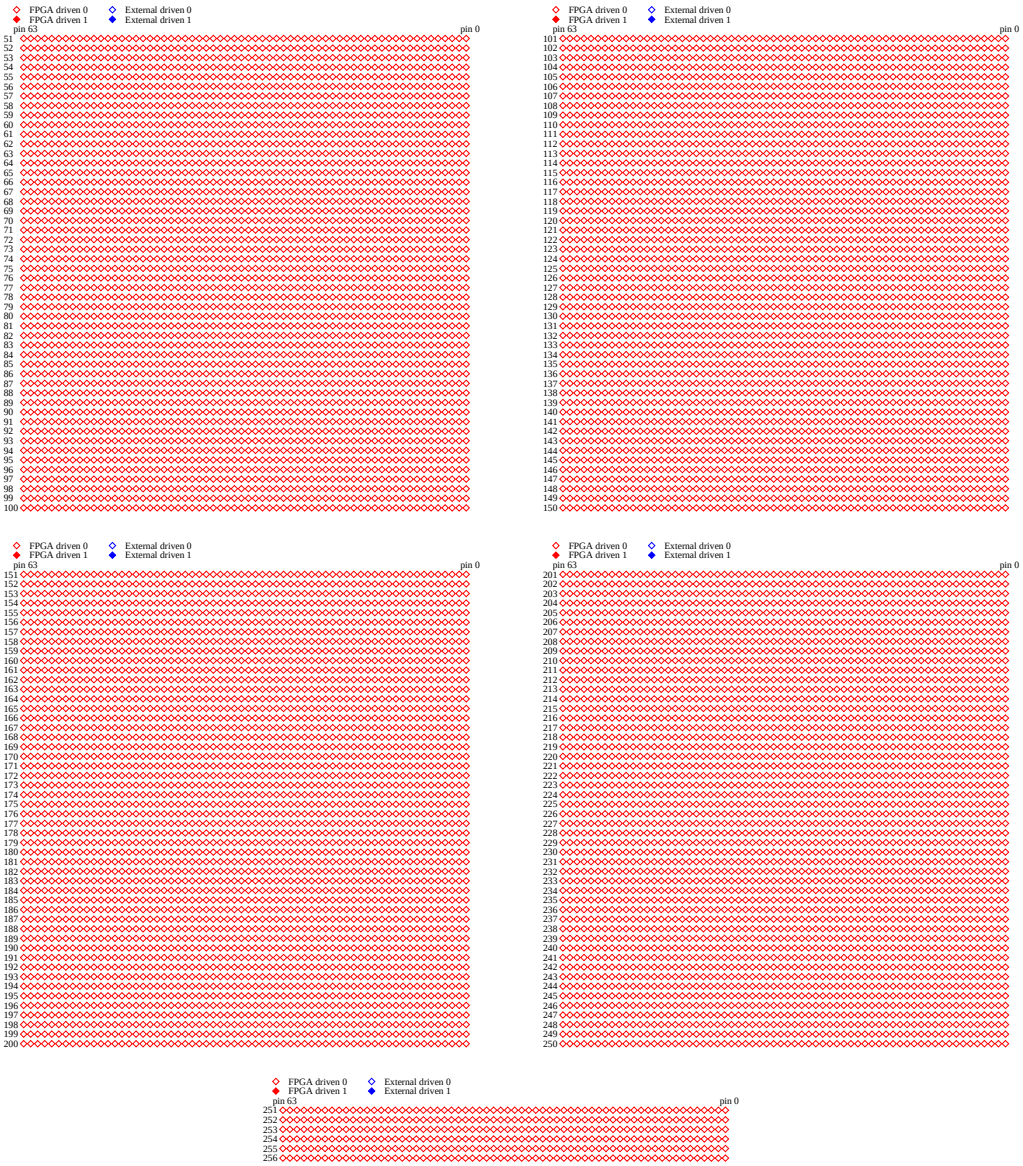


Figure 5.3: Rule 90 result states from 51 and onwards

Figure 5.4 shows the trajectory of rule 90 through state space before it reaches the attractor. It takes 16 steps for it to settle in the attractor. The red node represents the initial state.

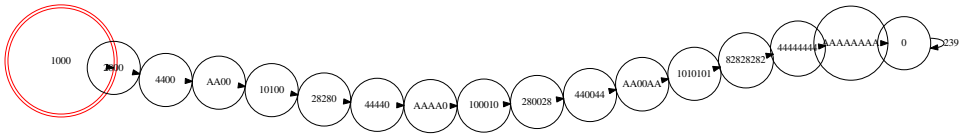


Figure 5.4: Rule 90 state space of FPGA driven pins

5.2 Rule 110

This program is generated from the script *src/host/rule_110.py*. In rule 110 the function each cores executes is: $f(M, R, L, M, R) = M \oplus R \vee ((\neg L) \wedge M \wedge R)$, where "M" is the middle neighbour (old state of the cell), "R" is right neighbour and "L" is left neighbour. The instruction tree is shown in figure 5.5.

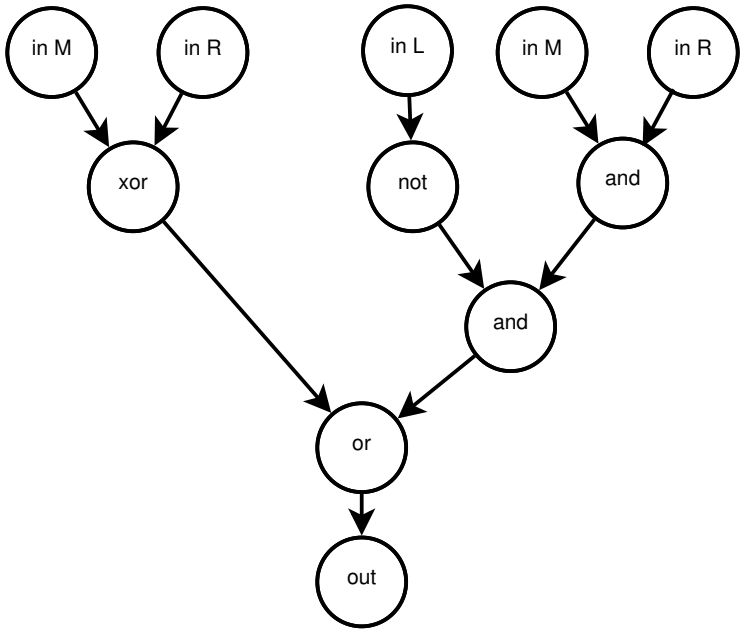


Figure 5.5: The rule 110 program visualized

5.2.1 Results

Rule 110 has a complex behavior. ECA rule 110, was proven to support universal computation [21]. As can be seen by figure 5.6 and 5.7 it does indeed

exhibit more complex behavior than the other experiments i.e. rule 90. Note that also this rule's experiment was run with a cyclic neighbourhood.

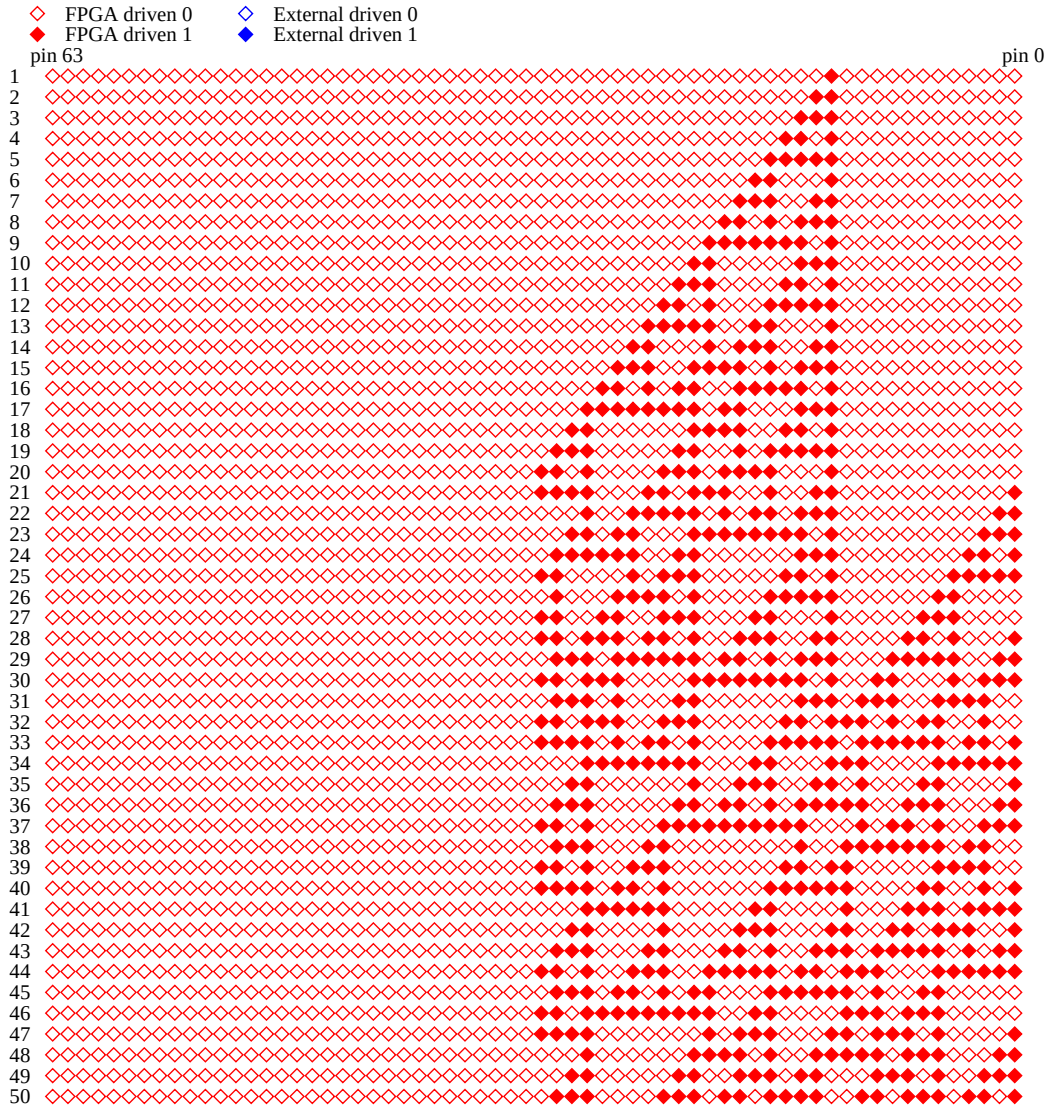


Figure 5.6: Results from rule 110

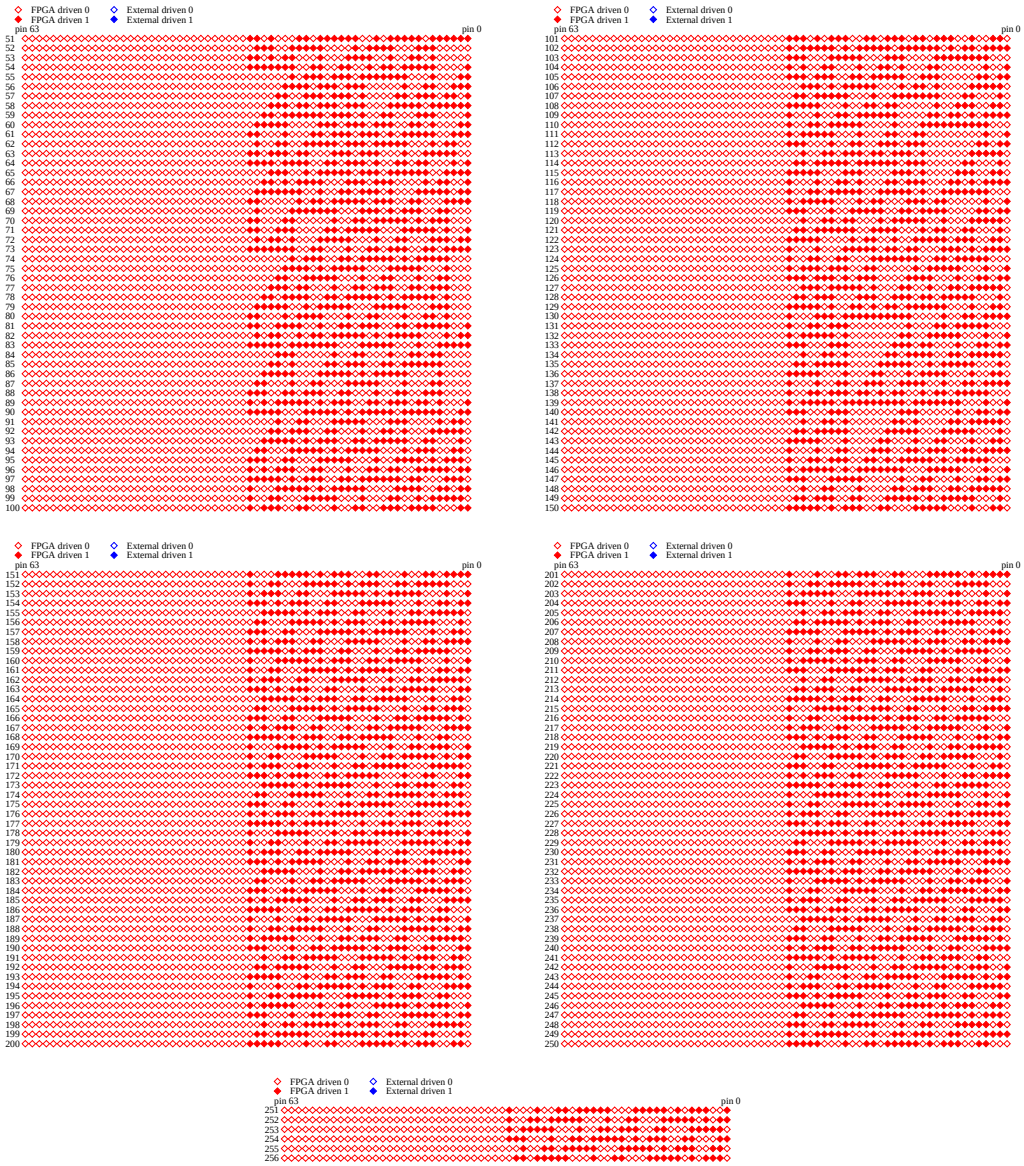


Figure 5.7: Rule 110 result states from 51 and onwards

It is hard to see from figures 5.6 and 5.7 that rule 110 settles in an attractor, but that can be clearly seen from figure 5.8, where it ends up in a cyclic attractor shown in the bottom left of the figure. The red node represents the initial state.

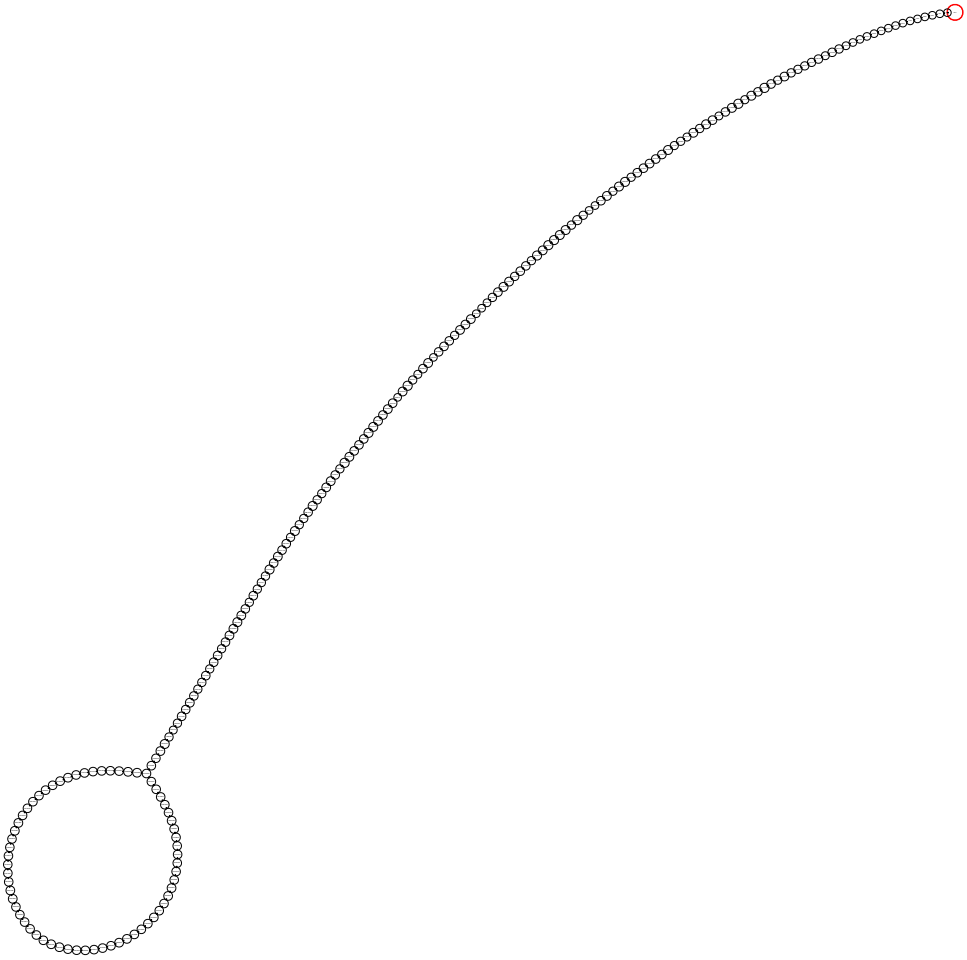


Figure 5.8: Rule 110 state space of FPGA driven pins

5.3 Rule 225

This program is generated from the script *src/host/rule_225.py*. The function it executes is: $f(R, L, M, L, R) = ((\neg R) \wedge (\neg(L \oplus M))) \vee (L \wedge R)$, where "M" is the middle neighbour (old state of the cell), "R" is right neighbour and "L" is left neighbour.

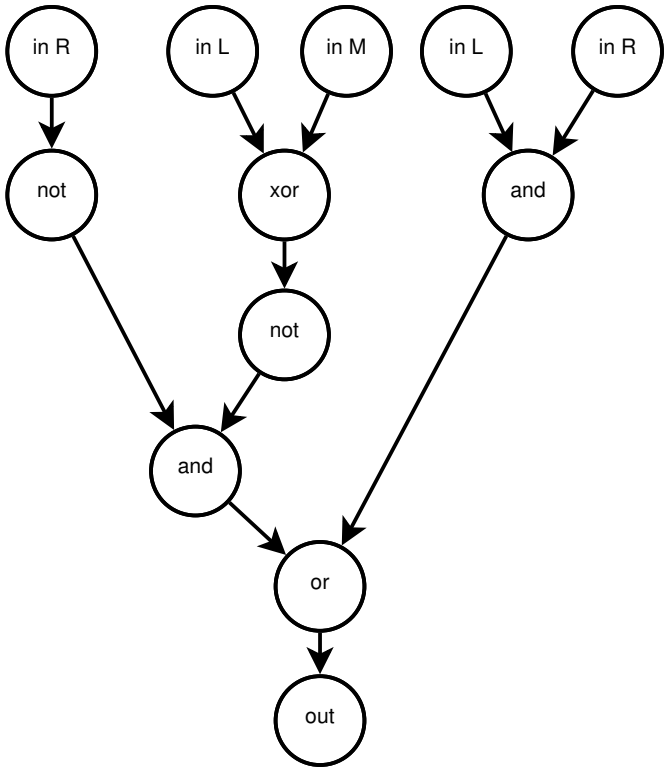


Figure 5.9: The rule 225 program visualized

5.3.1 Results

Rule 225's behavior can be seen in figure 5.10 and 5.11. As can be seen in the figures, a negative pattern (looking at the 0s instead of 1s) grows to the right continuing until it reaches the boundary and then appears on the left side. It is hard to tell from these figures how the trajectory is.

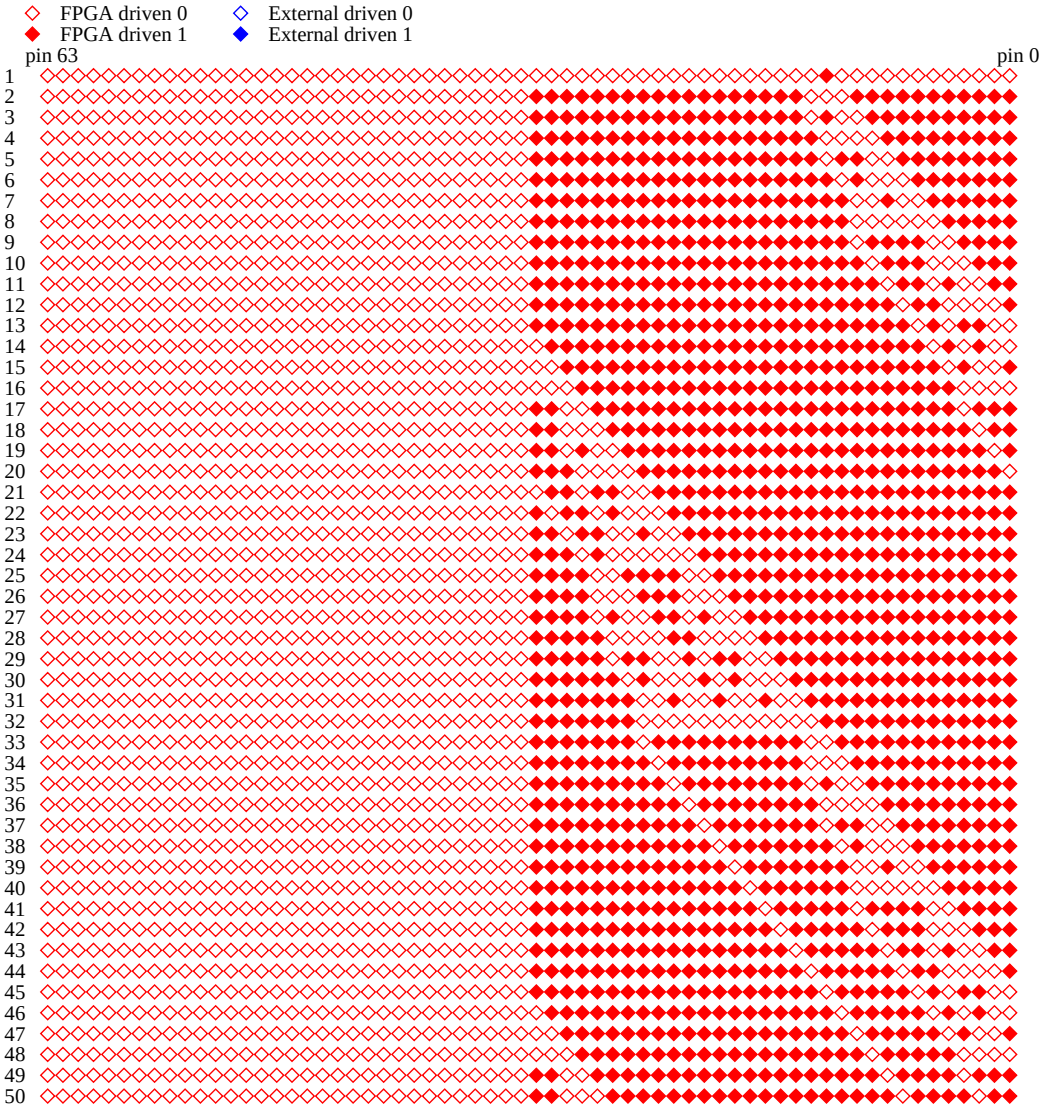


Figure 5.10: Results from rule 225

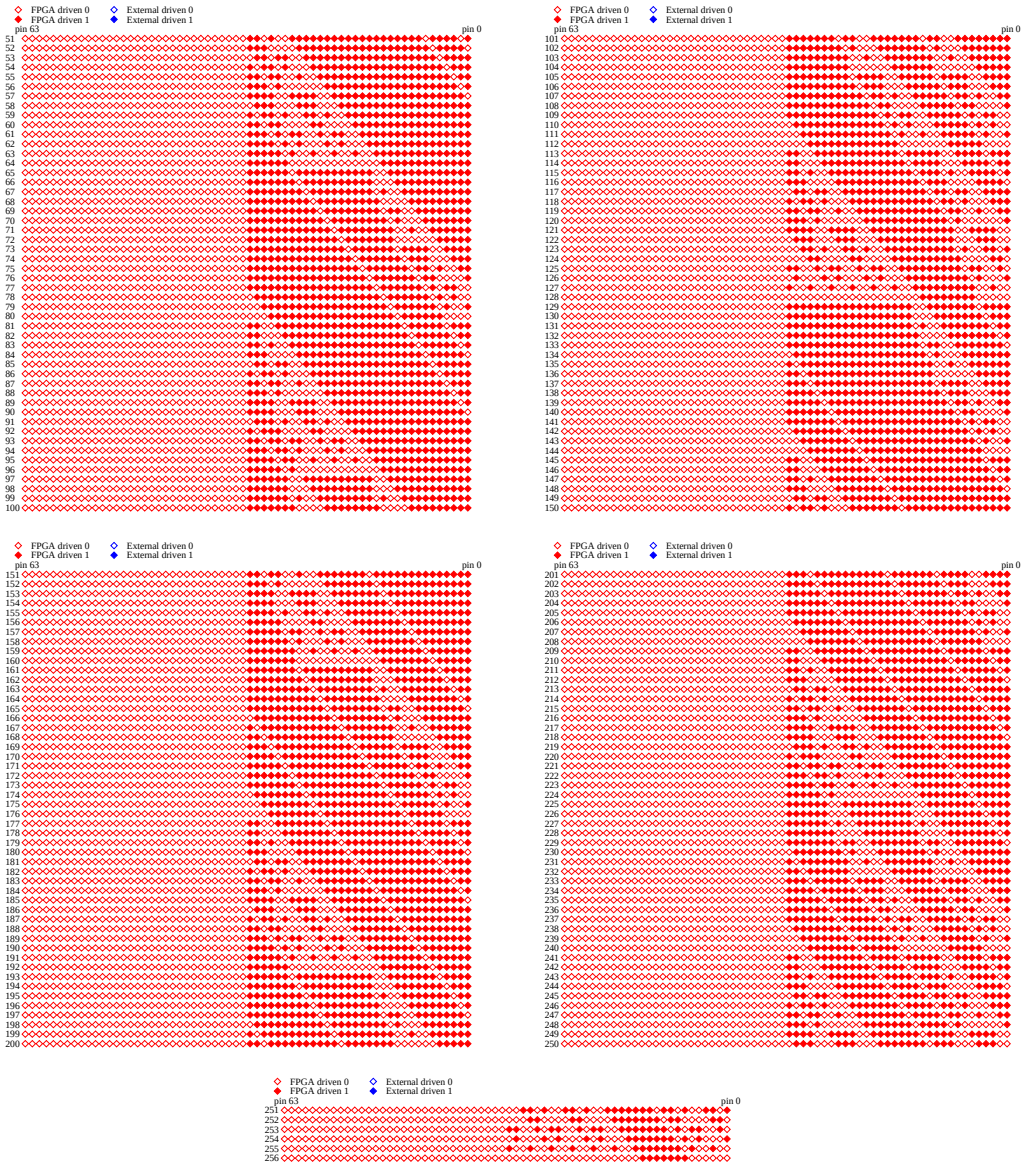


Figure 5.11: Rule 225 result states from 51 and onwards

Figure 5.12 shows a clearer picture of what happens in figure 5.10 and 5.11. The initial state is red and on the right side. The trajectory seem to be endless, at least for the first 256 samples. However there is 32 bits in use, so the CA have 2^{32} different states and only 256 samples is taken which is a very small subset of the possible states. It is not certain that the CA will end up in an attractor.

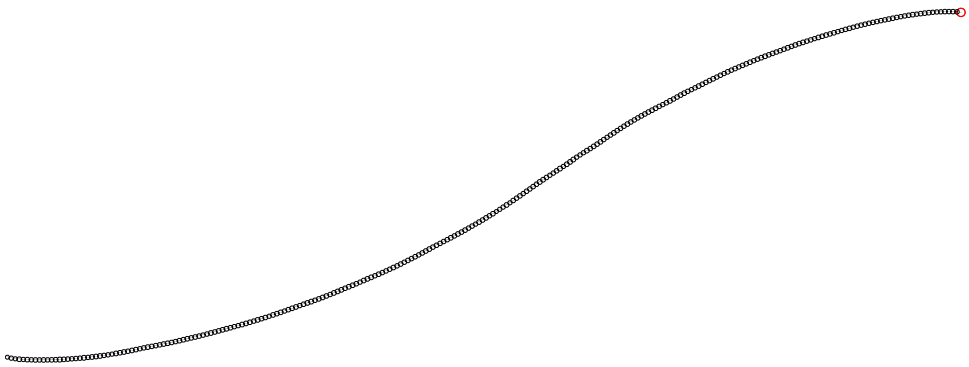


Figure 5.12: Rule 225 state space of FPGA driven pins

5.4 Uniform 1

This program is generated from the script *src/host/uniform-1.py*. Uniform 1 is a little different than the previous programs, because it does not only model a CA it also uses the output of the material. For this experiment the output of the material is known to be only 1s. The function it executes is: $f(O, E) = O \oplus E$, where "O" is the old state of the cell and "E" is external, or output from the material. Figure 5.13 shows the instruction tree of the program. Both old state and output from the material is taken as inputs, it then xors the values and outputs the answer.

The "E" or external value is taken from the cells respective external pin(defined for this experiment). The neighbourhood is defined such that, there is two inputs to each cell, one is the old state of the cell, the other is the cells respective material pin. So for cell 0, "E" is the 31 pin, while the "O" is 0. For cell 1, "E" is the 32, while "O" is 1 and so on.

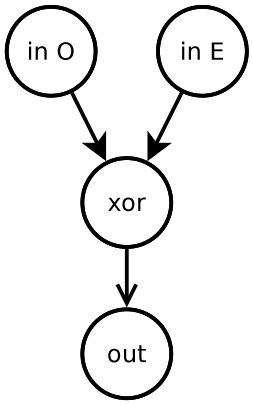


Figure 5.13: The uniform 1 program visualized

5.4.1 Results

This is the only experiment run with input from a material. The material is hardwired, that is fixed response, all 1s. Shown in figure 5.14 and 5.15, the material's response is on the left side, blue colour. The program run is $f(O, E) = O \oplus E$, which is the cells old state xored with the cells respective material pin. That is the far right cell (pin 0) uses pin 31 as its "E", while the next cell uses 32 and so on up to 63.

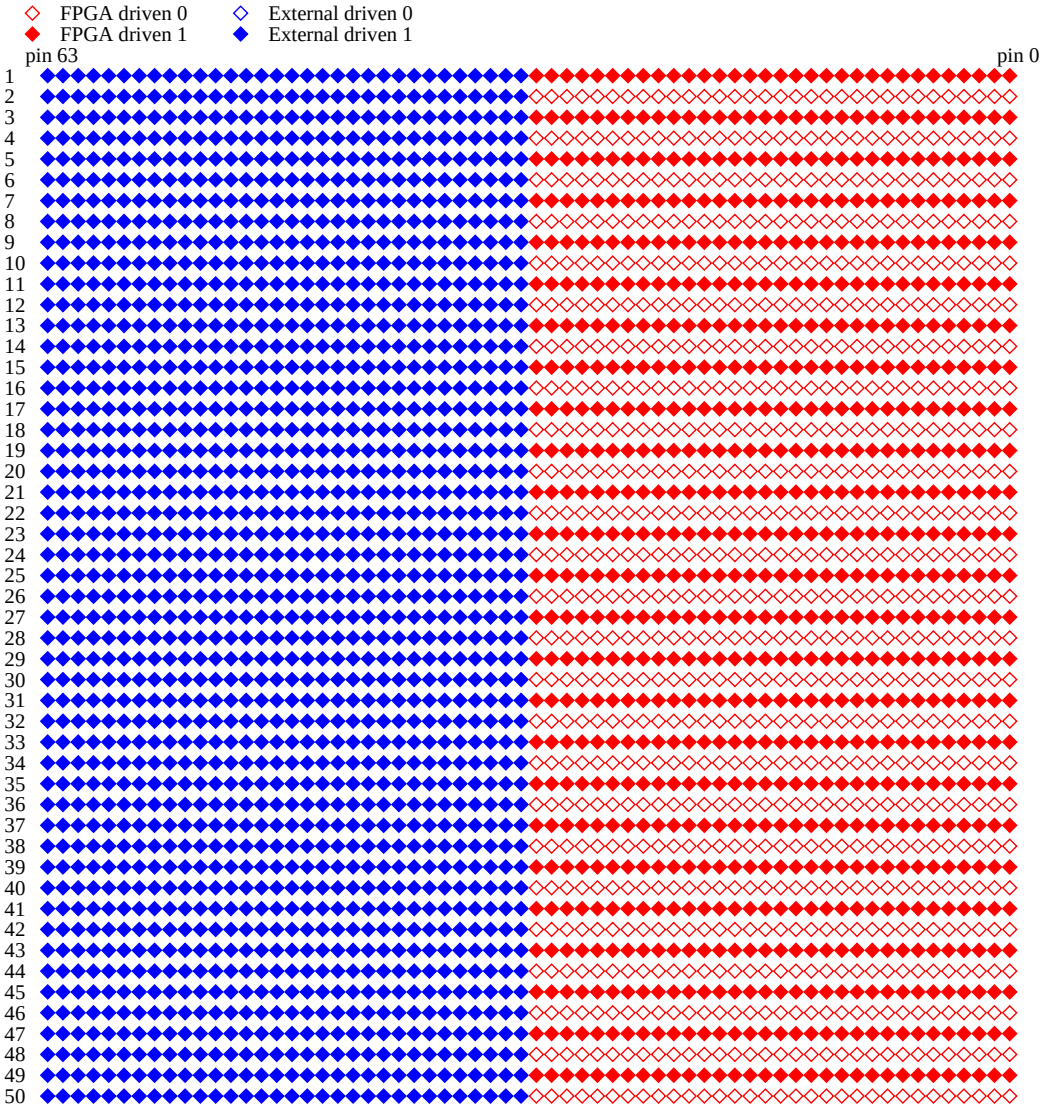


Figure 5.14: Results from uniform 1

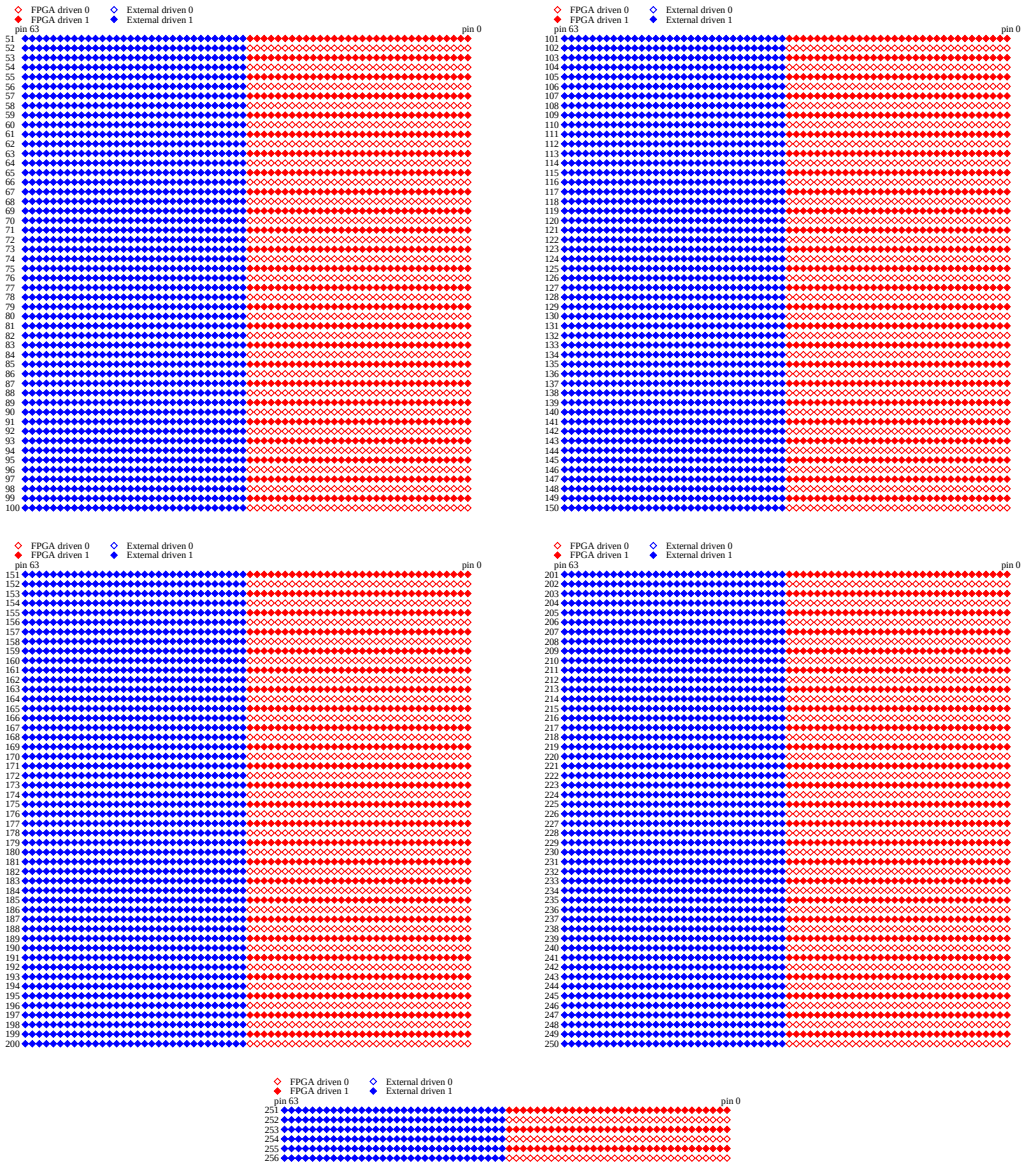


Figure 5.15: Uniform 1 result states from 51 and onwards

The state space figure 5.16 is of no interest since the program is far too simple, but it is included for completeness. It shows that the system oscillates between two states, state ”FFFFFFFF” and ”0”.

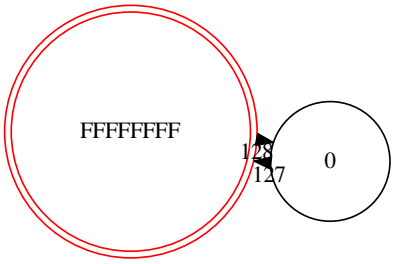


Figure 5.16: Uniform 1 state space of FPGA driven pins

Chapter 6

Discussion

Even with the new extension, Mecobo is built on fairly new concepts and likely require multiple iterations of design and implementation before a better system for doing experiments on materials is found. During the development there were several difficulties, most notably in the communication between the host and microcontroller over USB, and between the microcontroller and the FPGA, which took a long time to figure out. The interfacing between the microcontroller and the FPGA was rewritten as a result and some bugs was corrected in the USB communication.

The initial experiments described and executed in this master's thesis is only a start and is nothing more than a very small set of experiments the Mecobo platform is capable of executing. The Mecobo platform is capable of much more advanced programs and CA models, some of which are described in the chapter 3.2.

6.1 Conclusion

The Mecobo platform is extended with the CA CPU which enables the Mecobo platform to experiment with dynamic complex systems based on a hybrid CA material model. Most of the modules in the FPGA is rewritten to be able to use the new extension. On the software side, both the microcontroller firmware and libemb is extended with functions to allow a user to program the CA CPU and run experiments with the hybrid CA material system. The microcontroller firmware is also extended with a new memory API towards the new memory interface in the FPGA.

The system is tested and confirmed to be working with several CA rules and a CA in conjunction with an theoretical unconventional material. The material was hardwired to have constant response, to be able to verify that the system was working correctly.

A mathematical model is presented in section 3.2 which can model the updating process and behavior of many experiments which can be run in the hybrid CA material system of Mecobo. However if the properties and behavior of the material is unknown the model is incalculable, and only serves as a specification of how the updating process of each state is executed.

6.2 Future Work

The new extension CA CPU is using a simple core featuring just enough functionality to execute boolean functions. It could be interesting to extend the CA CPU to include more advanced functions. There is two main extensions which could be useful, first one is being able to modify the mode bit in the configuration and the second introducing jump, status flags and compare instructions in the cores.

Before trying to exploit any hybrid CA material system to solve a specific problem it is likely necessary to find a material which exhibit behavior indicating that it supports information transmission, storage and modification. These properties is needed to support universal computation. To do that, one could possibly approach the problem similarly to how Langton did [6], looking at the entropy of the system and quantifying the mutual information in the system's behavior. If the material seem to exhibit or support complex behavior, a GA with an appropriate fitness function may find CA programs and an initial configuration which solves the problem.

Other tasks include code cleanup in libemb, the microcontroller firmware and the FPGA.

Appendix A

CA CPU

The CA(Cellular Automata) CPU core pipeline was originally based on a general load store CPU pipeline which was made in the course "Hardware Construction". The architecture had a 16-bit RISC ISA(Instruction Set Architecture). However it quickly became clear while removing parts which was not necessary, that several key optimization could be done. Since the amount of space on the FPGA is fairly limited the ISA and pipeline was completely redesigned to make it more likely to fit on the FPGA.

The current CA CPU as of this writing, is tailor made for programming cellular automata or DDNs(Dynamic Discrete Networks) for our purpose. It has 64 cores, each with a stack-machine with a 4-stage pipeline and 8-bit RISC ISA. Each core has its own instruction memory. Each core is meant to control 1(2)-bit of the pin configuration. The cores control their respective numbers' pin configuration. That is, core 0 controls pin 0(1), core 1 controls pin 2(3) etc. Remember that the even-numbered bits in the pin configuration is the mode and the odd-numbered is the value. Th CPU does not know whether the values it reads from the configuration bits is coming from the material being tested, or if it is being driven from the FPGA. The cores can be programmed in many ways, but modifying the mode bits is not yet supported.

A.1 Register file vs stack-based architecture

The term "stack-based" here, refers to the fact that instead of a regular register file in the decode step, it has a register file which acts like a stack.

It was hypothesized that 64-cores was too ambitious for a small FPGA like the Spartan 3 XC3S500 PQ208. So it was crucial to make the design as compact as possible. Because of the limited space on the FPGA it was necessary to optimize the amount of hardware synthesized and instruction memory, while still having the ability to function like a CA.

To be able to address the 64-bit configuration signals, it is needed $log_2(64) = 6$ bit. And it would suffice with 2-bit opcode, that is 4 different instructions. So the instructions are 8-bit. There is currently only 3 different type of instructions, so in the future there is room for another "in" instruction to read in mode bits if we would want to support complete reconfiguration.

In theory it could have been designed as a conventional register file based CPU with 6 bits to encode both ALU function, source and destination registers, but it would limit the ALU functions and how many registers an instruction can address too much. Hence to avoid having to encode register addresses in the instructions, the result was a stack-based CPU.

A.2 Multicycle vs Pipelined

In a multicycle design, there is only 1 instruction which is executed at the time. In a pipelined design, the next instruction gets fetched as soon as the instruction before it is done in the fetch stage, in this case the pipeline can have up to 4 instructions executing at the time. One in each pipeline stage.

The two methodologies differ is a trade-off between size of hardware and processing speed. The goal is to minimize the CPI(clocks per instruction) without adding too much hardware. By exploiting the instruction level parallelism the CPU can achieve nearly $CPI = 1$. Some architectures take this even further by issuing and retiring multiple instructions at the same time, effectively reducing the CPI more ($CPI < 1$), these architectures are also called superscalar.

The CA CPU multicycled version can be seen in figure A.1. Notice how the hardware is significantly simpler than the piplined version in figure A.2. There is no hardware to control forwarding, and the control unit is also simpler, because there is only 1 instruction in the whole CPU at the time. While the pipelined version can have an instruction in fetch, decode, execute and writeback simultaneously.

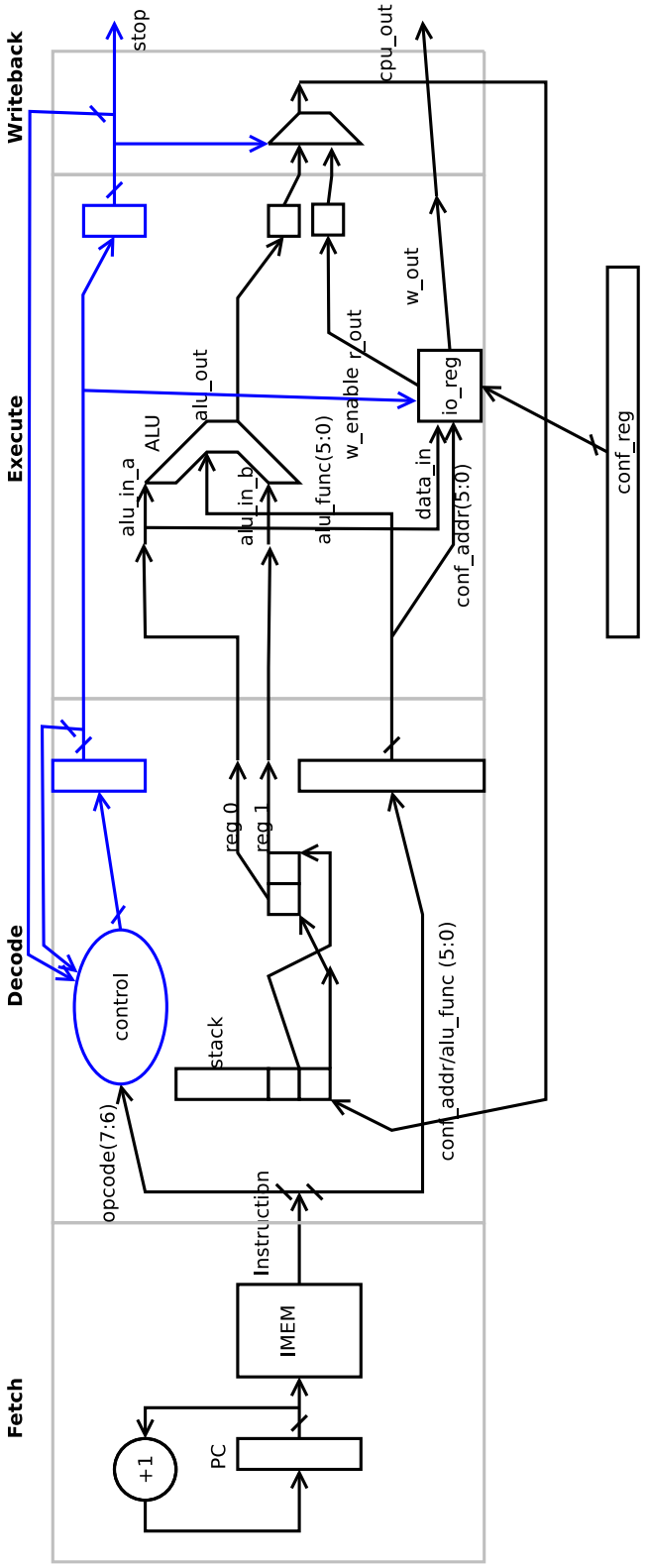


Figure A.1: Multicycle

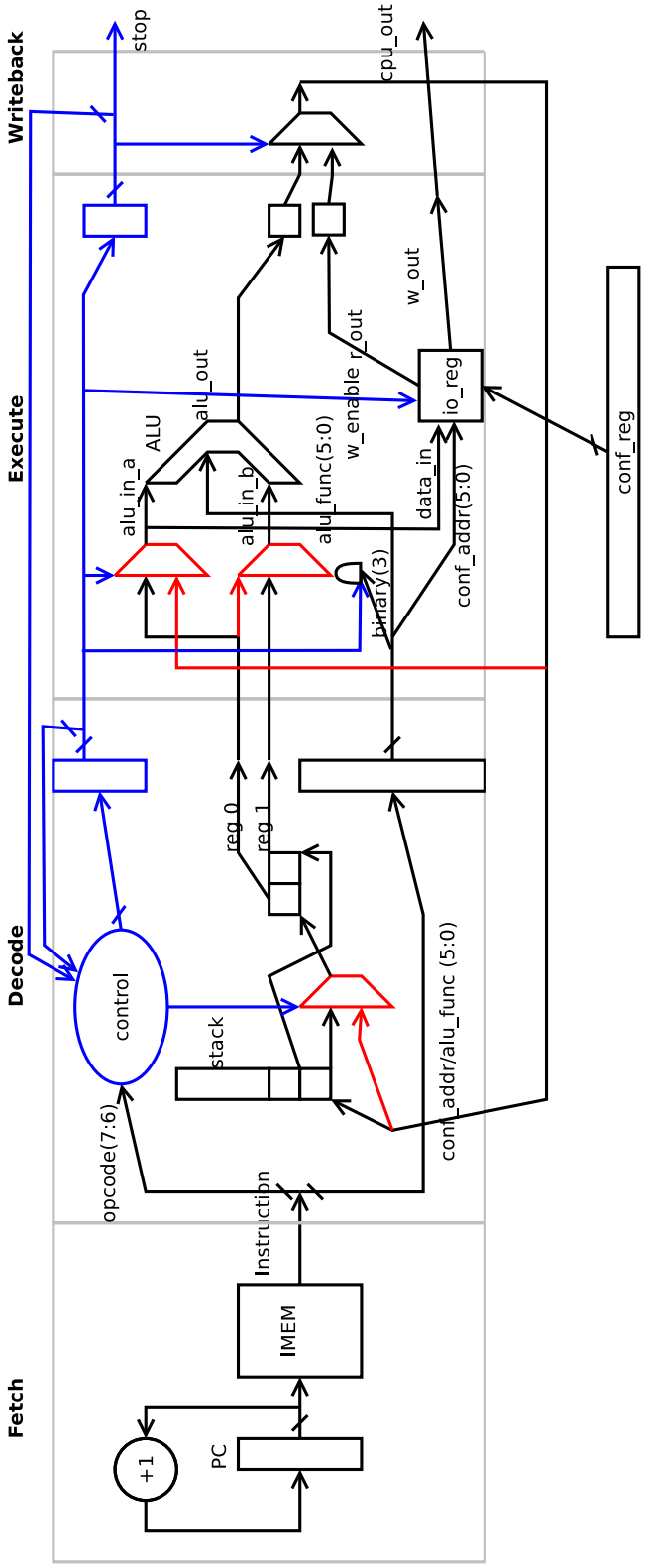


Figure A.2: Pipeline

Figure A.3 and A.4 shows how 4 instructions execute in a pipelined CPI design and a multicycle CPU design, respectively. Note how the multicycle CPU uses 16 cycles to execute 4 instructions, while the pipelined version requires only 7 cycles.

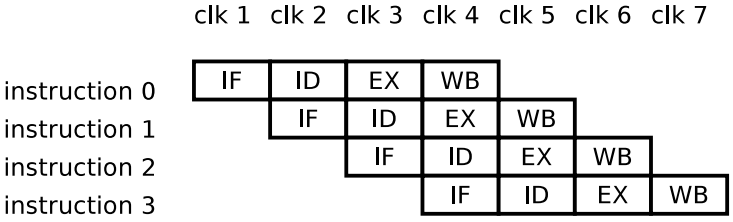


Figure A.3: Pipelined instructions

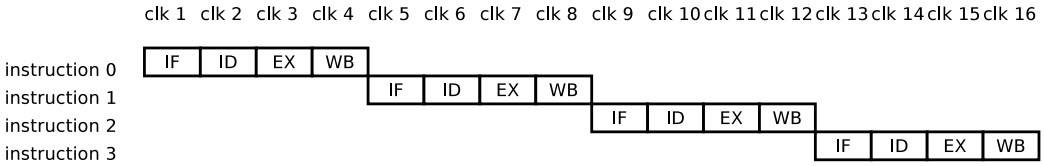


Figure A.4: Non-pipelined instructions, assuming all instructions require 4 cycles each.

A.3 The pipeline

All stages uses registered output. See figure A.2 for an overview of the pipeline.

A.3.1 CPU module

The CPU module (see *fpga_code/cpu.vhd*) is the toplevel of the core. It ties together the 4 pipeline stages, fetch, decode, execute and writeback.

A.3.2 Fetch module

The fetch module (see *fpga_code/fetch.vhd*) contains a register for the program counter. This is connected to the memory module. The address in

the PC(program counter) register is connected directly with the instruction memory. The PC increments each clock cycle until it wraps around.

A.3.3 Decode module

The decode module (see *fpga_code/decode.vhd*) is where most of the complexity in the pipeline lies. It consist of a combinatorial decoder which controls forwarding logic and what operation to do on the stack. The stack has 4 operations, do nothing, pop 1, pop 2 and push.

A.3.4 Execute module

The execute module (see *fpga_code/execute.vhd*) instantiates the ALU and the I/O register module(which is just a abstraction for handling reading the configuration bits and writing output bit. The execute module also implements several multiplexers used for forwarding.

A.3.5 Writeback module

The writeback module (see *fpga_code/writeback.vhd*) is the last stage in the pipeline it has a mux selecting between input or data from the ALU. The output data is also forwarded to execute, as well as skipping the stack in the decode stage in case of binary ALU instructions.

A.3.6 Memory module

The memory module (see *fpga_code/mem.vhd*) is the instruction memory implementation. It consists of a 16-bit wide read/write port and a 8-bit wide read port. This module is specifically designed to fit the rest of the extended Mecobo platform and its memory controller to not break compatibility with old programs in respect to the memory interfacing. The 8-bit read port is used by the CA CPU core to fetch instructions.

A.3.7 I/O register module

The I/O register module (see *fpga_code/io_reg.vhd*) implements a multiplexer network, which selects which pin to read from in the configuration register. Currently this is limited to value bits and not mode bits.

A.3.8 ALU module

The ALU module (see *fpga_code/alu.vhd*) is a very simple combinatorial circuit, currently supports "not", "and", "or" and "xor".

A.4 Hazards

The control in the decode stage handles setting flags to control the forwarding muxes as well as setting the appropriate stack function to execute. The three types of hazards are; structural hazards, data hazards, and control hazards.

A.4.1 Structural hazards

The one structural hazard that exists is also a data hazard. The fact that an instruction in the decode stage want to pop a value off of the stack while an instruction in decode wants to write to it, makes it a structural hazard because both wants to use the stack.

This is solved by forwarding data past the stack in the decode stage, when the instruction needs data that is in the writeback stage. This results in that neither of the instructions use the stack, since it is a push/pop relationship, the stack is not modified.

A.4.2 Data hazards

To handle data hazards, there is two points where data is being forwarded it is from writeback to decode and writeback to execute. This logic is drawn in red on the pipeline overview in Figure A.2.

A.4.3 Control hazards

There is currently no control hazards, since the architecture was completely rewritten and simplified, all the branch instructions was removed to keep the architecture small and simple. However in the future branching might be added again, so the 2 different control hazards which are identified are listed for completeness. These are the issues that will occur without some form of control hazard detection:

1. Conditional branches depends on the value of a status register of a previous instruction.
2. When a jump is executed, the instruction right after the jump instruction will be executed as well.

Point number 1 is solved by forwarding the value of the status output from ALU if the instruction currently in the execute stage writes status register. This is forwarded to control, which handles the conditional branching. If the instruction does not write status register, simply pass along the current value of the status register.

Point number 2 is solved by forwarding the new PC value, directly into the instruction memory, and setting PC to the new PC value + 1. This works in all cases where the address to jump to is gives as an immediate, either absolute or relative.

A.5 CA CPU architecture details

As explained earlier, the 16-bit instruction format is now an 8-bit format. With 3 types of instructions, alu type, input type and output type. Notice the lowest 6 bits of the O-type instruction is not used. These are planned to being used in the future, to be able to configure more than 1 output bit. Going from 16-bit to 8-bit was a big optimization and makes room for larger instruction memories which to be able to compute larger programs. Larger programs do not have as big impact on the stack size as the instruction memory, since the stack is just a fraction of the memories size. The stack is currently 16-bits.

Table A.1 describes the different instruction formats in use, the meaning of

the different values in the table is as follows:

opcode What operation the CPU is going to perform. This is the same bits for all instruction formats.

conf_addr The number of a bit in the configuration register.

alu_func What operation the ALU is going to perform.

– Not used to anything, should be passed as all zeros.

name	**bit 7-6**	**bit 5-0**
A-type	opcode	alu_func
I-type	opcode	conf_addr
O-type	opcode	–

Table A.1: Instruction format

A.6 ALU

Table A.2 describes all the implemented operations, status flags are currently not in use so they are not set. Common for all ALU operations is that bit number 3, indicates whether it is a binary operation or unary. The NOP instruction is regarded as a unary instruction, and will be decoded as any other unary instructions. That is, NOP will have its value forwarded like the value came from the ALU.

The reason why the NOP instruction is an ALU instruction is that it made the decoder easier to implement at first. However this might change in the future.

These values for the *alu_func* field can also be found in *fpga_code/cpu_package.vhd* with the `ALU_FUNC_` prefix.

name	**funct**	**description**
NOP	000000	Do nothing
NOT	000001	Unary instruction, inverts ALU port A.
AND	001001	Binary instruction, anding ALU port A and B.
OR	001010	Binary instruction, ors ALU port A and B.
XOR	001011	Binary instruction, xors ALU port A and B.

Table A.2: ALU function list

A.7 Opcodes

To better utilize the limited instruction width, the instruction formats in table A.1 might change to fit more instructions, since the alu_func field has 2 bits which is currently unused. This will yield a little more complex decoder but, in return the instruction still fit in 8-bits. Table A.3 describes the various opcodes and what instruction format they use. These can also be seen in *fpga_code/cpu_package.vhd* with the
`OPC_` prefix.

name	**opcode**	**format**	**description**
ALU	00	A-type	ALU functions, the ALU will carry out the command in the alu_func field.
OUT	01	O-type	The stops the core and the output value will be registered in the io_reg module.
IN	10	I-type	This reads the bit number denoted by the conf_addr field, from the configuration data register.

Table A.3: Opcode list

A.8 Control

How the pipeline is controlled can be seen in table A.4. ID is the decode stage, EX - execute and WB - writeback. The *push* signal is not currently used for anything. *alu_a_fw* controls the mux infront of ALU port A. *stop* is the stop signal indicating that an out instruction is coming and the respective core is finished computing. *stack_fw* is controlling the mux bypassing the stack in the decode stage. *write_out* controls the write enable port on the io_reg module in the execute stage, indicating a value should be written in the w_out register.

Note that even though it decodes "out" instructions followed by any other instruction, those sequences are not yet considered legal programs. After 1 out instruction the respective core is considered finished, and any other signals it might set on the out port, will not have any effect. This is accomplished by a oneshot register in the ca_cpu module. When the cores are reset, this register will also reset, making it ready for a new round.

ID	EX	WB	Action
in	in	in	Set *push*, push.
in	in	out	Set *push*.
in	out	in	Set *push*.
in	out	out	Set *push*.
out	in	in	Set *push*, set *alu_a_fw*, set *stop*, set *write_out*, push.
out	in	out	Set *push*, set *alu_a_fw*, set *stop*, set *write_out*.
out	out	in	Set *stop*, set *write_out*, pop 1.
out	out	out	Set *stop*, set *write_out*, pop 1.
in	in	alu	Set *push*, push.
in	alu	in	Set *push*.
in	alu	alu	Set *push*.
alu	in	in	Set *push*, set *alu_a_fw*, if binary: set *stack_fw* else: push.
alu	in	alu	Set *push*, set *alu_a_fw*, if binary: set *stack_fw* else: push.
alu	alu	in	Set *push*, set *alu_a_fw*, if binary: pop 1.
alu	alu	alu	Set *push*, set *alu_a_fw*, if binary: pop 1.
out	out	alu	Set *stop*, set *write_out*, pop 1.
out	alu	out	Set *alu_a_fw*, set *stop*, set *write_out*.
out	alu	alu	Set *alu_a_fw*, set *stop*, set *write_out*.
alu	out	out	Set *push*, if binary: pop 2 else: pop 1.
alu	out	alu	Set *push*, if binary: pop 2 else: pop 1.
alu	alu	out	Set *push*, set *alu_a_fw*, if binary: pop 1.
in	out	alu	Set *push*.
in	alu	out	Set *push*.
alu	in	out	Set *push*, set *alu_a_fw*, if binary: pop 1.
alu	out	in	Set *push*, if binary: pop 2 else: pop 1.
out	in	alu	Set *alu_a_fw*, set *stop*, set *write_out*, push.
out	alu	in	Set *alu_a_fw*, set *stop*, set *write_out*.

Table A.4: Pipeline control

A.9 Assembler

This section describes the initial design of the assembler and the language. The assembler is likely to be implemented in either C, Perl or a Bash script, the current implementation is a python program which differs from this initial design.

The general format of the instructions for the assembler is as described in figure A.5. The operation mnemonic comes first, followed by the operands. All tokens are whitespace separated. Bit address in the configuration are prefixed with $, and are assumed to be in decimal.

```
# Sample assembly program
# Calculates not(($10 xor $11) and ($12 xor $13))
in $10
in $11
xor
in $12
in $13
xor
and
not
out
```

Figure A.5: Example assembler syntax

Thie example program can be seen in figure A.6. The red arrows shows how to generate a program for this architecture. An operation is done when all child nodes of the current node has been visited and done. Each core will execute one such program.

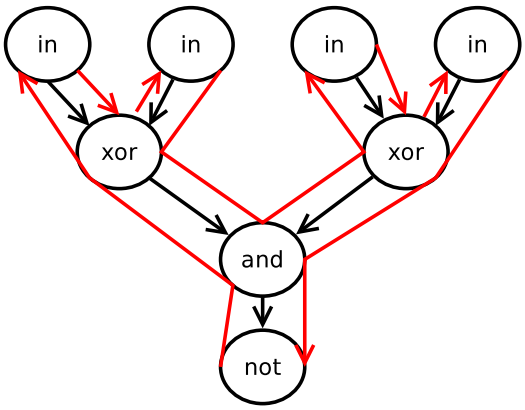


Figure A.6: Example program

To give an example in the context of CA, think of the "in" instructions as the edges to each node(in context of CA not like the nodes in figure A.6). That is, it defines this core's neighbourhood. The neighbourhood can be defined

freely, each core can address all other nodes, this allows us to program more general DDNs(Discrete Dynamic Networks) where there is not necessarily a local neighbourhood like in CA.

A.10 Simulation verification

The CPU core is verified using small programs and the correct output. If the correct value is written to cpu_out after the *stop* signal has been asserted, the test is OK.

The testbench takes a 16 bit aligned stream of instructions, the last word, must be padded with zeros if the program do not have an even number of instructions. The instructions are written into the cores memory, 2 instructions at the time(since 1 instruction is 8-bits), and the core starts when the reset flag is de-asserted. The core stops when the *stop* signal is asserted. The CPU testbench can be found in *fpga_code/cpu_test.vhd*.
To summarize the steps in the testbench:

1. Write instructions into memory.
2. De-assert the reset signal to start the core.
3. Run until *stop* is asserted.
4. Compare result.

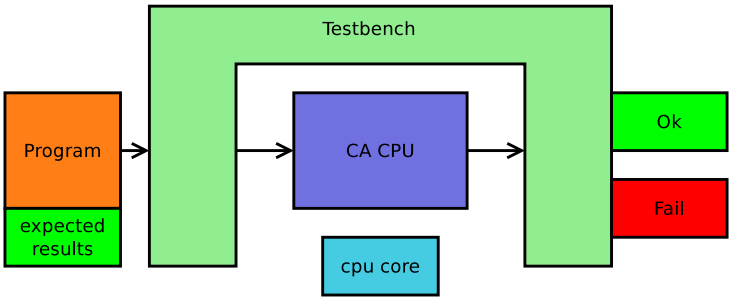


Figure A.7: Testbench

A.11 Corner cases

With a pipelined design there may be several different sequences of instructions that expose certain errors in the design, if they are not handled properly. In pipelined design it is important to test all the forwarding logic and in general the logic that handles the different hazards.

In this stack based design. The important points was the muxing on the alu inputs in the execute stage. That is unary and binary instructions which needs the result from the previous instruction. In the decode stage it is important to check the it does bypass the stack when it should.

Note that in the execute stage there is a second mux on ALU port B. The reason for this is that when there is a binary instruction in the execute stage and 1 of its values comes from the writeback stage, the second value should come from the stack, but the second value which was indeed popped from the stack in the decode stage is now on ALU port A, hence why this value is muxed into ALU port B.

It is easy to see from the pipeline control table A.4 which instruction sequences that is the most complex. The most interesting ones are the combination of "in" and "alu" instruction sequences, since "out" instructions followed by other instructions are likely not to occur considering it is not yet considered a legal program.

A.12 Synthesis results

As can be seen below the hypothesis about the FPGA size turns out to be correct. With a 64-core CA CPU, the design use 125% of the slices. So the current Mecobo hardware does no support the entire CPU. However until a new revision of the board with an updated FPGA is made, the CA CPU can be used with 32 cores.

```
Selected Device : 3s500epq208-4
 Number of Slices:                     5851  out of   4656   125% (*)
 Number of Slice Flip Flops:           2688  out of   9312    28%
 Number of 4 input LUTs:              11350  out of   9312   121% (*)
    Number used as logic:              9302
    Number used as RAMs:               2048
```

```
Number of IOs:                       2628
Number of bonded IOBs:               2627  out of    158   1662% (*)
   IOB Flip Flops:                     64
Number of GCLKs:                        1  out of     24     4%
```

The IO numbers are not interesting, because the CA CPU is the toplevel synthesized. So all the signals which are connected to the memories, memory controller, and command control are inputs and outputs, which it won't be in the complete design of the Mecobo FPGA hardware, since they will be connected internally.

Bibliography

[1] Gordon Pask. Physical analogues to the growth of a concept. 1958.

[2] A. Thompson, P. Layzell, and R.S. Zebulum. Explorations in design space: unconventional electronics design through artificial evolution. *Evolutionary Computation, IEEE Transactions on*, 3(3):167 –196, sep 1999.

[3] J.D. Lohn, D.S. Linden, G.S. Hornby, and W.F. Kraus. Evolutionary design of an x-band antenna for nasa's space technology 5 mission. In *Antennas and Propagation Society International Symposium, 2004. IEEE*, volume 3, pages 2313 – 2316 Vol.3, june 2004.

[4] Eric W. Weisstein. "rule 30." from mathworld–a wolfram web resource. http://mathworld.wolfram.com/Rule30.html.

[5] S. Wolfram. Universality and complexity in cellular automata. *Physica D: Nonlinear Phenomena*, 10(1-2):1–35, January 1984.

[6] Chris G. Langton. Computation at the edge of chaos: phase transitions and emergent computation. *Phys. D*, 42(1-3):12–37, June 1990.

[7] Melanie Mitchell. Life and evolution in computers.

[8] Odd Rune S. Lykkebø. Design and implementation of a prototype platform for evolution in materio. 2010.

[9] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, N. Salami, N. Kajihara, and N. Otsu. Real-world applications of analog and digital evolvable hardware. *Evolutionary Computation, IEEE Transactions on*, 3(3):220–235, 1999.

[10] S. Harding and J.F. Miller. Evolution in materio: a tone discriminator in liquid crystal. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1800 – 1807 Vol.2, june 2004.

[11] William Ross Ashby. An Introduction to Cybernetics. 1957.

[12] Heinz von Förster. On self-organizing systems and their environments. *Self-organizing Systems (ed. Yovitz and Cameron)*, pages 31–50, 1960.

[13] J.F. Miller and K. Downing. Evolution in materio: looking beyond the silicon box. In *Evolvable Hardware, 2002. Proceedings. NASA/DoD Conference on*, pages 167 – 176, 2002.

[14] S. Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55:601–644, July 1983.

[15] S. Ulam. Random processes and transformations. In *Proceedings of the International Congress on Mathematics*, volume 2, pages 264–275, 1950.

[16] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[17] E. F. Codd. *Cellular Automata*. Academic Press, Inc., Orlando, FL, USA, 1968.

[18] Alvy Ray Smith, III. Simple computation-universal cellular spaces. *J. ACM*, 18(3):339–353, July 1971.

[19] E. Berlekamp, J. Conway, and R. Guy. *Winning Ways for your Mathematical Plays*, volume 2. Academic, 1982.

[20] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1982. 10.1007/BF01857727.

[21] Matthew Cook. Universality in Elementary Cellular Automata. *Complex Systems*, 15(1):1–40, 2004.

[22] Melanie Mitchell, James P. Crutchfield, Peter T. Hraber, In G. Cowan, D. Pines, D. Melzner (editors, and Complexity Metaphors. Dynamics, computation, and the "edge of chaos": A re-examination. In *Complexity:Metaphors, Models, and Reality*, pages 497–513. Addison-Wesley, 1994.

[23] John H. Miller and Scott E. Page. *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton University Press, 2007.

[24] Moshe Sipper. The emergence of cellular computing. *Computer*, 32(7):18–26, July 1999.

[25] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[26] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[27] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, USA, 1996.

[28] Xilinx. XAPP463 - Using Block RAM in Spartan-3 Generation FPGAs. http://www.xilinx.com/support/documentation/application_notes/xapp463.pdf, 2005.

[29] Atmel. AVR 32-bit Architecture Manual. http://www.atmel.com/Images/doc32000.pdf, 2011.

[30] Atmel. AT32UC3A Datasheet. http://www.atmel.com/Images/doc32058.pdf, 2012.