# Reconfigurable Cellular Automata Computing for Complex Systems on the SPACE Machine

by

David Frederick James George B.Sc.

2005

This thesis is presented for a masters degree by research

at the University of Western Australia.

# Abstract

Many complex natural and man made systems are inherently concurrent in nature, consisting of many autonomous parts that interact with each other. Cellular automata allow the concurrency and interactions of these complex systems to be modelled. Using a reconfigurable a computing platform for running cellular automata models allows the natural concurrency of digital electronics to be directly exploited by the system being modelled.

This thesis investigates methods and philosophies for developing cellular automata models on a reconfigurable computing platform, the SPACE machine. Modelling and verification techniques are developed using a process algebra, Circal. These techniques allow the desired behaviour of a system to be specified and simulated. The model is then translated into a digital design, which can be verified as correct against the behavioural model using the Circal system.

Three cellular automata system are used to develop the methods and philosophies. The Game of Life is used to investigate how to model and implement CA on the SPACE machine. The Philosophies and techniques that are developed for the Game of Life are used in the following systems. More complex cellular automata models of road traffic are used to further develop the modelling techniques developed in the Game a Life. A user interface, which was created for viewing the outputs from the Game a Life, is extended to allow cellular automata cells to be dynamically placed and moved about on the computing surface, allowing the user to observe and modify experiment in real time. A cellular automata based cryptography system is then used to further enhance the techniques developed, and particularly to explore the area of producing dynamically reconfigured circuits as the inputs to the system change.

The thesis concludes that there are many real life complex systems, such as road traffic simulation and cryptography, which require high performs systems to run on. The methods and philosophies developed in this thesis allow CA systems to be modelled using process algebra and run directly in digital hardware, allowing the natural concurrency of the hardware to be fully exploited.

# Table of Contents

# Table of Figures

# Acknowledgements

I would like to thank Professor George Milne as principle supervisor of my thesis, in particular to thank him for being patient with me and encouraging me through this work.

I am also grateful for many valuable discussions with colleagues and friends including Alex Cowie, Bernard Gunther, and Oliver Diessel.

Finally, to my wife and friend, Susan George for her support, input and bearing with it!

# 1. Introduction

## 1.1 Motivation of Thesis

### 1.1.1 Cellular Automata and Reconfigurable Computing

Many real life problems are inherently concurrent while traditional computer architectures are essentially serial in their computations. The Scalable Parallel Architecture for Concurrency Experiments (SPACE) Machine platform [8], [63] makes the inherently concurrent nature of digital electronics accessible for computations. Cellular Automata (CA) can be used as a mechanism for modelling concurrent problems and realizing models on the SPACE Machine. Implementing concurrent systems on a concurrent architecture allows the problems to be modelled at very high speeds, which is necessary in many situations, such as simulating the effect of a road accident on traffic flow in order to divert traffic and reduce congestion.

This thesis investigates the applicability of using CA as a mechanism for realizing fine-grained models of concurrent systems on reconfigurable hardware and derives philosophies for the implementation of CA systems [5], [96] on parallel reconfigurable hardware, in particular the SPACE Machine series [8], [60], [63]. A theoretical model for a CA is presented and written in terms of Circal [58] processes. This model has then been successfully translated into digital circuits. The marriage of CA modelling and reconfigurable computing is demonstrated in three applications; simulating the Game of Life [22], modelling multiple-lane road traffic and performing data encryption / decryption in cryptography. The applications are all examples of complex dynamic systems that may be successfully modelled by CAs and implemented on the SPACE Machine. While the SPACE machine is used for all experiments in this thesis the principles developed here are applicable to any FPGA based architecture that has a uniform layout of logic cells. Furthermore, examples of non-homogenous CA are implemented. This novel design paradigm, that achieves reconfigurable CA computing on the SPACE Machine, is particularly important as the following motivation explains.

Firstly, CA systems are an important approach to studying many complex physical systems that are difficult to model; particularly systems that demonstrate chaotic behaviour such as weather systems, gas models or fluid flow. Secondly, reconfigurable computing is an important tool that allows the dynamic redesign of digital logic circuits in response to the constraints of a particular

application. The use of such a parallel reconfigurable computer is a particularly natural choice for implementing the type of computations required by a CA.

The merge of CA and reconfigurable computing is particular important since current hardware for cellular automata (eg. the CAM-8 Machine [77]) are only suitable for implementing *homogenous* CA. Such homogenous modelling greatly restricts the type of problem that can be modelled. In particular, such modelling would have restrictions in many physical systems that are interesting to model with CA. The atmospheric system is one example of a non-homogenous CA model where it would be appropriate to model different 'types' of cells that could be found in the system. For example different cells representing high atmosphere than those in the lower atmosphere, or cells representing one type of gas over another. Where it is desirable for cells to have different rule according to their position then non-homogenous modelling is required and reconfigurable computing has the advantage over current cellular automata hardware in that it is possible to implement non-homogenous models. This thesis demonstrates this as reality in road-traffic simulations.

Additionally existing reconfigurable hardware has not been used for simulating CA. Some traffic simulations have been done on early versions of the current SPACE Machine. In general the physical construction of reconfigurable computing platforms is not amenable to CA type problems. In particular the boards do not have a regular 2-dimensional grid layout as the SPACE Machine series does. This feature of the SPACE Machine makes it particularly natural to explore CA implementations, where the cell neighbourhood is an all-important concept, allowing CA to be implemented very efficiently.

### 1.1.2 The Game of Life, Road Traffic Simulation and Data Encryption

Three systems were chosen to implement on the Space Machine and the motivation for selecting them is now presented. Importantly, the systems not only demonstrate the CA paradigm but also make important theoretical and practical contributions to the investigation of CAs and reconfigurable computing. The Game of Life is the classic, and simple, CA that makes it possible to investigate how CA models might be theoretically specified and implemented on a reconfigurable machine. Road-traffic modelling is an important real-world application that demonstrates the importance of non-homogenous modelling while data encryption / decryption

is another highly important real world application where the theoretical potential of run time reconfigurability is an exciting possibility in making secure, fast, encryption using CA models.

Firstly the Game of Life was used to develop modelling techniques and design philosophies for dynamic cellular systems. Experiments with the Game of Life revealed that a cell can be modelled by a two-process system where one process was the cell logic (behaviour) and the other process models time. The time process is the same for all cells, regardless of the cell update rule (see section 2.2.1), while the cell logic is the update rule. This decomposition was applied to all the CA systems studied, so demonstrating the importance of this general model established with the classic Game of Life CA. These experiments also investigated how the Circal process algebra [58] can be used to specify the behaviour of the automata. The Circal system specification of the CA can be run, and used as a model to prove the correctness of the automata model. Circal then permits the automated comparison of digital circuitry with the behavioural process specification, and is thus an important tool to use when the CA are specified in terms of behaviour to ensure the implementation is correct on the reconfigurable computing platform.

Secondly, ‘better than real time’ traffic modelling presents a particular challenge with respect to minimising road congestion. This is particularly the case when responding to emergency situations such as a car breakdown or accident on a freeway that impedes the normal flow of the system. Steps must be taken to re-route vehicles in order to minimise congestion and the CA traffic models presented here are of particular value in providing real time answers to ‘what if’ questions. The CA simulations would make it possible to experiment, in better than real time, with a variety of routing options in response to particular road conditions and subsequently to apply the most appropriate routing options. Theoretically the road traffic problem demonstrates an interesting class of non-homogenous CAs that existing cellular automata machines, and CA programming languages, cannot currently model. This makes the use of the reconfigurable computer particularly interesting and valuable since areas of the FPGAs (see section 2.3.3) can be configured differently to model different cell types in non-homogenous CAs. The use of process algebra to both model and verify the implementation of the CA models allows CA models to be both quickly and accurately constructed for the reconfigurable computer.

Thirdly the cryptography application is an increasingly important area once again for *both* CA models and reconfigurable hardware. It is natural to explore the CA modelling techniques on the

SPACE Machine in context of this application. The application is important for CA models due to the links that have been made with dynamic complex systems (see section 5.2.2). It is possible to use the cells of a CA to model a complex system that is sensitive to changes in its initial state, and is unpredictable. This complex system can be used to encode a message (i.e. become the key of the encryption). While reconfigurable hardware is not *necessary* for applying such CA algorithms to encryption / decryption there is however a huge potential in using reconfigurability. This is in the speed that a hardware solution can provide and the additional security that hardware solutions can provide over software, by preventing examination of data or algorithms, see section 5 for more information.

## 1.2 Contributions of Thesis

There are a number of specific contributions made by this thesis including the following:

1. Developed a philosophy, techniques and tools for modelling and implementing CA on a reconfigurable computing platform.
2. Modelling non-homogeneous and homogeneous (see section 2.2.7) CA: The thesis has developed a theoretical model of homogeneous and non-homogeneous CA, using Circal specification language that is suitable for implementation on reconfigurable hardware.
3. Implementation of a homogeneous CA model on reconfigurable hardware: The thesis has realised the implementation of the theoretical model in the game of life.
4. Implementation of non-homogeneous CA models on reconfigurable hardware: The thesis has created a family of cellular automata-type road traffic models (for multi-lane freeways) using CA cells to model segments of roads.
5. Investigation of reconfigurable CA in encryption: The thesis has successfully developed a reconfigurable CA based encryption system on the SPACE machine and has discussed the potential importance of run-time configurablility in this particular CA application.
6. A visual interface to the SPACE machine for the development of CA models. This interface is a valuable tool in terms of CA experimentation enabling the dynamic behaviour of non-homogenous evolving systems to be rapidly constructed, visually observed and recorded.

## 1.3 Outline of Thesis

Chapter two reviews the relevant background literature for the thesis including the origin and background of Cellular Automata (2.1), physically realised Cellular Automata hardware and reconfigurable computing systems that may be used to model CA (2.2), an introduction and overview of concurrency (2.4) and the specification of CA models using CA programming languages and the Circal process algebra (2.5) before outlining a summary of the state of the art (2.6) from which this thesis shall proceed.

Chapter three develops techniques and philosophies for modelling and implementing CA. It then goes on to use these techniques to model and implement the Game of Life on the SPACE machine. A background and description of the cell rules is provided to the Game of Life (3.2), together with an investigation of the general way that a CA cell might be modelled and physically realised upon the space machine (3.3). Two particular ways of implementing the CA cell on the SPACE machine are considered (3.4) before implementing a digital circuit to implement the more optimal design. Finally the results of some simulation testing are included in a discussion (3.5).

Chapter four considers the use of CA to model a complex physical system that may be understood in CA terms, that is a road traffic system. The importance of this application is considered reviewing some previous road traffic simulations (4.2) including some that use CA. An important contribution of this thesis is made in modelling multi-lane traffic and the particular concerns of multi-lane carriageway simulations are considered (4.3), before demonstrating how this CA for multi-lane traffic simulation may be specified in Circal and implemented in digital logic on the SPACE Machine (4.4). The development of the traffic simulations also required the development of a graphical user interface. This tool is described (4.5) making particular note of its applicability to the design of any CA type system, emphasising its interactive nature and particularly pioneering contribution to the SPACE machine series. The results of the test simulations are discussed (4.6) demonstrating the success of the CA for simulating multi-lane freeway traffic.

Chapter five provides an introduction to cryptography (5.1) including reversible and irreversible dynamic systems in the context of using CA for encryption and decryption. The proposal of this

thesis for using an encryption algorithm implemented by CA on a reconfigurable computing (5.3) is described and experimental results presented (5.4) before making a summary (5.5).

Chapter six concludes the thesis providing future directions for both the general paradigm of reconfigurable CA computing on the SPACE Machine, (6.2) and also for future directions in the particular applications of road traffic modelling (6.3) and cryptography (6.4), highlighting the important and interesting problems as well as suggesting potential research directions before concluding and evaluating the contribution of this thesis (6.5).

# 2. Literature Review

## 2.1 Chapter Overview

This chapter provides an introduction to, and a review of current literature, for the main topic covered in this thesis. It begins with an overview of cellular automata, starting with their history, and then explains what cellular automata are and what they can be used to model. Existing cellular automata machines are then reviewed. Reconfigurable computing is then introduced, including a description of the SPACE machine, and the concept of concurrency, which brings reconfigurable computing and cellular automata together. Following this a number of existing cellular automata programming languages are discussed before introducing Circal, a process algebra, which is used in this thesis to describe cellular automata as well as to model and simulate the digital hardware which is implemented on the SPACE machine.

## 2.2 Cellular Automata

### 2.2.1 Origin of Cellular Automata (CA)

Cellular automata were invented in the 1940's by the mathematicians John von Neumann and Stanislaw Ulam, while they were working at the Los Alamos National Laboratory in northern central New Mexico [96], [5]. Ulam was working on cellular games where each pattern was composed of a square (or triangular or hexagonal) cell on a chessboard like surface. All growth and change of patterns on the chessboard took place in discrete jumps. The fate of a given cell, at the next time step, depended only on the current state of the cell and the states of its neighbouring cells. Ulam discovered many patterns grew almost as if they were alive. A simple starting pattern could evolve into a delicate, coral-like growth while two patterns might ‘fight’ over territory, sometimes leading to mutual annihilation. Ulam may be credited with the original concept of CA while von Neumann developed the idea while considering self-reproducing systems.

John von Neumann spoke of a ‘complexity barrier’ - the imaginary barrier separating simple systems from complex systems. A simple system can give rise to systems of less complexity only. In contrast, a sufficiently complex system can create systems more complex than itself. Exact self-reproduction is a feature of systems right on the complexity barrier - systems that preserve but do not increase their level of complexity in their offspring. A complex system is

one whose component parts interact with sufficient interactions that they cannot be predicted by standard linear equations; there are so many variables are at work in the system that its overall behaviour can only be understood by studying the system as a whole. Breaking the problem into parts to discover its behaviour does not work with complex systems; the whole is greater than the sum of the parts.

### 2.2.2 CA, A Finite State Machine

A CA is a lattice (grid) of cells where each cell is a form of Finite State Machine (FSM). A FSM is defined in [71] as;

> *A model of computation consisting of a set of states, a start state, an input alphabet (symbols), and a transition function which maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function.*

This can be defined mathematically as:

A deterministic finite state machine is a quintuple <Q,I,delta,q0,F>

where:

Q is a finite set of states;

I is a finite set of input symbols;

delta is the transition function which describes how the rule for how the FSM changes from one state to another, i.e. delta: Q x I->Q;

q0 element of Q is the initial state;

F contained in Q is the set of final states (or accepting states).

That is, the FSM has a known set of states that it can be in at any one time. The inputs to the FSM are constructed from the input symbols. The FSM then changes from one state to the next as defined by the transition function, which takes as input the current state of the machine and the current set of inputs. The FSM always starts in a known state. If, after processing all inputs the inputs the machine finishes in one of the final states then the input sequence is considered to be valid, otherwise it is considered to be invalid.

This formal definition considers a final state for the CA in determining, whether the input states were valid. This is useful for models that need to determine whether the data that has been

processed was a valid set of data. The models in this thesis do not need to consider whether the input data was valid so the set of final states is considered to be the complete set of states.

For example consider an automatic door controller, where the door has sensors to detect whether it is open or closed. A FSM may be defined with 4 states, open, opening, closing and closed. The set of input symbols in this case may be from two buttons, one to open the door, the other to close it. The transition function describes what the FSM should do if it gets an input in each of its states, for example if the door is open and the close button is pressed the door will start closing, but if the close button is pressed while the door is already closed it will ignore the input (remain in the closed state). An additional initialisation state could be created as the initial state of the machine in which the current position of the door is determined from sensors on the door. The final state is not necessary in this example but could be defined as an out of order state, and could be entered when the door position sensors indicate that the door has stopped working.

FSMs are often expressed using a state transition diagram. This is a diagram that generally has a circal representing each state of the FSM. The circles have directional lines between them, representing state changes. Each line has text indicating what input conditions would cause the source state to change to the new destination state. Figure 2-1 shows a simplified state diagram for the example given above.

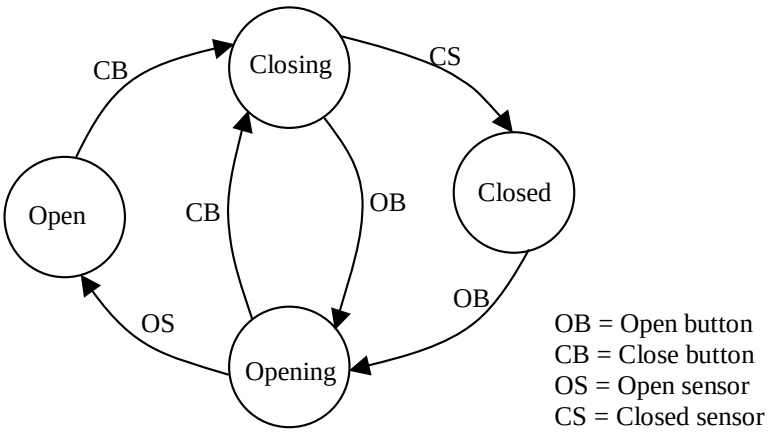



**Figure 2-1 Example state diagram**

### 2.2.3 What is a CA?

An automaton, or automata, is also defined as an entity that acts under control of a set of rules (transition function) where the rules make it possible for the automaton to transit between a numbers of predefined states. Once started, the automaton can continue to act on its own without intervention. The automaton is a mathematical model consisting of a set of states and transitions between those states. A rule defines how to move between the states. A cellular automaton (CA) is a set of automata (often called cells) arranged on a lattice that may be 1, 2, 3 or more dimensional, arranged on square, hexagonal, or any other regular lattice. Each cell has a neighbourhood, defined to be a set of cells that are within a certain relative distance from itself. This is usually cells that are physically close to the cell on the lattice, e.g. those cells that are touching the cell. All cells on the lattice are updated synchronously and the state of the entire lattice advances in discrete time steps producing generation after generation of cell states. Thus a CA consists of two components [64]: 1) the transition rule that gives the update state S(t+1) for each cell as a function of its neighbourhood and 2) the cellular space or lattice of cells each obeying identical transition rules with identical connections between neighbouring cells.

As an example, consider the following very simple rule for a 1-dimensional CA that if either the cell to the left or the cell to the right is alive (exists) then the cell will be alive in the next generation. Otherwise the cell will be dead (cease to exist) in the next generation. The neighbourhood of a cell is simply the cells on either side. The rule transitions are summarised by

Figure 2-2, which also illustrates a series of discrete time steps for the 1-dimensional lattice. A '+' represents a living cell, and a '-' represents a dead cell. The first table shows the rule for a cell. In the 'neighbourhood & state' columns the state of the cell is the middle column, with the neighbours on each side. The 'cell state S(t+1)' is the state that the cell will transition to at the next time step. The other two tables show the progressions of the automata for different starting states.

| Transition Rules for a 1-dimensional CA |
|---|

| Neighbourhood & State S(t) | | | Cell State S(t+1) | Neighbourhood & State S(t) | | | Cell State S(t+1) |
|---|---|---|---|---|---|---|---|
| + | - | - | + | - | - | - | - |
| + | - | + | + | - | - | + | + |
| + | + | - | + | - | + | - | - |
| + | + | + | + | - | + | + | + |

| Temporal Development of the CA | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Start | - | + | - | - | - | + | - | - |
| S(t) | + | - | + | - | + | - | + | - |
| S(t+1) | - | + | - | + | - | + | - | + |
| S(t+2) | + | - | + | - | + | - | + | - |
| etc. | | | | | | | | |

| Temporal Development of the CA | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Start | - | - | - | + | + | - | - | - |
| S(t) | - | - | + | + | + | + | - | - |
| S(t+1) | - | + | + | + | + | + | + | - |
| S(t+2) | + | + | + | + | + | + | + | + |

**Figure 2-2 Transition Rules and Development of a simple 1-dimensional Cellular Automaton**

The behaviour of CAs is often illustrated using space-time diagrams (for example Figure 2-3) where the configurations of states in the lattice are plotted as a function of time. A 2-dimensional time-space diagram vertically strings together the 1-dimensionmal CA lattice configurations. A 3-dimensional diagram would string together successive configurations of a 2-dimensional lattice.

### 2.2.4 Classes of Automata

Stephen Wolfram [101] worked with a one-dimensional variant of von Neumann's cellular automata, as in the example above. Each cell determined its next state from its own current state and the current state of its immediate neighbours, one on each side. The outputs of his experiments were represented using a space time diagram as described above. Wolfram explored the various cell transitions for these single dimensional CA while investigating the computational abilities of CA [102] [104]. He found that the CAs exhibited four different types of behaviour. From this observation he classified the behaviour of the CA into four classes;

1. Class 1 patterns organise themselves into a spatially homogenous state, for example all dead cells or all live cells, i.e. the state of each cell becomes constant (alive or dead).

2. Class 2 produce a sequence of periodic stable structures where the automata cycles through a fixed number of states.
3. Class 3 grow indefinitely at a fixed speed (i.e. the number of live cells increases with time).
4. Class 4 produce complicated localised structures that move through space and time.

Figure 2-3 shows examples of each of Wolfram's four classes of CA, from [103].

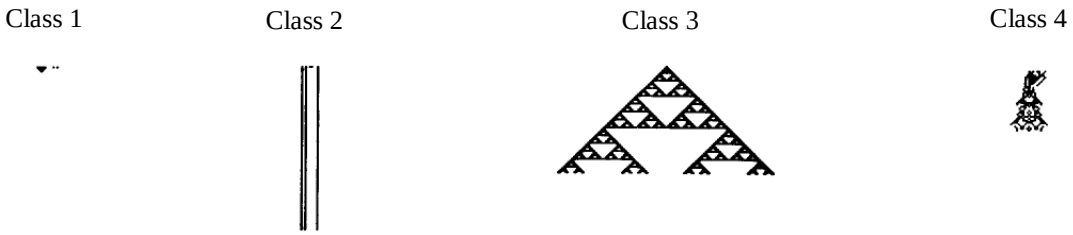



**Figure 2-3 Example of Wofram's 4 classes**

Wolfram also looked at the effect of small change in the initial state and found that small changes produced no change in final state with Class 1, regions of Class 2 patterns were affected by initial state while Class 3 demonstrated changes over a region of ever increasing size and Class 4 had completely irregular changes. CA automata evolution may also be viewed as computation and this poses interesting theoretical questions about computability and decidability. No simple theory or formula has been found to describe the overall behaviour of such systems, and the consequences of their evolution cannot be simply predicted, but can only be found by direct simulation and observation. This makes it particularly important to be able to model CA systems in real time, or faster.

It has also been shown that CA can be used to perform various computations, including synchronization, counting, ordering and random number generation [87] [102]. Computations using CA can be achieved using genetic algorithms to process data [66] [67] [76] [87].

CAs have particularly been applied to the simulation of complex systems [103]. The notion of computation at the edge of chaos is important [66]. Langton, [43], performed a number of experiments to explore the relationship between the proportion of 'non-quiescent' output states in the rule, which he called $\lambda$ and the behaviour of the CA. For a binary state CA (a CA where each node is either living or dead) the proportion of 'non-quiescent' output states, $\lambda$, is the

proportion of input states that produce a living cell, from the cell rule table. Langton found that, in general, as λ Moves from 0 to (1-1/k), where k is the number of states each cell in the CA can have, the behaviour of the rules passes from 'fixed point' to 'periodic' to 'complex', to 'chaotic'. These states are roughly equivalent to Wolfram's four classes. It has been hypothesised [65] that when biological systems must perform complex computation in order to survive, then evolution tends to select such systems near a phase transition (a value of λ close to where behaviour changes) from ordered to chaotic behaviour. Hence these classes of automata, identified by Wolfram, are interesting to consider when such complex systems are being modelled.

### 2.2.5 Modelling Physical Systems with Cellular Automata

Current mathematical models of natural systems are usually based upon differential equations that describe the smooth variation of one parameter as a function of a few others (eg. the Navier-Stokes equations). While these models work well for some natural systems (e.g. fluid flow models), attempts at obtaining a generalised theory, which can be used to model any natural system, using a differential equation formalism have met with limited success [40]. CA provides an alternative approach to modelling physical systems by discretely modelling small parts of a system and the interactions between these parts. Both natural physical systems exhibiting growth and change, such as granular systems, fluid flow, biological, and bushfires may be modelled as well as man-made physical systems such as road traffic flow or crowed movements can be modelled using CA.

When using a CA to model physical systems the physical space of a problem is divided up into many small, generally identical cells, each of which would be in one of a finite number of states. The state of a cell represents the current state of its part of the physical system (e.g. it may represent whether a gas particle is present or not). The state of the cell evolves according to a rule that is both local (involves only the cell itself and nearby cells) and universal (all cells are updated simultaneously using the same rule).

### 2.2.6 Lattice Gas Models

Lattice gas type models are very similar to CA models and appear as a highly developed subculture of general CA research. The LGA (lattice gas automaton) is a particular type of CA that is used for the simulation of viscous fluid flow. Fluid particles are defined where each

particle represents a finite mass of fluid. Motions of these fluid particles are constrained to move in a mesh like 'lattice' and collide only at the intersections ('nodes') of these lines. Frisch et al [21] concluded that for flow in 2-dimensions, a hexagonal lattice is satisfactory. Instead of many complex interactions and the near infinite motion characteristics of a real fluid there are a greatly reduced number of interactions in the model. When a collision occurs various outcomes are possible. In some cases the direction of particle flow changes, while in others no change in direction occurs. Some experiments modelling LGA on the SPACE machine have been performed [86].

### 2.2.7 Homogeneous and Non-Homogeneous CA

The behavioural definition of CA, as described in section 2.2.3, where all the cells in the CA obey the same rules are referred to in this thesis as homogeneous CA. It is possible, and sometimes desirable, for some of the cells to have different rules, and possibly different neighbourhoods, this type of CA are referred to as non-homogeneous CA in this thesis. An example of where non-homogeneous CA is usefully is modelling traffic, where, if each cell represents a section of road, then a slip lane may require different behaviour to a section of the main carriageway, see chapter 4 for further discussion.

## 2.3 Hardware for Cellular Automata

### 2.3.1 Overview

Conventional computers are ill suited to run the inherently uniform and parallel CA models and so discourage their development mainly due to performance problems. In the 1960s a new trend in computing design began with the construction of cellular logic machines. These machines emulated the cellular automaton by using a single high-speed processing element to operate sequentially on an array of binary data. The Cellscan machine was built in the United States in the 1960’s [77] and was followed by GLOPR (Golay Logic Processor) and BIP (Binary Image Processor) [72]. These machines were used for basic image processing tasks. Then in 1981 Sternberg introduced the Cytocomputer [91], which included multi-state processing elements in a pipeline. Cellular logic was performed by means of a lookup table, where the rule for the cells is stored as a table in memory, mapping each set of input values to an output value.

### 2.3.2 Cellular Automata Machines:

#### 2.3.2.1 CAM-8

CAM-8 is a mesh-architecture multi-processor machine optimised for large-scale simulation of a wide range of CA models, particularly simulations of physical systems. It was developed by Margolus at MIT Laboratory for Computer Science [46], [47], [48]. It enables the modelling of fine-grained systems, importantly making possible the modelling of n-dimensional CA. It has been used for modelling physical systems including the movement of gas particles in 3-dimensional space.

A CAM-8 system consists of a number 'processing nodes' physically connected together in a 3 dimensional array. A processing node typically simulates a few million cells in the CA.

The accessible neighbourhood of each cell is large and this contributes to optimal operation of the system. The processing nodes each run synchronously, using a local copy of the cell updating algorithm, each processing node updating its portion of the CA space serially, thus the CAM-8 only simulates parallel updating of cells, which is actually happening serially. Maximum performance is only possible when the number of bits to represent the state of the cell, multiplied by the number of neighbours communicated with, is less than or equal to 16. Larger problems require multiple steps for communications and cell updates. Thus there are limitations to the size of problem suitable for this architecture.

From a programmer’s viewpoint, CAM-8 appears as an n-dimensional array of programmable cells, accessible from a Sun SPARC station. Programs are currently written using a modified version of FORTH (C/C++ libraries and a custom CA language are under development).

Processing nodes consist of dynamic RAM (DRAM) and a static RAM (SRAM). The DRAM holds the states of the CA cells and is also used to communicate states between cells, allowing each node to obtain the state of the nodes in their neighbourhood. The DRAM consists of 16 'bit planes'. Each of the bit planes is physically constructed from one bank of DRAM. A cell consists of one bit from each of the bit planes. The bit planes can be independently shifted, by any amount, in any direction. This is achieved by modifying the addresses to the bit planes. The SRAM holds the CA rules in a lookup table and is used for updating CA cells. Updating of cells is performed by the data for a cell being passed through the lookup table and written back into the cell. CAM-8 also includes a custom STEP (Space Time Event Processor) chip [49]. The

STEP chip is an application specific integrated circuit. The custom STEP chip controls the running of the model. The CA model is run in two phases. Firstly the bit planes are shifted, which communicates cell states to their neighbouring cells. Secondly, the cells are updated using the lookup table. This process is pipelined. One cell is updated every clock cycle so there is no time penalty on the cell updates. The sliding bit plain model for communications gives each cell potentially a very large neighbourhood (a few million cells), although the usefulness of such a large neighbourhood is not known.

#### 2.3.2.2 Firefly

The Firefly machine is a hardware implementation of a cellular programming algorithm, which is an evolutionary (genetic) algorithm implemented as a non-homogeneous CA. The Firefly machine was developed by Sipper [87] [20] to demonstrate that an evolutionary algorithm could be fully implemented in hardware. The Firefly machine consists of 9 FPGAs connected in a row, a display, some switches for controlling the machine plus some peripheral circuitry The machine has no external connections (apart from power). The evolutionary algorithm is implemented as a single dimensional CA with each cell having a neighbourhood of one cell to the left and one to the right. The whole machine updates all of the CA cells in parallel, providing a high speed of cell updates. The logic in the FPGAs is fixed, no reconfiguration is possible, and hence the cell rule cannot be changed.

#### 2.3.2.3 CAMEL

CAMEL (Cellular Automata environment for systEms modeLing) was developed by Domenico Talia et el at the Università della Calabria, Italy in the mid 1990's [16] to allow the development of high speed cellular automata machines while hiding the underlying architecture from the user. The CAMEL system uses its own language, CARPET, for describing the Cellular Automata (see section 2.5.1.6).

The engine of the machine, which runs the code simulating the CA cells, is constructed of 32 INMOS transputers connected in a toroidal mesh. This has connected to it two additional transputers, a controller, which coordinates the transputers in the engine, and a graphical controller, which allows the user to monitor and modify the simulation as it is running. The whole system is connected to a host PC. Each of the transputers in the engine runs identical programs on the data for the cells for which they are simulating.

The user interface to the system allows the user to setup the simulation (e.g. number of steps to run for, size of the simulation etc) and allows global data, available to all nodes, to be modified, which permits applications to be tuned on the fly. It also provides a graphical visualisation of the data in the cells as the simulation is running.

The CAMEL environment has been used to simulate various CA systems including lava flow modelling, traffic flow, and image processing and genetic algorithms. The approach taken in the modelling of CA in the CAMEL environment has been to develop relatively complex models of the CA's, with each cell having multiple states and complex update rules.

While the CAMEL environment increases the speed of simulating CA systems by distributing the CA across multiple processors, each processor is still responsible for updating many cells in sequence.

#### 2.3.2.4 CEPRA

The CEPRA (Cellular Processing Architecture) series of machines [35] were developed by Rolf Hoffmann et el at the Technische Universität Darmstadt, Germany. The series of machines, of which there are 6, started in 1994 with the CEPRA-8L [38], which consisted of 8 FPGAs. The cell update rule was loaded onto each of the FPGAs allowing the states of 8 cells to be calculated in parallel, enabling data from the host machine to be processed by the CA rules on the FPGAs. The current machine, CEPRA-S (Cellular Processing and Stream Processing) [37] was completed in 2001. It consists of 2 FPGAs, plus data, program and general use memory. One of the FPGAs interprets is a controller which can interpret instructions from memory. It also generates 8 independent addresses for the second FPGA to work on. The second FPGA is the execution unit, it can access and process data from the 8 address in parallel. The FPGAs can be reconfigured for the particular type of application they are running. The CEPRA-S is constructed as a PCI card, which plugs into a standard workstation, allowing it to act as co-processors to perform CA computations.

These machines are optimised to process the CA states, stored in memory, quickly. The machines are only able to process a few cells at a time.

### 2.3.3 Reconfigurable Computing

Reconfigurable computing refers to a combination of hardware and software, where the software configures the functionality of the hardware to perform a particular task (or tasks) at runtime. The software can then interact with the hardware by sending and receiving data to and from the reconfigurable hardware for processing. The software is also able to modify and control the hardware while it is running, for example the software could change the speed of one of the clocks that run the hardware, or it could change the behaviour of one of the hardware functions, or change the interconnections between the functions in hardware. This means that the software is able to reconfigure 'on-the-fly' to meet the needs of the target application. Applications which have been found to be suitable for reconfigurable computing range from 'number crunching' and computing algorithms, to user interfaces, to communication systems interfaces and protocols.

The Field Programmable Gate Array (FPGA), invented by Xilinx, Inc. in 1984, is the basis of the reconfigurable computer which may also include other field-programmable devices including field-programmable interconnects (FPICs) and other devices. FPGAs consist of a number of logical cells whose logical function, and the connections between the cells (routing) are configured (by software) before the device is used.

FPGAs are not only found in reconfigurable computers, they are used in a wide variety of electronic products. They are often used in place of conventional logic gates and Application Specific Integrated Circuits (ASICS) to allow hardware designers to develop and modify circuits after the physical hardware has been built. For example to fix a problem found during testing or add some new functionality, without having to build a new board. FPGA's have been used in harsh environments, such as space, where there is still research to be done to enable new technologies to be used, but where the advantages of being able to reconfigure hardware remotely are high [11]. The fundamental difference between using FPGAs for traditional hardware implementations and reconfigurable computing is that the software that configures the hardware is static, that is the hardware is always configured the same, to perform one specific task whereas in reconfigurable computing the FPGA may be configured differently for each problem, and possibly changed depending on the data input to the problem.

A reconfigurable computing machine therefore consists of an array of logic gates (FPGAs) that can be reconfigured (reprogrammed) to perform different functions. Specifically the logical function of each gate and the interconnection between the gates can be configured. The configuration of the gates can happen both before the function is run, or during the execution of the function. The state of the system may be read directly from the logic gates, or via interface circuitry. A reconfigurable computing machine is generally controlled, and initially programmed using a conventional workstation. The reconfigurable computing machine then acts as a co-processor for the workstation, performing the function that it has been programmed to perform. The host workstation can then be used to monitor the outputs from the reconfigurable computing machine, storing them or displaying them as required.

While FPGAs have been utilised in the STEP chip of CAM-8 (section 2.3.2.1), the firefly machine (section 2.3.2.2) and the CERPRA machines (section 2.3.2.4) where they have been custom programmed to support CA programming, the FPGA in a true reconfigurable computer is a far more general purpose in that the configuration could be made to support CA or other processing operations.

### 2.3.4 Reconfigurable Computing Machines

A number of reconfigurable computing machines have been built from FPGAs including MORRPH and Splash 2. All of them consist of multiple FPGAs connected via simple bus architectures where the FPGAs have the potential for using attached RAM. The architectures are separated from the host processor by several layers of hardware and software, making fine-grained parallel processing between the two systems inefficient. The MORRPH board [17] is an ISA card for PCs that contains six FPGAs. The Splash 2 machine [4] comprises a Sparc-2, an interface board, and up to 16 Splash 2 boards. Each Splash 2 board consists of 17 user configurable Xilinx 4010 FPGAs connected together as a linear array. The FPGAs may also be interconnected through a full crossbar switch. Each FPGA is also connected to 4Mb of local RAM, which is directly accessible by the host. The Splash 2 machine has successfully been used for a number of applications including text searching and image processing.

#### 2.3.4.1 The SPACE Machine

The Scalable Parallel Architecture for Concurrency Experiments (SPACE) Machine is an experimental platform for developing reconfigurable computing applications. It was developed

at the University of Strathclyde, Glasgow, Scotland starting in 1989 [8], [60], [63]. Its successor, SPACE-2, has been developed at the University of South Australia, and was brought into production in the late 1990s. It is the 2-dimensional uniform surface of programmable logic cells that makes SPACE and SPACE-2 (see section 2.3.4.2), ideal for implementing CAs.

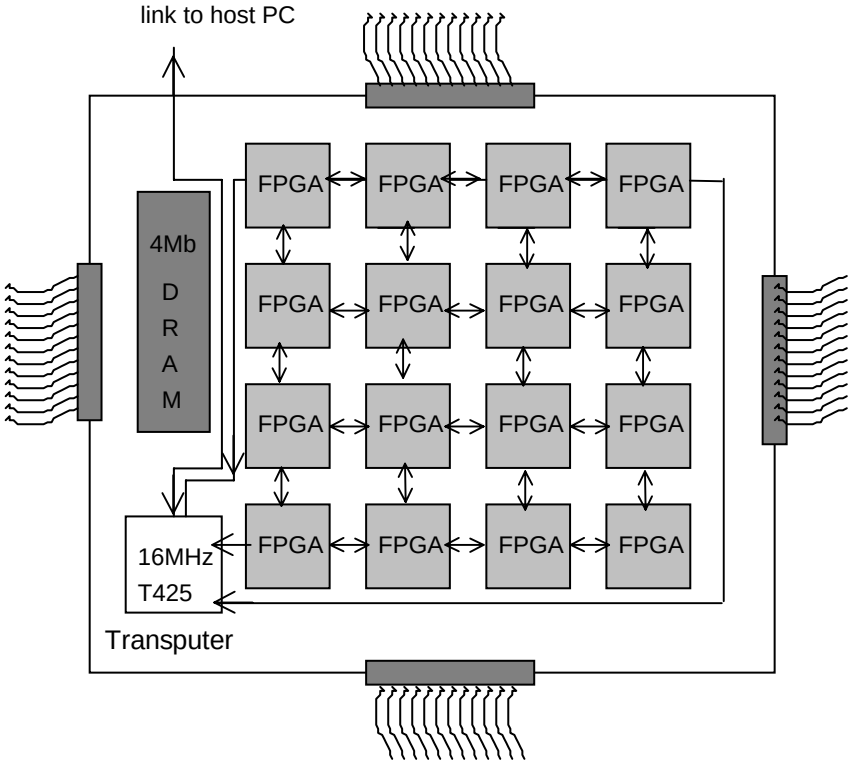



**Figure 2-4 SPACE machine processing board.**

The SPACE architecture is a highly regular, scalable array of logic function cells, as illustrated in Figure 2-4. The SPACE processing board uses 16 CAL1024 FPGAs, whose boundaries are connected in a two-dimensional mesh to achieve genuine continuity in cell-to-cell communications across the board. Links from the 4 chips at board edges are brought out to ribbon cables for virtually transparent expansion across multiple boards. A single SPACE board appears simply as a uniform 128x128 grid of logic/routing cells, each of which can be directly addressed from the host for reconfiguration, reading and setting. Random access allows the computing surface to be partially reconfigured at run-time. During computation state resides locally at the computational nodes, which eliminates contention for centralised memory. Performance is improved by containing most communications to between neighbouring computational nodes.

A Transputer is used to control the DRAM, memory-mapped CAL chips, and programmable clock generators, and to manage data transfers with the external host computer - an IBM-compatible PC, running Linux. In normal operation the Transputer executes a simple operating system for SPACE that provides the host with high-level functions for configuring, testing, and observing the computing surface. Each node of the CAL FPGA provides any 2 input logic function, or a 1 bit memory plus bi-directional communications with neighbouring cells. Figure 2-5 shows an example CAL node configured as a two input or gate with inputs from the SW and NE corners, its output connected to the SE corner and a communication link from the NW to SW.

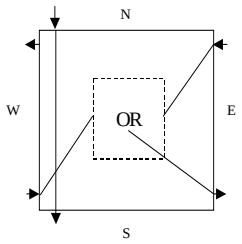



**Figure 2-5 An example configured CAL FPGA node**

The SPACE Machine facilitates the direct representation of CA models on the hardware due to its regular 2 dimensional structure. The compact cell organisation allows many CA cells to be constructed on a single FPGA. The simple logic function, combined with communications links, allow the CA cells to be built so that, when arranged in a grid, they can communicate their state with their neighbouring cells. Similar features are found on its successor SPACE-2.

#### 2.3.4.2 SPACE-2

SPACE-2 is a modular reconfigurable computing platform, comprising of one or more identical processing boards accessible from a host over a 64-bit PCI bus. As in the original SPACE platform, processing boards may be linked together to form computing surfaces of appropriate size. Each processing board contains 8 Xilinx XC6216 (or XC6264) FPGAs arranged in a 2-dimensional toroidal mesh, a maximum of 32Mb of SRAM for general-purpose use, and programmable clock and pulse generators. Each XC6216 configurable cell offers a 2-input gate or multiplexer followed by an optional D-type flip-flop the equivalent of three or four cells on the Algotronix device. Figure 2-6 shows the function unit logic for a XC6216 cell. SPACE-2 boards constructed from the XC6216 makes 32K cells (128K gates) available, growing to 128K cells (256K gates) when constructed from XC6264 FPGAs.

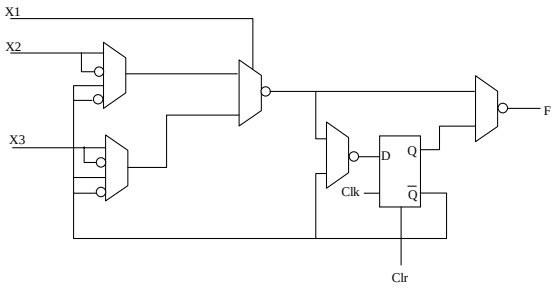



**Figure 2-6 A basic XC6216 function unit**

The control logic on SPACE-2 provides a number of features to support simulations. Clock frequency can be programmed, and set length pulse trains may be generated to run a simulation for some predetermined duration. The board can also be configured to interrupt the host at the conclusion of pulse sequences.

The ability to directly access the hardware on the space machine allows the concurrent nature of CAs to be mapped directly onto the naturally concurrent hardware.

## 2.4 Concurrency

This section explores the importance of concurrency and how CA realized on reconfigurable hardware can provide high speed concurrent computational performance.

Concurrency, meaning to perform more than one operation at the same time, is inherent in almost all natural systems. Natural system are generally constructed from millions of parts, each of which operates all of the time, that is each part of the system is operating at the same time as all of the other parts of the system. For example consider a leaf of a plant, which is made up from millions of cells. Some cells maybe taking sap towards the root of the plant, while others may be absolving sunlight or moisture. The important thing is that all the cells are all working at the same time.

In contrast, conventional computers generally only perform one operation at a time. For example if a conventional computer wanted to simulate the activities of the cells in the leaf it would have to simulate each cell sequentially, so if it was to simulate a million cells performing one operation each it would need to perform a million operations, one for each cell. Many computing algorithms are concurrent in nature, e.g. manipulating all elements of an array, but due to the constraints of the conventional computer need to be performed serially.

Even large supercomputers, like the Connection Machine, are essentially constructed from a number of conventional processors each of which which work on a part of the problem at the same time, hence only allowing relatively small (maybe a few hundred) operations to be performed simultaneously.

In contrast, digital electronics are naturally a lot more concurrent in nature. For example if you take a million and gates and change the inputs to them all, all of the outputs will change at (about) the same time, i.e. they are all working all of the time. Due to their concurrent nature it is natural to construct digital circuits that perform operations concurrently.

It is obvious that if a machine can do many operations at the same time (assuming it can do each operation as fast, or almost as fast) then it will be able to complete the many operations much quicker than the machine that performs its operations serially.

## 2.5 Specifying Cellular Automata

To model and implement CAs it is desirable to specify the behaviour of the cells in a format that can be interpreted by a computer. This section examines several existing specific programming languages that are used for describing and modelling CAs. The section then goes on to discuss how process algebra's, specifically Circal, can be used to describe and model CAs.

### 2.5.1 CA Programming Languages

Several languages exist for programming CAs including; CAM Forth, Cellang, Agents, Cal Cellsim, CAPPET and CDL. All of the CA languages examined provide a syntax to describe a cell update rule. The languages vary in the complexity of the problems that may be expressed. Some of the major differences between the languages expressive power are: Size of the neighbourhood, number of parameters per cell, and the number of states each cell may be in. All of the languages, with the possible exception of CARPET (which has limited support for non-homogeneous rules) restrict the model to a uniform cell rule. While this satisfies the classical definition for CAs, and simplifies the laying out of cells, it restricts the problem domain that these simulators can be used for; in particular these languages can only simulate *homogenous* CAs.

#### 2.5.1.1 CAM Forth

CAM forth [49] is the current Cellular Automata (CA) development language for CAM machines, and is an ongoing project, started in the early 1990's by the Information Mechanics group at the MIT Lab for Computer Science. This language is an extension of the standard Forth language. The CAM forth language has been expanded for the latest CAM machines to allow for models with more than two dimensions. CAM Forth is used to describe the behaviour of the cells. The layout of the cells is set by the construction of the machine. Initialisation of cells is performed using an interactive user interface, which allows users to set cell states prior to running an experiment. To describe the behaviour of a cell in CAM Forth the programmer writes a program to set the value of a pre-defined variable, PLN#, to be the cells 'next state' based on the cells current state and the state of the cells neighbours. Each cell can maintain and generate two additional states, PLN0, and PLN1. The pre-defined variable CENTRE is set to the current state of the cell each time that the program is run. Other pre-defined variables contain the current states of the cell's close neighbours.

#### 2.5.1.2 Agents

An alternative programming paradigm for CAs is that of agents introduced by I. Stephenson [89], [90]. Classically CAs have been described from the viewpoint of the cell space. In the agents paradigm the CA is described from the viewpoint of the interaction between active cells. Each agent may posses a number of parameters that are used to influence their behaviour. The Creatures system [89], [90] describes CAs in terms of active cells (agents) and how they interact with each other, as opposed to describing a rule that is active for every cell in the system. The motivation for agents was to reduce processing times for large systems with relatively few active cells compared to the total number of cells. This saves the time of updating cells that are not affected by the active cells and allow parallel programs to be constructed where active cells are distributed across processors, as opposed to sections of the CA space. The agent paradigm allows more than one agent to be present in a cell at a time. The agents may only interact with other agents that are in the same cell, which unfortunately makes it impractical to implement many real life modelling problems such as traffic type models using cars as agents since it is necessary to know where other cars in the neighbourhood are before moving.

#### 2.5.1.3 Cellang

Cellang was developed as a CA programming language and defined by J.D. Eckart [18], and is an ongoing project which was started in the early 1990's. Cellang is an imperative programming language that forms part of a Cellular Automata Simulation System. The simulation system has a separated viewer that permits the behaviour of the automaton to be observed and permits deterministic rules to be specified. The system allows multi-dimensional spaces to be defined. Cells may contain multiple fields, where each field value is an integer (which may be constrained). The complete system has been implemented to run programs written in Cellang, this includes a compiler, an 'abstract virtual cellular automata machine' and the viewer [19]. The language also allows the use of agent cells (see section 2.5.1.2). A Cellang program consists of two parts being; i) the declaration of the cell structure and ii) a description of the cell rules. The declaration of the cell structure states the number of dimensions of the automata and the number, and types, of states in a cell. Opposite edges of the structure are connected together. The cell rule describes the updates for the cells using arithmetic and binary operators. It uses an array type notation to refer to the cells neighbourhood. Rules may use arithmetic and logical operators, some control statements as well any number of variables when describing a rule. Cellang also provides two other pre-defined variables: *time*, which is incriminated by the system before each cell update, and *random*, which contains a different random number for each cell at each cell update.

#### 2.5.1.4 Cal

Cal (CA language) was developed as a simple CA language for implementing CA models on the Scamper CA simulation system [74]. The neighbourhood of each cell is either; 4 (N,E,S,W), 6 (Hexagonal), or 8 (N, E, S, W, NE, SE, SW, NW). Cal allows commands to be placed into the program that, among other things, can specify the neighbourhood shape, displaying messages, and load the initial cell states from a file. Cal programs consist of a series of statements (rules) that describes when a cell state will move from one state to another. The rules are in two parts; the first part describes the state change from 1 to 0. The second part is the 'mask' that must be satisfied for the state change to occur. The mask consists of a set of values, one for each neighbour, to be tested by the rule. The mask can contain the following wild card characters: i) '?' Match any state, ii) '!' Match any non zero state, iii) 'o' Match any odd state and iv) 'e' match any even state. An example of a rule (for a four cell neighbourhood) that would change a

cell from 0 to 1 if the cells to the East and West are 0 is ‘0 -> 1 : {?,0,?,0}’. Cal also provides two counters that can be used with the mask. The first counter, called the mask counter, is a count of the number of matches between the mask and the neighbourhood. The mask counter may be tested in a rule by following the rule with ‘#’ and the operator and a value. As an example of using the mask counter consider a rule that will change from 1 to 0 if at least three of the N,E,S,W neighbours, in an eight neighbour neighbourhood, are 0. The rule is written: “1 -> 0 : {0,0,0,0,?,?,?,?} #> 6”. There is also another counter, called the state counter, which can be used. This counter is a count of the values of the cell states of the neighbours which are in positions marked by the special character ‘*’ in the mask.

#### 2.5.1.5 Cellsim

Cellsim [44] is a CA programming environment for Sun based UNIX systems. It was developed by Chris Langton and Dave Hiebeler, Los Alamos National Laboratory, New Mexico, USA. The environment allows each CA cell to maintain one parameter which represents up to 256 states. CA cell rules are written in ‘C’ using function templates supplied with the package. A Cellsim program consists of at least two ‘C’ functions. The first function is an initialisation function, which must at least set a global pointer, *update function*, to point to the cell update function. The second function is the cell update function that contains ‘C’ statements to implement the cell rule. The cell update function is called with a pointer to a structure that contains the values of the states of the cells neighbours. The system also provides three other variables: ‘time’ (the current time step) and ‘parm1’ / ‘parm2’ that may be set in the initialisation function for use by the cell update rule. The user may also define two additional functions: a before function and an after function, which are called at each time step before and after updates respectively. These functions receive a pointer to a structure that contains an array with all cell values in it (for analysis). When using 256 states per cell the system calls the *update function* for each cell location at evert time step. If the number of states per cell is restricted to 4, 8, or 16 the system will build a lookup table for the cell rule, reducing the cell update times. The system is also provided with libraries to distribute the cells across nodes on a Connection Machine CM-2.

#### 2.5.1.6 CARPET

CARPET [88] was developed as a programming language to simulate CA in the CAMEL environment (see section 2.3.2.3). The language is based on the C programming language with

additions for CA programming. All CARPET programs follow the same structure. They start with a section that describes the structure of the CA system. This section declares the number of dimensions in the system (this is limited to be between 1 and 3), the CA cell rule radius (i.e. the maximum distance from a cell that the current cells update rule can access), the state variables for the cells, and the neighbourhood (naming which cells the current cell can access). It is also possible to declare global variables within the structure section which can be initialised and are readable by all the cell rules (the global variables can be modified by the user interface, see section 2.3.2.3). Finally follows the cell update rule. This is written using standard C syntax and may have local variables. All cells have the same update rule, but it is possible to access the position of the current cell in the lattice, allowing non homogeneous (see section 2.2.7) CA to be implemented at the expense of performance (as the rule needs to determine which rule to run based upon position before the rule is run).

#### 2.5.1.7 CDL and GCA

CDL (Cellular Description Language) [31] [32] was developed by Christian Hochberger et el at the Technische Universität Darmstadt, Germany to allow simulation of CA on the CERPA (section 2.3.2.4) machines. A compiler has been written [32] to compile CDL directly onto the CERPA hardware. Translators have also been produced to convert CDL to C [31] and Java [99] for the purpose of running on simulators. The language structure is based on the Pascal language. The basic layout of a CDL program is similar to that of CARPET (section 2.5.1.6). There is a declaration section, where the cell states, neighbourhood, cell layout and cell radius are declared. Then there is the cell update rule. All cells use the same update rule. The language does not allow conditional loops to be written, which allows the compiler to unroll all loops when compiling into hardware.

The language has been developed into an object-oriented language, CDL++ [33], which allows cellular objects to be moved around on a grid [33] [34] in a similar way to the agents paradigm described in section 2.5.1.2.

GCA (Global Cellular Automata) [36] is a model for CA which was developed by Rolf Hoffmann et el at the Technische Universität Darmstadt, Germany in 2001. This model extends the definition of CA by not restricting the neighbourhood of the CA cells and allowing the neighbourhood of each cell to change as the state of the cell changes. This model has been

integrated unto CDL by using pointers for the cell neighbours. These pointers can then be changed at runtime to point to any other cell to change the cells neighbourhood.

#### 2.5.1.8 Other General Purpose Languages

Various system based around general purpose programming languages have been used to develop CA. These include JCASim [99] a Java based simulator. StarLogo [78], which is an adaptation of the Logo programming system and allows the user to create agents (see 2.5.1.2) which can interact with each other. A Mathematica extension toolkit [23] allows various types of CA to be constructed in Mathematica. These systems, while generally simple to use, are designed to run on standard computing systems meaning that each cell in the system is updated sequentially, and that the simulations of large systems is much slower than on systems where many cells can be updated in parallel.

### 2.5.2 Process Algebra: The Circal System

The term ‘process algebra’ was coined in 1982 by Bergstra & Klop [3] to describe an existing group of formalisms for modelling concurrent systems including CCS [54], CSP [79] and Circal [55] [56][57][58][59][62] Process algebra now refers to an algebraic approach for the study of concurrent processes. Its tools are algebraical languages for the specification of processes and the formulation of statements about them, together with a calculi for the verification of these statements. There are now a number of process algebra languages including that used by the Circal (CIRcuit CALculus) System. Circal is a high-level language for the specification of concurrent systems [58] and is used in this thesis for describing concurrent systems. It uses a process algebra to rigorously describe and perform verification and simulations of concurrent systems. Each process in Circal has a state in which it can perform or respond to actions. While the diagrammatic representation of processes and states is useful, with systems that are more complex it is advantageous to have a language to describe the system. Circal maps diagrammatic representation of processes and state into a formal syntax. A full description of the Circal syntax can be found in [53]. In summary some of the process operators are (the first five operators are behavioural operators):

```
/ \       (Deadlock) corresponds to a process which has terminated or
deadlocked.
<-        (Definition) is the definition operator.
m P       (Guarding) represents a process which performs action set m and
evolves
```

```
          into process P.
P + Q     (Choice) represents a process which can perform either P or Q.
P & Q     (Non-determinism) represents a process which can perform either P
          or Q but this is decided by the process itself.
P * Q     represents the process which can perform P and Q together.
```

The Circal algebra has been embedded in the high level language XTC. The resulting language is called XCircal, which can also be used as a hardware description language. XCircal has many features derived from the programming language C. Some of the principle features of XCircal which assist in the construction of process models are:

1. Types: ‘Event’ for Circal action sets and ‘Process’ for Circal processes (arrays of these types may also be used)
2. Functions: It is possible to construct parameterised functions, which build, and then return processes.
3. Control Structures: ‘for-loops’ and ‘if’ statements can be used in the construction of processes.

Circal program specifications may be constructed from truth tables (tables where each row specifies an output for a set of inputs, see Figure 4-16 for an example) so that when the program is run using the Circal System, processes, which perform the behaviour described by the truth tables, are constructed and may be manipulated. This makes it possible to use Circal for the formal specification and verification of synchronous and asynchronous digital hardware. The verification mechanism is based on the notion of testing equivalence [53] to ensure that the behavioural specification is identical to the implementation. This ensures that the implementation, which will be realized on the reconfigurable computer, is identical to the behavioural model. Time can be modelled as clock ticks, which can be manipulated in the same way as other processes without any extension to the process algebra framework.

Using the Circal language to specify the behaviour of CAs is straightforward and allows a CA system to be easily modelled. Circal also allows non-homogeneous CAs to be modelled, something that other CA modelling languages do not allow.

Circal has been shown to be useful for modelling complex systems [7], [58], [53], and has already been employed to model complex CA systems [80]. Using process algebra specification

is a very different approach to the conventional cellular automata models, which are generally described using matrices of cells and operations on these matrices. As well as specifying and verifying the implementation of the system, the use of a process algebra leads to an elegant approach for simulating such systems.

## 2.6 Summary

This literature review has identified that CA models are suitable for modelling complex physical systems. It has observed that the model of computation provided by the CA requires that the system is understood and modelled in terms of particles or cells with individual rules. These rules are local requiring interaction only with neighbouring cells. It has also identified that there are many important complex systems that need to be modelled using more appropriate techniques to deal with issues such as accuracy, tractability, complexity etc, and in particular modelling man-made as well as natural systems. Thus, this thesis will explore three different problem areas that require CA solutions including the classic Game of Life, road-traffic simulation and data encryption / decryption.

It has also noted that hardware implementations of CAs pose interesting problems compared to conventional computing architectures. In response to this cellular computing machines have been developed, notably the CAM-8 Machine. The current CA specific hardware is limited in its ability to perform computations that are massively parallel, and they are also limited in the number of cells that they can update each clock cycle. The field of reconfigurable computing is a promising alternative to CA specific machines. The reconfigurable computer maybe configured to a variety of architectures as is necessary for the application. In particular the reconfigurable computer is suitable for implementing CA as they expose the inherent concurrency of the hardware, allowing direct mapping of the concurrent nature of CAs onto the reconfigurable computer. Of the various reconfigurable computing architectures that are in existence the SPACE Machine is the most promising for implementing CAs, particularly due to the regular 2-dimensional layout of its FPGA board. Its architecture allows for all the cells to be directly implemented in hardware, hence all cell states of the CA can be updated each clock cycle, in parallel. This architecture is also well suited to applications, such as CA, where nodes only need to communicate with other nodes that are close to themselves. Thus this thesis will develop philosophies and techniques for implementing of CAs on the SPACE Machine reconfigurable computer.

Finally this review has identified that there are a variety of ways of specifying CA systems. In particular a variety of programming languages designed to specify CAs. Some of these languages are general purpose and others are tied to a particular machine (e.g. CAM-Forth). The review also identified a process algebra based language that is suitable for modelling CA systems. In addition this algebra has the advantage of being both a tool for specification and also for verification. In particular for the specification and subsequent verification of CA processes implemented in digital hardware. Thus this thesis will utilise the Circal System for specifying and verifying the CA models.

# 3. Modelling the Game of Life on the SPACE Machine

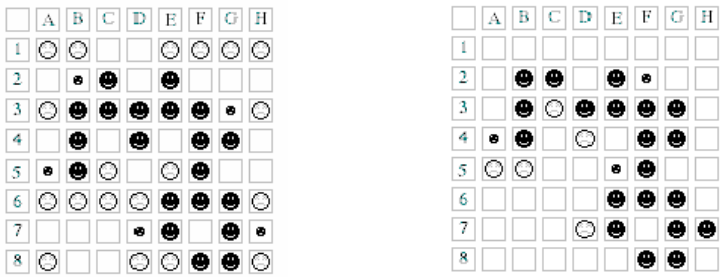
## 3.1 Chapter Overview

The Game of Life is used in this thesis to investigate how to model and implement CA on the SPACE machine. Philosophies and techniques that are developed are used throughout this thesis. The chapter firstly provides a background to the Game of Life, including its origins and its rules. It then describes a general CA model, which is used for all CA in the thesis. The development of Game of Life is then described, including its modelling in Circal and its implementation on the SPACE machine. Finally, a user interface that controls and monitors the Game while it is running on the SPACE machine is presented.

## 3.2 The Game of Life

### 3.2.1 Background and Description

A well known CA model is the ‘Game of Life’ invented by mathematician John Conway, in the 1960's and popularised by Martin Gardener in his Mathematical Games column in the October 1970 issue of Scientific American [22]. It is one of the simplest examples of a CA. This game is an attempt to simulate the effects of congregation, loneliness, and overcrowding among living cells on a 2-dimensional surface with neighbourhoods consisting of the eight adjoining cells, four adjacent orthogonally, four adjacent diagonally. The game starts off with a preset distribution of living cells, the number of which can be a parameter to the game, and evolves through generations. Each cell on the 2-dimensional surface is either alive or dead. The game progresses in discrete time steps. All cells share the same update rule. An alive cell will remain alive at the next time step if it has exactly two or three neighbours that are alive. A new cell will be born, in a dead cell, if that cell has exactly three neighbours who are alive.

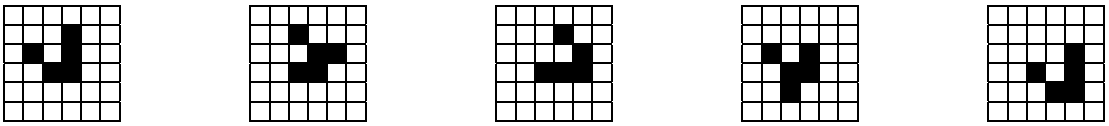


**State S(t)** **State S(t+1)**

**Figure 3-1 Showing snapshot of Game of LIFE from State S(t) to State S(t+1)**

Figure 3-1 shows two snapshots of the Game of Life between states S(t) and S(t+1). The earlier state on the left of the figure shows 20 living cells (large black smiles) and 19 (white frowns) that have died from the previous round and 5 (small black smiles) have been born. Moving to the next state on the right there have been 5 deaths as cells have been either lonely or overcrowded. In particular the cell at location 5A in state S(t+1) would have died from loneliness and the cell at 3C from overcrowding.

Despite the simplicity of the rules governing the changes of state as the automaton moves from one generation to the next, the evolution of such a system is complex indeed. There is a rich mathematical structure that can only be appreciated fully by watching the intricate behaviour that arises from such a simple set of rules. Many starting patterns grow chaotically for a while and then stabilise. Other patterns emerge which are ‘self-reproductive’, in the sense that they do not change shape from generation to generation. Some configurations may alternate between two different patterns from generation to generation and some configurations contain patterns that appear to slowly crawl across the lattice, such as the famous glider pattern, as illustrated Figure 3-2, which repeats at time step S(t) and S(t+4) moving one square diagonally every four time steps.



S(t) S(t+1) S(t+2) S(t+3) S(t+4)

**Figure 3-2 The Glider Pattern**

### 3.2.2 The Rules of the Game

The rules for the game of life, as stated by M. Gardner [22] are:

1. Survivals. Every cell with two or three neighbouring cells which are alive survives for the next generation.
2. Deaths. Each cell with four or more neighbours which are alive dies from overpopulation. Every cell with one or no neighbours dies from isolation.
3. Births. Each empty cell adjacent to exactly three alive neighbours-no more, no fewer-is a birth cell.

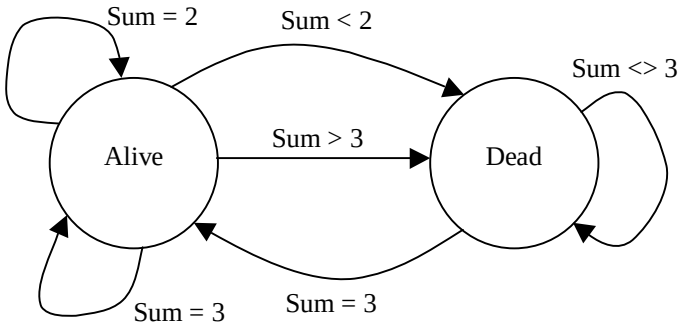



**Figure 3-3 Transition Diagram for the Game of Life**

Figure 3-3 shows a state transition diagram describing the rules of the game of life. The value *sum* is the number of neighbours of that cell that are still alive. Clearly a cell will die of isolation (when sum < 2) or from overcrowding (sum >=3) as described in Conway's original game.

## 3.3 Designing a Cellular Automata for 'Life' on the SPACE Machine

### 3.3.1 Introduction

Before developing a CA for the game of life on the SPACE machine a generalized CA model, which can be used for implementing any CA model is developed.

### 3.3.2 General Model for a CA Cell

As stated in section 2.2.2, a CA consists of a set of rules that define how a cell transitions between states. This section presents a general model that separates the cell behaviour rules from the motion of transitioning between states (at each time step), this allows the behaviour of

the cell to be modelled, and implemented, without having to constantly remodel, and implement, the transition between time steps. This model will be the basis for the implementation of the game of life, presented later in this chapter, and as a basis for the other CA implementations presented in this thesis. The concept of time can be separated from the cell behaviour rule, allowing each to be modelled as separate processes. One process describes the behaviour of the cell and consists of a rule that generates the state of a cell at the next time step from the current states of itself and its neighbours. The other process is the updating, or time, process that maintains the current cell state and replaces its state with the new state at each clock step. Figure 3-4 illustrates the decomposition of a CA cell into these two processes. In summary they operate as follows:

1. The first process called 'cell behaviour' takes two sets of inputs: i) the current state of its neighbours (the output of their time processes) and ii) its own current state. This process generates one output being the state of the cell for the next time step i.e. State (t+1). This process will be different for each type of CA cell modelled.
2. The second process called 'time' takes two inputs being: i) the new cell state, State (t+1), generated by the cell behaviour and ii) the clock. It outputs the current state i.e. State (t) which is replaced by the new state at each clock tick. This process definition is the same for each type of CA cell modelled and advances the overall model by one time step. The time processes are all clocked by a global process.

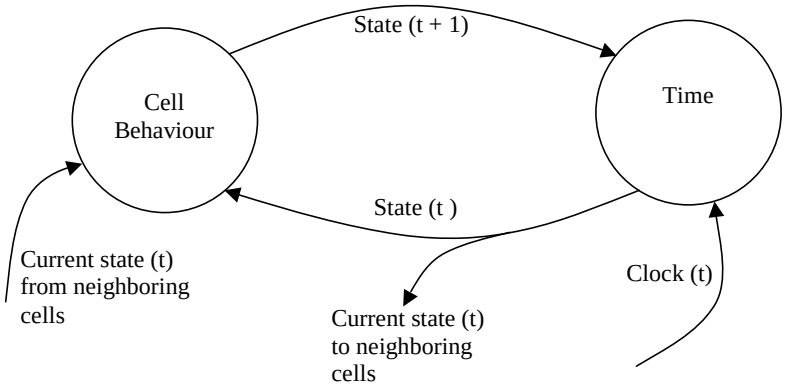



**Figure 3-4 State Transition Diagram for Updating a Cell's State**

As stated the time process will be identical for all CA cells types with only the cell behaviour differing, which is the Time process can be composed with any cell behaviour to form a CA cell. The time process, defined in Circal, is shown below (E is the empty, or dead, state and F the full, or alive, state).

```
E <- t E + (t nextstate) F
F <- (t currentstate) E + (t currentstate nextstate) F
```

Where the clock ticks, t, can be generated by the following Circal process:

```
T <- t T′
T′ <- t T
```

The time process can be implemented in logic on the SPACE machine with two follow / hold latches as shown in Figure 3-5.

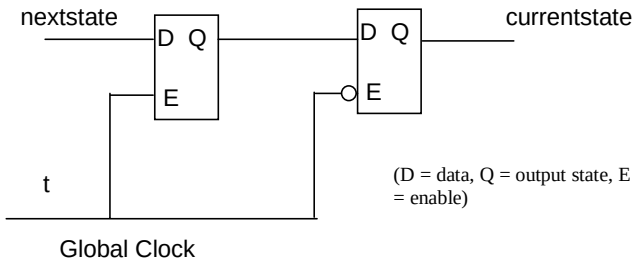



**Figure 3-5 Implementation of the Time process for the SPACE machine**

## 3.4 Cell behaviour for 'Life' on the SPACE Machine

### 3.4.1 Overview

It is necessary to express the rules of the Game of Life in such as way that individual cells can be modelled and individually implemented. In particular it is necessary to devise a way of expressing their logic such that cells may be laid out on the regular structure of the SPACE machine. Two alternative implementations are considered.

### 3.4.2 Direct Implementation of a Cell

The first implementation that was considered was a direct implementation of a cell, in logic, of the behavioural rule. The current state of the cells eight neighbours are labelled *a* to *h* and *S(t)* is the state of the cell at the current time with *S(t+1)* the state at the next time period. The following logic expression describes the behaviour of a cell in the Game of Life moving from time t to time t+1 (neighbours states are denoted a,b,d,…).

$$S(t+1) = (a\&b\&c + a\&b\&d + \ldots + f\&g\&h) + ((a\&b + a\&c + \ldots + g\&h) \,\&\, S(t))$$

That is the state of the cell at S(t+1) will be alive if any combination of 3 neighbours are alive, or the cell itself is alive and any two neighbours are alive. This model was *not* implemented due to the large number of functions in the expression. To implement this directly using two input logic gates would require: 28 AND gates for all combinations of two inputs (second expression), 56 additional AND gates to increase this to all combinations of three inputs (first expression), 85 OR gates and an AND gate to add the final S(t) expression which equals 170 two input gates. While it may be possible to collapse the gate count for this implementation it was considered that given the limited number of gates available on the FPGAs the gate count would not be reduced to a level that would allow a reasonable number of life cells to be implemented, and so this method was not used.

### 3.4.3 Incremental Implementation

An alternative algorithm that can be implemented is to maintain two logic lines; one that is high if exactly two inputs are active and one that is active if exactly three inputs are high. The algorithm generates these two lines by adding each input (neighbours state), one at a time, to a 'running total' and calculating the number of inputs that are active. The behaviour is modelled using two processes, P and R. Process P performs an incremental count. It takes the count from a previous P process and the state of a neighbour as input and calculates a new count as its output. A P process is hence associated with each input. The R process takes the count from the final P process and the current state of the cell as input and produces the next state of the cell as its output.

The P process makes use of the fact that it is not necessary to calculate counts that are greater than three as higher counts all lead to a dead cell. Figure 3-6 illustrates the P process that takes counts 0, 1, 2 or 3 as input together with the current state from a particular neighbour (e.g. NE, N, NW) and produces counts 0, 1, 2 or 3, as output.

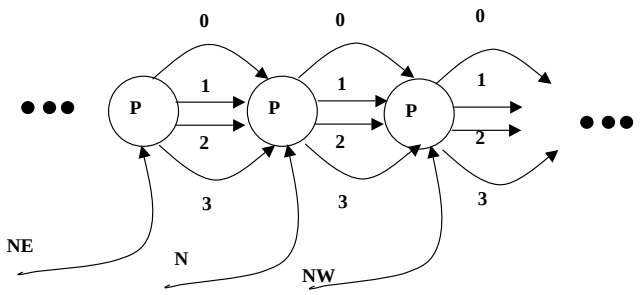


**Figure 3-6 Illustration of Inputs / Outputs of Process P**

The p process behaviour can be described in Circal as follows:

```
P <- P + (zeroIn zeroOut)P + (oneIn oneOut)P + (twoIn twoOut)P +
         (threeIn threeOut)P + (zeroIn input oneOut)P + (oneIn input
    twoOut)P +
          (twoIn input threeOut)P + (threeIn input )P + (input)P
```

To implement the cell behaviour process P requires just ten two input logic gates. Eight of these processes can be composed together to produce a process which generates a count of all eight neighbouring states. The first three P process in the chain can be simplified as the maximum possible input count (to the process) is less than three. The final two stages can also be simplified as only a sum of two or three is of interest. The final gate count for the simplified eight input P processes is 56.

The second behaviour process of a life is the R process, which produces the next state of the cell from the current state and the output of the P processes.

This process is described Circal as follows (note that currentstate and nextstate are state of the cell generated at the last clock tick and the new state of the cell respectively, as described for the time process in section 3.3.1):

```
R <- R + (twoOut)R                              /* no birth, not three
neighbours */
       + (threeOut nextstate)R                  /* birth */
       + (currentstate)R                        /* death, <2 or >3 neighbours
*/
       + (twoOut currentstate  nextstate)R      /* survival */
       + (threeOut currentstate  nextstate)R    /* survival */
```

Taking the events that produce a ‘nextstate’ leads to the following logic equations where twoOut and threeOut are the inputs from the previous P process:

```
(!twoOut & threeOut & !currentstate) OR (twoOut & !threeOut & currentstate)
OR
(!twoOut & threeOut & currentstate)
          ≡
 threeOut OR (twoOut & currentState)
```

Thus the R process can be implemented with one AND gate and one OR gate. This makes a total of 58 gates to implement the life behaviour model, where a life cell is a composition of eight P process (some of which are simplified) an R process, and a Time process.

As stated the implementation is the logic circuit derived from the eight P processes and one R process. The logic for the first two P process can be combined and simplified as:

```
Zero: is high when neither input is high (a NOR b),
One: is high when only one input is high (a XOR b),
Two: is high when both inputs are high (a AND b).
```

The remaining P processes calculate the total number of inputs that are high from the previous P process and the state of the next neighbours input. For example, if the next neighbours input is high (the neighbour is alive) then the ‘new count’ is now ‘old count + 1’, if it is low (the neighbour is dead) then ‘new count equals old count’. Given that the highest count that is required to be detected is 3 it is not necessary to keep track of counts higher than this. It is also possible to remove some of the circuitry from the last couple of inputs that would generate the 'Zero' and 'One' lines as these are not required by the algorithm. The resulting circuit is shown in Figure 3-7. The circuit consists of 58 two input gates plus a flip-flop to capture the new state.

The logic for the process was also modelled in Circal and its equivalence with the behavioural model proved using the Circal system.

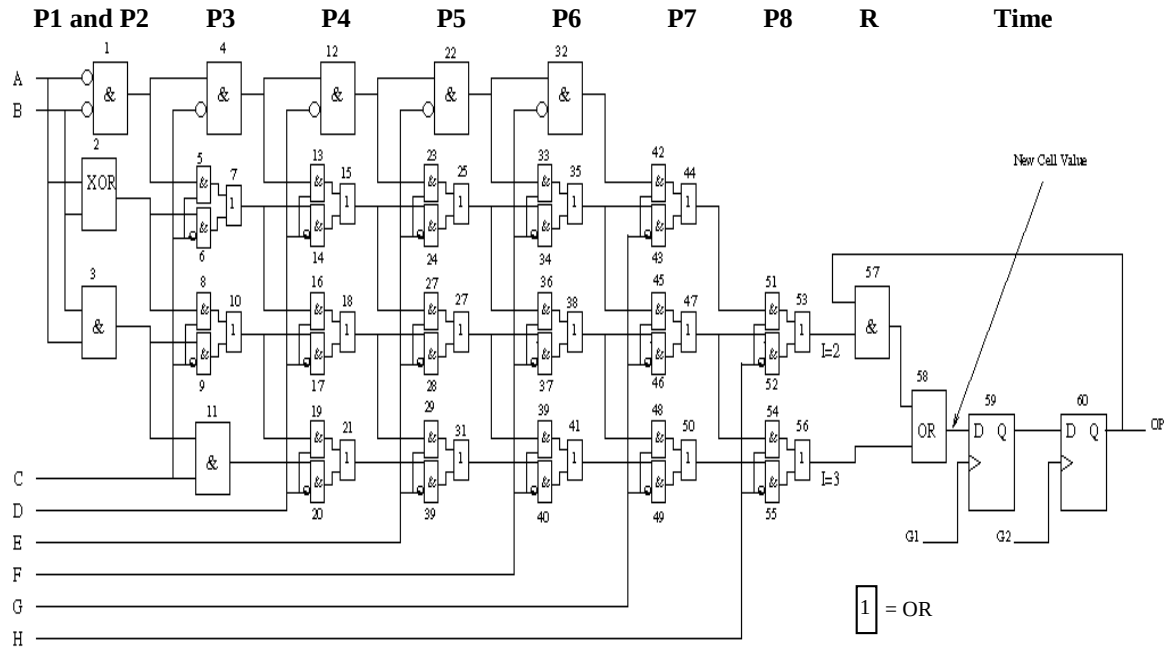



**Figure 3-7 Life Cell Circuit Diagram**

### 3.4.4 Realization on the SPACE machine

The circuit in Figure 3-7 was manually placed into a grid (13x7), or tile of FPGA cells. Each tile had its current state routed to its four corners. Each tile also provided routing across its corners to allow its neighbours to access the states of the cells on their diagonals. When laying out a circuit on SPACE it is necessary to ensure that the pairs of master slave flip flops remained physically close together on the same FPGA. This is because of clock skew and propagation delays. A program was then written to place the life cells onto SPACE machine.

Initialising or modifying the state of the cells is not straightforward. The current state of each cell in the game is maintained by the pair of master slave flip-flops. The first of these flip-flops is in the hold state when the clock is idle (low), and holds the current state of the cell. Unfortunately it is not possible to set the state of a flip-flop to low if the data into the flip-flop is high. So to set the state of the flip-flop requires that the preceding gates output must be reconfigured to a logic 0. The state of the cell can then be set, and the previous cell can then be reconfigured back to its original function.

### 3.4.5 A User Interface

In order to control and allow the cell states of the Game of Life to be viewed a graphical user interface was developed. The user interface consists of a 17 x 9 cell display grid where each

location in the grid represents a cell within the game. A white circle was used to represent a 'dead' and a black circle an 'alive' cell. Figure 3-8 illustrates a screen shot from the Game of Life initialised with a glider, which is a pattern that moves slowly across the grid of cells as the game runs. The board may be initialised to a pre-defined configuration, random or other. The user interface controls the clock to the SPACE machine, clocking it at slow speed to allow the outputs to be observed. The interface reads the states of all the cells between each clock cycle and displays their values.

To control the game the user is provided with three control buttons, one to exit the program (exit), one to switch the clock off (setup) and another to start it running (run). It is possible to change the state of cells either when the clock is off (to initialise the game), or while the game is running. To change the state of a cell the user clicks on that cell, which changes the state of the cell (on the SPACE board as well as on the interface). If the game is running the cell state is changed between board clock cycles.

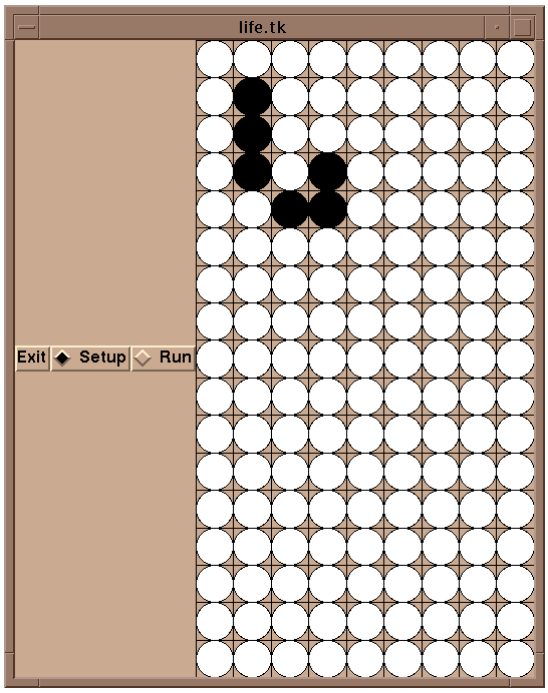



**Figure 3-8 Screen Shot from the Game of Life**

Tcl/Tk [73] was chosen as the language to implement the user interface. Tcl/Tk was chosen as it provides an interface to access to functions written in ‘C’, which are required to access the SPACE operating system, and it allows graphical user interfaces to be built quickly. The development of the user interface was in two parts :

1. Porting the SPACE host interface to Tcl commands. The Tcl host commands are used to control the clock, download the circuit, read the cell states and to reconfigure the board to set cell states.
2. Implementing the graphical user interface as a Tcl/Tk script.

## 3.5 Summary

This chapter has explored the basic development of a CA model for the Game of Life that is implemented on the SPACE Machine. In particular, a general model for a cell of the automaton is suggested based on decomposing the cell into a behavioural process and a time process, allowing the behaviour to be expressed as its own process without concern for time. This way of modelling the CA naturally lends itself to a formal specification in the Circal system, as Circal allows the process to be separately specified and then composed together to form larger systems, allowing the time process model to be reused for other CA implementations. The design of digital circuits implementing the formal Circal process descriptions was verified and the validity against the behavioural model. The CA cells are placed on the SPACE machine in a regular grid with a common interface between all cells. This common interface makes it possible to change the behaviour of individual cells, either statically (that is the system can be stopped and an initial pattern setup) or dynamically, (the states of the cells can be changed while the simulation is running). The next chapter explores this potential of implementing the CA model on a parallel reconfigurable computing machine in the domain of road traffic modelling.

# 4. Simulating Road Traffic on the SPACE Machine Using Cellular Automata

## 4.1 Chapter Overview

More complex CA models of freeway traffic flows are presented in this chapter. Theses models are used to further develop the modelling of CA behaviours using Circal, and then, to develop models of the digital implementation of the behavioural models, again using Circal, proving that the implementation is equivalent to the behavioural models.

This chapter first reviews some existing traffic simulation systems, implemented both in software and in hardware. It then presents the models that are to be used for simulating multi lane freeway traffic. Circal behavioural models and models of the logic to implement them on the SPACE are then presented, along with examples of how these models can be used to simulate road traffic using the Circal system. The models are then realized onto the SPACE machine, and a user interface, which allows CA cells to be interactively placed onto the SPACE machine, and the states of the cells to be observed in real time, is presented. The chapter concludes with a discussion of the results from the experiments run in this chapter.

## 4.2 Modelling Road Traffic

### 4.2.1 The Traffic Modelling Problem

Road traffic modelling is important for a number of reasons. Two main application areas of traffic modelling might be distinguished as: i) real time traffic control and ii) longer term planning. In the first instance urban traffic control systems are required to meet transport objectives by the adaptive control of road space over time. In particular to modify network parameters in real-time, such as traffic signal timing, to minimise delays particularly in light of abnormal conditions both in urban centres and on arterial roads. For example, we might wish to know the consequences of diverting traffic due to a vehicle breakdown in the middle lane of a freeway during rush hour. In this application the importance of faster than real time modelling or simulation is paramount. It is necessary to explore the effects of decisions and know their consequences in order to take appropriate on-line action. In the second instance traffic simulation is useful for longer term ‘what if’ planning questions in road system planning, evaluation and operation management. Simulation enables parameters affecting the traffic to be controlled and systematically examined so helping decision-makers to arrive at a precise diagnosis and compare the cost-benefits of various traffic planning alternatives. Factors such as

congestion, travel time, number of stops and fuel consumption can be estimated and optimal configurations estimated from modelling.

Depending upon the exact physical road system being addressed it might be necessary to model single lane or multi-lane traffic, urban centres or freeways, slip road intersections or junctions or perhaps some other aspect of the road network. In terms of traffic modelling 1-dimensional traffic modelling has been used to refer to single-lane roads, while 2-dimensional refers to traffic on a 2-dimensional grid. These modelling tasks may present different problems. Some modelling techniques for single-lane modelling have actually been supported by data from multi-lane highways although it is thought that more complex methods are needed for multiple-lane highways. In particular lane changing rules are known to cause problems in microscopic simulations [52] since the decision to change lane is not based on knowledge from the nearest cars only, but include information from cars further away, while possible such information is seldom used in microscopic modelling due to the increased complexity it would introduce to the models. Nagel suggested that homogeneous multi-lane traffic simulation (where there are symmetrical lane changing rules and all vehicles are the same) behaves similarly to single-lane models, although deviations from homogeneity would introduce more and more additional effects [69]. The complexity of traffic behaviour is part of the modelling challenge, especially when modelling requires not only fidelity but must also happen in real time.

Traffic shows some characteristics of a complex system. There are some conditions where the state of the system is deterministic such as when there is a steady light flow of traffic along a road where there are no obstructions, hence the next state of the system will simply see each car move forward on the same section of road. There is chaotic behaviour, such as can be seen in a congested city centre, where it is not possible to determine the behaviour of the system from simple calculations, the system must be modelled to discover how it will respond. The evolution of a particular traffic system is affected by many things. Small incidents on a busy motorway such as someone braking too hard or changing lane badly can multiply in a near chaotic fashion to cause traffic jams and snarl up traffic flow. In the same way a single traffic light failing at rush hour can cause multiple knock-on effects in a city centre causing gridlock. Additionally the interaction of these many individual phenomena makes the future evolution of traffic systems particularly hard to predict and simulation can assist in this process. It would be interesting to consider what causes instability and hence worsens traffic movement, and for this reason a model that enables such complex systems to be understood would be valuable to increase our understanding of traffic flows and what actions are likely to improve traffic flows for various

situations. As with other complex physical systems there is a choice between modelling methods basically stemming from a mathematical versus CA based approach.

Traffic flow is concerned with finding relationships between velocity of a vehicle, the density of vehicles and the throughput or flow [24]. In traffic science the fundamental diagram represents the relationship between these three variables. There are a variety of ways of simulating traffic flow. They may be divided into static (e.g. constraint based) and dynamic systems. The dynamic systems can be macroscopic, microscopic or mesoscopic. Macroscopic models aggregate individual vehicle behaviour into analytic flow equations (e.g. Boltzmann equations). Macroscopic models simulate the road network and produce less accurate information on traffic flow. Microscopic systems track individual vehicles as they move along the roads interacting with other vehicles and traffic signals (eg. CA). Many mathematical microscopic car-following theories are continuous and typically concentrate on single-lane carriageways giving detailed information about the loadings. The particle hopping or CA based approach deals with discrete space and time. The use of CA in traffic simulation was first suggested by Gerlough [26]. A final class of mesoscopic models may also be identified which models discrete vehicles but calculates time using flow equations and delay queues [50].

### 4.2.2 Traffic Simulation Systems

There are several examples of traffic simulation systems already in existence. The Paramics software suite [51] developed in consultation with Edinburgh-based transport consultants SIAS Ltd, and Quadstone offers a traffic simulation capability where traffic is modelled as individual vehicles to produce congestion and flow patterns. This simulation package can be used to study individual junctions or large-scale traffic patterns including facilities for handling roadside information displays to enable real-time traffic control system design and control. Quadstone and SIAS have used Paramics on a number of traffic planning and control projects for national, regional and local government in the UK. While Paramics permits cars to be transferred between roadways it is only possible for one car to move from a particular roadway to another during any clock tick, restricting the realism of the simulation for congested road.

The traffic network simulation model TRITRAM (Traveller Information and Traffic Management) [50], developed jointly by CSIRO and the Roads and Traffic Authority of New South Wales is designed to enable real time adaptive traffic control in urban arterial road networks with signalised intersections. In particular TRITRAM is aimed to enable modelling many intersections and uses an object-oriented process-based approach. The TRITRAM road

network is described in terms of Nodes (intersections) and Links (roadway between intersections). TRITRAM incorporates microscopic and mesoscopic models of traffic flow. The TRITRAM system can be run using parallel architectures.

MULTSIM [27] is designed as an investigative tool to allow the effects of changes to the travel environment to be assessed. MULTSIM simulates the driver of a vehicle and provides them with a set of goals. They are released down the road subject to rules that govern driver behaviour. The rules include behaviour of other vehicles and features of the driving environment. It assumes that flow patterns on the road can be realistically represented if the behaviour of individual drivers can be modelled with a reasonable degree of accuracy. It handles traffic travelling in one direction along a divided road. MULTSIM has been used to test the effects of dynamic advisory speed signs with respect to travel time, number of stops and fuel consumption. MULTSIM relies upon an accurate modelling of vehicle and driver behaviour, which may vary considerably between regions.

The main requirement of traffic management systems is that traffic simulation happens in better than real time and faithfully models complex road networks over a large scale, perhaps the whole of a large metropolitan area. The complex nature of such a physical systems means that the large scale, real time, and road traffic-modelling problem is still a very real challenge. Two research teams have considered the problems of high-speed traffic simulation of large networks at the University of Maryland [92] and the University of Edinburgh [51] and both systems are based on Connection Machine implementations. Junctions and road links are not without their problems in these models. This thesis addresses the modelling problem by designing CAs for large complex multi-lane traffic networks, and in order to achieve better than real time processing, the parallel, reconfigurable, FPGA based architecture of the SPACE Machine is used, so addressing the challenge of simulating large complex systems in real time.

### 4.2.3 Cellular Automata and Traffic Simulation

The original use of CA for traffic models [26] in 1956 aimed at making the simulation fast enough to be useful for real time traffic applications. It utilised a single-bit coding and fairly sophisticated driving rules, however, the bit-coded implementation made it impractical for many traffic applications. The model incorporates randomisation so that three different properties of human driving can be taken into account (fluctuations at maximum speed, over reactions at breaking and retarded acceleration) [69].

Nagal and Schreckenberg modelled 1-dimensional traffic. Nagal [68] shows how a Boolean model for traffic flow reaches a critical state by itself in a bottleneck situation. Each cell is a 7.5m road segment whose interpretation is the length of car plus the space between them in a jam. Each cell may be full or may be empty and vehicles have a velocity from 0 to max. One update of the system consists of four separate steps that are performed in parallel for all vehicles. These steps are: i) acceleration - where the velocity may advance if there are no cars ahead, ii) slowing down - where the velocity will decrease when cars are sighted, iii) randomisation - to reduce the vehicles velocity on a probabilistic basis and iv) car motion - advancing the vehicle forward a number of states. Through these rules the general properties of single lane traffic may be modelled by CA. Step iii) was found to be essential in simulating realistic traffic flow otherwise the system was deterministic. An extension to multi-lane traffic is the subject of further mathematical investigation.

Wagner [98] is concerned about comparing the CA model with empirical data. Wagner observes that it is possible to design microscopic lane changing rules that give the correct macroscopic behaviour regarding lane usage when compared with flow curves. Wagner compares the CA models with data from German roadways. In particular the microscopic CA model was found to assist with the macroscopic lane changing phenomena, where typically information from more distant cells is needed to determine whether to change lane or not. The rules permit lane change only if there is no other car that would be hindered by the change. Wanting to change lane is simulated by hindrance in progression along the current lane.

### 4.2.4 Traffic Simulation and FPGAs

The Judge (Just Designed to be Good Enough) traffic modelling system is designed to simulate traffic flows in an urban environment [83], in particular to provide rapid response to non-recurring congestion from unusual incidents. The modelling technique is based upon modelling each vehicle in the network as a separate entity. Traditional methods included complex driver behaviour, vehicle performance, road lane widths and other information and the JUDGE system tries to minimise the amount of data that is actually relevant in making a simulation with view to address the rapid prediction requirement. To model traffic the JUDGE system breaks roadways down into 6m segments that correspond to small cells, which can hold a vehicle. With every time tick the vehicle will move ahead one cell provided that cell is empty. Fast and slow cells are introduced to effectively model the speed with which vehicles would progress and assist with modelling traffic flow. Junctions are accounted for where vehicles can enter and

leave a cell from any of the four directions; otherwise roads may join by simple abutment. In particular the model is implemented in FPGA-based hardware. The FPGA-based approach converts the network components into circuits and runs them in parallel allowing many of the vehicle movements to be made in parallel without the expense of a Connection Machine. They achieve a 1 hour simulation of a large city network in milliseconds of real-time, although the accuracy of the simulations is hard to estimate.

Another hardware implementation at the University of Strathclyde [60], [80] utilised the Circal system for modelling. In particular for modelling urban road junctions with circuits designed and executed on the SPACE Machine. The hardware array is tiled with many finite state automata (a CA type model) corresponding to a particular section of road. In particular the roadway is split up into a number of sections (cells), each cell representing a length of road (i.e. 5 metres), which is either occupied by a vehicle or not. A car could advance to the next cell if that cell was empty, otherwise should remain in its current cell. The model is advanced at discrete time steps. With the traffic simulation model the road cell is the only component to embody state and it either contains a car or not. Communications between a car and other components are achieved by four lines synchronised with a clock tick. The lines are used to pass a car in or out of the cell. This means that cars travel along a roadway with at least a single car space between them.

The work in this thesis extends the traffic simulation model of University of Strathclyde [80]. In particular it models a multi-lane freeway, together with on ramps and off ramps, which permit traffic to join and leave the freeway. The vehicles change lane as speed and traffic conditions change or as they progress and enter or leave the freeway. It is assumed that all vehicles travel at the same maximum speed if the road is not blocked. A CA is designed to model this behaviour and the rules are modelled using the Circal system (and subsequently implemented on the SPACE machine). The model is advanced at discrete time steps. Each time step represents the time it takes a car to travel the length of one road cell.

## 4.3 Multi-Lane Traffic Simulation with CA

### 4.3.1 Cells of the Automaton

A basic model for a CA cell has been determined in 3.3 where different models of CA were considered. The model suggested is one consisting of 2 processes. One process being the cell’s

behaviour while the other is the timing process. This basic model of a cell will be utilised in these traffic experiments (see Figure 3-4).

A number of different CA behaviour models are designed to simulate different aspect of freeway traffic flow in a simple but realistic manner. The models are then composed together to form sections of freeway. The set of rules for each of the CA type was derived from our knowledge and observations of traffic flow on freeways. These rules are used to simulate where a vehicle will move to on the next update depending on the state of the neighbouring cells (eg. a vehicle may change lanes if the cell in front is full). The rules of vehicle movement are then transposed into cell rules, which determine whether a cell will be full or empty at the next clock tick depending on the state of its neighbouring cells (eg. an empty cell may become full if the cell behind is occupied). In particular the set of road cells consists of the following cell types with associated behaviour:

1. Basic road cell: allows vehicles to only travel forward in the same lane.
2. Overtaking cell: as the basic road cell plus it allows vehicles to move out one lane if their lane is blocked.
3. Pull in cell: as for the basic road cell plus it allows vehicles to move in one lane if that lane is clear.
4. On ramp: allows vehicles to merge into the main traffic flow
5. Off ramp: allows vehicles to leave the main traffic flow
6. Slow cell: simulates slow sections of roadway

The freeway model is constructed by connecting a number of these cells together to form each lane of the freeway. Figure 4-1 shows a typical section of a three-lane freeway with a single lane on ramp and off ramp constructed from traffic cells. A block of cell types (eg overtaking cells) are generally placed across the whole width of the freeway, at various points, to faithfully mimic the structure of the physical system allowing vehicles to interact with vehicles in adjoining lanes (as in overtaking, for example)

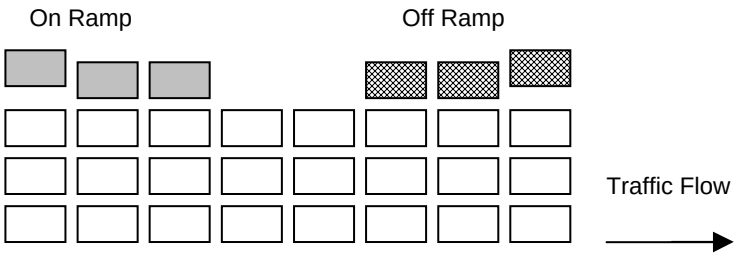



**Figure 4-1 Three Lane Freeway**

### 4.3.2 Basic Road Cell

The basic road cell has the same behaviour as the basic road cell developed at the University of Strathclyde [80]. The basic road cell is the building block for other road cell types. The rule for the cell is:

> *An empty cell will fill if the cell behind is full, otherwise it will remain empty,* i.e. the 'vehicle' in the cell behind will move forward into this cell if it is empty.
>
> *A full cell will empty if the cell in front is empty, otherwise it will remain full,* i.e. the 'vehicle' in this cell will move forward if the cell in front is empty.

The basic cell is further discussed in section 4.4.3.

### 4.3.3 The Inter lane Processes

Inter-lane processes do not simulate the location of vehicles (state), they act only as routing to allow vehicles to traverse lanes according to a set of rules, that is they do not "contain" vehicles, they just facilitate and control the sideways movement of vehicles. They are used to allow vehicles to interact with other vehicles in adjoining lanes (as in overtaking, for example) and are placed between basic road cells. Inter-lane processes communicate with their neighbouring inter-lane processes (across the lanes) as well as with the road cells in front and behind. Figure 4-2 shows how the inter-lane processes interface with the basic road cells. The inter-lane processes can be thought of as being composed with the preceding basic road cells to form a block of road cells (with state) that interact across a number of lanes. These composed cells form a multi-lane road where traffic can flow between lanes.

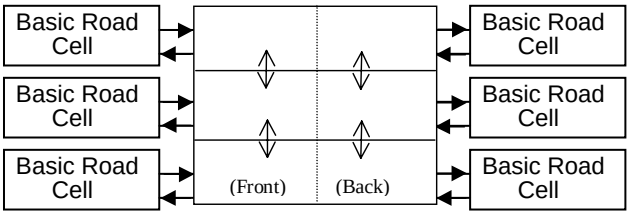



**Figure 4-2 Inter-lane Processes and Basic Road Cells**

The inputs to the basic road cell determine whether an empty cell will fill or a full cell will empty. The inter-lane processes generate these inputs from the current states of a number of neighbouring cells. They are constructed as two processes; one process decides if a full cell (from the preceding basic road cell) is able to empty. This process is called the front process. Its output is connected to the next input of the preceding cell. The other process is called back and decides if an empty cell (in a following basic road cell) will fill. The output of this process is connected to the prev input of the following cell. It was found that the behaviour for these two processes could be independently derived from the inter-lane process rules.

### 4.3.4 Overtaking and Pulling In

Given that a multi-lane carriageway can be constructed there is now the possibility that a car may change lanes. Cells, based on the UK / Australian open road, where vehicles travel in the left lane unless overtaking, are developed. Two types of cell are developed, an overtaking cell and a pull in cell. These cells are distributed along the carriageway. Through experiment it was found that placing an overtaking or pull in cell between every five to ten basic road cells produced traffic flows similar to data collected from real traffic flows. Placing slightly more overtaking cells than pull in cells reduced the number of lane changing cells that were required to achieve realistic traffic flows.

Firstly an overtaking cell is developed. This cell will allow a car to move forward if the cell in front is free. The overtaking function allows cars to continue travelling forward if the cell in front is not empty by pulling out to the lane on their right, if there exists space to do so. Figure 4-3 illustrates the preferred direction of movement for vehicles that are travelling forward with the possibility of overtaking. The behaviour of cars through the overtaking function can then be stated as:

1. A car will move forward if the cell in front is empty (preferred direction of travel)

2. A car that cannot move forward will move forward right if that cell and the cell to the right is empty (a car on the right has priority)
3. A car that cannot move forward or forward right will remain in its cell

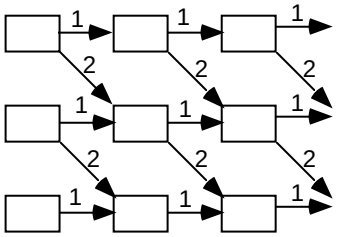


**Figure 4-3 Preferred Direction of forward movement for vehicles starting overtaking**

The pull-in cell allows cars to pull in if there is space in the lane to their left. A car in this cell has priority to move into the cell directly in front of it, but prefers to move into the cell forward and to the left. A car will move forward and left if the cell to the left is free and the cell forward and left is free. If a car cannot move forward and left and the cell in front is free it will move forward.

Figure 4-4 illustrates the preferred direction of movement for vehicles completing their pulling in. The behaviour of cars through the pull-in function is summarised by the following:

1. A car will move forward left if that cell and the cell to left (this car has priority on the cell) are empty (preferred direction of travel)
2. A car that cannot move forward left will move forward if the cell in front is empty
3. A car that cannot move forward left or forward will remain in its cell

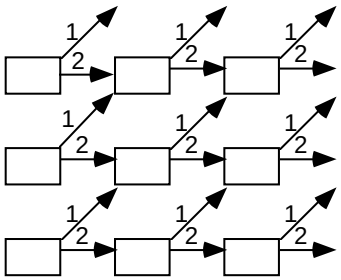


**Figure 4-4 Preferred direction of forward movement for vehicles pulling in**

These rules can also be expressed from the viewpoint of the cell, which is what is required for the behaviour process. The overtaking cell behaviour can be rewritten as:

1. A empty cell will fill if the cell behind is full or the cells behind left and left are full

2. A full cell will empty if the cell in front is empty or the cells right and forward right are empty.

The pull in behaviour can be also be rewritten as:

1. A empty cell will fill if the cell back right is full and the cell behind is empty, or the cell behind is full and either the cell back left or the cell left is full.
2. A full cell will empty if either the cell in front is empty or the cells to the left and forward left are empty.

### 4.3.5 On Ramps

The on-ramp allows cars to merge into a main carriageway from the left hand side. It models vehicles that are on a slip road about to join the main carriageway. The on ramp may be any number of lanes wide and can be connected to a section of multi-lane road. For this definition the on-ramp consists of groups of cells that span the main carriageway and the on-ramp itself, for the section of roadway where the on ramp interacts with the main carriageway. Figure 4-5 illustrates how a two-lane on-ramp merges with a three-lane carriageway. Each cell in the figure corresponds with a basic road cell and an on-ramp function; the arrows indicate the preferred direction of travel.

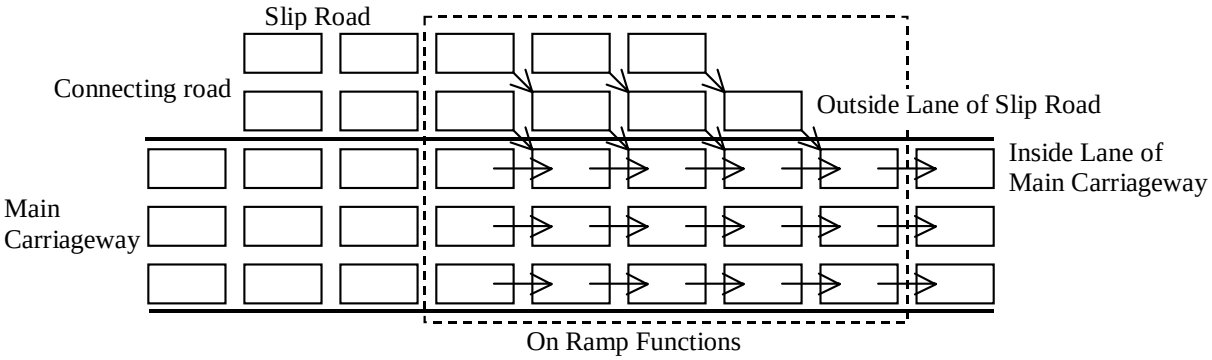



**Figure 4-5 An example of on-ramp modelling**

In this model cars that are in the outside lane of the slip road have priority to move into a space on the inside lane of the carriageway; this simulates the on ramps slowing the flow on the main carriageway. The cars in the outside lane of the on ramp will move forward if the inside lane of the main carriageway is blocked. Cars on the inside lane of the main carriageway do not have priority on any cells, but can move forward or forward right. Cars on the inside lane of the slip ramp have priority to move forward and a preference to move forward right. The cars in the

outside lanes (all lanes except the inside lane) of the main carriageway use the rules from the overtaking function. Importantly four component functions can be identified for the on-ramp function: outside lane of the slip road, inside lane of the main carriageway, other (outside) lanes of the main carriageway and other (inside) lanes of the slip road.

## 4.4 Implementing Multi Lane CA Traffic Simulation on the SPACE Machine

### 4.4.1 Specification in Circal and Digital Implementation

Circal process algebra is used as both a modelling and a verification tool to demonstrate that each CA behaviour process is a faithful model of the intended behaviour and that the implementation in digital logic is correct. Firstly the behaviour of the cell types is modelled in Circal. Each of the behavioural models is then simulated, using the Circal System [1], against a number of test patterns, and the results examined against the original behaviour descriptions to establish correctness. Circal composition was also used in simulation to connect the road cells together to form short sections of roadway that were stimulated with test data to confirm that the integrations produced the expected traffic behaviour. A simple program was written to display the results of these simulations in a meaningful way (as shown in Appendix 2: Results of Traffic Simulation Experiments).

The cell behaviour was modelled in Circal using a truth table model. The cells are modelled using the Circal pre-defined type Bool. The Bool type maps onto logic design better than plain Circal Events and also explicitly shows where a ‘lack of a signal’ is significant.

Each of the cell types was constructed as a rectangular ‘tile’ for placement on the SPACE machine’s computing surface. The states of neighbouring cells, and a global clock are taken as inputs, with the current state of the cell being it’s only output. All tiles are three FPGA cells high (across the carriageway) and feature a standard interface on their left/right (along the carriageway) boundaries. Tiles may interface to any other type of tile on their left/right boundary. This allows different cell types to be connected together in a ‘lane’ (traffic flowing left-right). However they have different interfaces on their top/bottom edges, forcing the same type of cell to be placed across a series of cells (eg. overtaking cells should span the width of a carriageway). Variants of the cell types were created for the edges of the carriageway, which prevented vehicles from moving onto or off of the carriageway.

The digital circuits were themselves modelled in Circal by composing Circal models of the constituent logic gates together. The Circal System was able to automatically perform verification by equivalence proof between the behavioural and logic processes, proving that the specification and implementation processes are functionally and temporally identical. A graphical user interface was developed to allow tiles to be placed interactively onto the SPACE machine, and to monitor the cell states as the simulation runs. This was used to test the models on the SPACE machine, and to perform simulations of larger freeway models than was possible using Circal alone.

### 4.4.2 Time Process

As described in 3.3.1 the basic operation of a cell in a CA may be split into 2 processes: that is, the cell logic process and a time process. Figure 3-4 illustrates the finite state machine for this modelling. The time process is considered first since this will be common to all other cells. The process advances the model from one time step to the next. It takes nextstate and clock ticks as input and produces the current state as the output.

The time process sets it output, Currentstate (Cs) to the state of the input, Nextstate (Ns) at each tick of the global clock (t). The process then maintains this state until the next global clock tick. A Circal process encapsulating this logic is shown below. This process has been shown to be equivalent to the behavioural model using the Circal system verification by equivalence proof.

```
1: Process TimeLogic (Bool Ns,Cs){
2:   static Bool ns,cs,int1,int2
3:   static Process T(ns,cs,t,int1,int2)
4:   T = (CDLATCH(t,ns,cs))
5:   return ((T[Ns/ns,Cs/cs]) - (int1 int2))
6: }
```

The Time process can be implemented using a single D type flip-flop or two cascaded follow-hold latches. Figure 4-6 illustrates the connectivity of the time logic with a cells behaviour logic (C). Note prev is the current state of the previous neighbouring cell, next is the current state of the subsequent neighbouring cell, cs is the current state of this cell and ns is the next state for this cell.

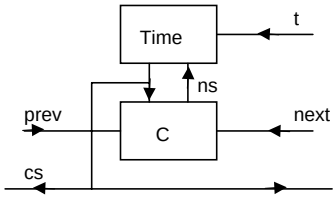



**Figure 4-6 General Cell Construction**

### 4.4.3 The Basic Road Cell

The Basic Road Cell is the fundamental building block for all of the other kinds of cell. The state of this cell (Full of Empty), at the next time step, is determined from these inputs as shown in Figure 4-7 where F = full and E = empty. The rule for the cell's behaviour is:

1. An empty cell will fill if the cell behind is full, otherwise it will remain empty.
2. A full cell will empty if the cell in front is empty, otherwise it will remain full.

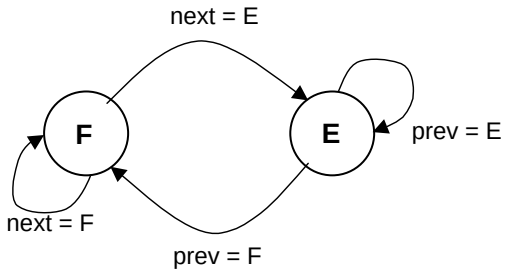



**Figure 4-7 State transition diagram for the basic road cell behaviour**

The cell has two inputs, 'prev' and 'next' which represent the current state of the previous and next cells, physically adjacent. Its own state 'currentstate' is also an input. The output is the next state of the cell. A process encapsulating this behaviour is shown below.

```
1: Process BasicRoadCell (Bool Next, Prev, Cs) {
2:   static Bool ns,cs,next,prev
3:   static Process C
4:   C <- (next.0 prev.0 cs.0 ns.0) C +
5:        (next.1 prev.0 cs.0 ns.0) C +
6:        (next.0 prev.1 cs.0 ns.1) C +
7:        (next.1 prev.1 cs.0 ns.1) C +
8:        (next.0 prev.0 cs.1 ns.0) C +
9:        (next.1 prev.0 cs.1 ns.1) C +
10:       (next.0 prev.1 cs.1 ns.0) C +
11:       (next.1 prev.1 cs.1 ns.1) C
```

```
12:  return ((C * TimeLogic(ns,cs))[Next/next,Prev/prev, Cs/cs] - ns)
13:}
```

The digital logic for the cell is also modelled using Circal. A process encapsulating this digital logic is shown below. This process has been shown to be equivalent to the behavioural model using the Circal system verification by equivalence proof.

```
1: Process BasicRoadCellLogic (Bool Next, Prev, Cs) {
2:   static Bool ns,cs,next,prev,int1,int2
3:   static Process C(next, prev, ns, cs ,int1, int2, int3, t)
4:   C = (AND(cs,next,int1) * X1BARAND(cs,prev,int2) *
5:                                          OR(int1, int2, ns))
6:   return ((C * TimeLogic(ns,cs))[Next/next,Prev/prev, Cs/cs] -
7:                  (ns int1 int2 int3 ))
8: }
```

The basic road cell requires two AND gates followed by an OR gate together with the time logic. The circuit diagram for the basic road cell is shown in Figure 4-8. The Time process has been constructed from two follow / hold latches instead of one D type flip flop as it is intended to implement the road cell on the SPACE machine which only has follow / hold latches. The basic road cell has been implemented on a tile 2 cells wide.

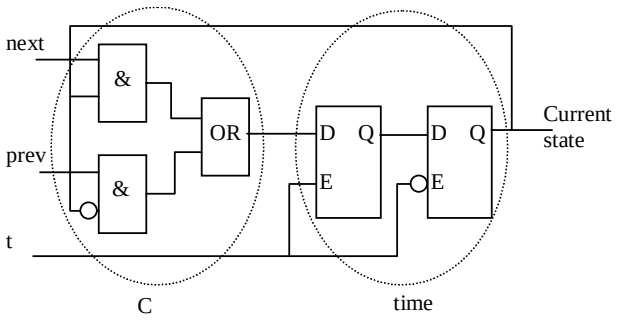



**Figure 4-8 Logic Circuit for a Basic Road Cell**

### 4.4.4 Building a Carriage Way

#### 4.4.4.1 Single Lane Sections

To construct a section of roadway using Circal a number of road cells must be composed together. This section illustrates how to construct a simple single lane carriageway by composing a number of basic road cells together.

Figure 4-9 shows how a number of basic road cells may be connected together to form a section road.

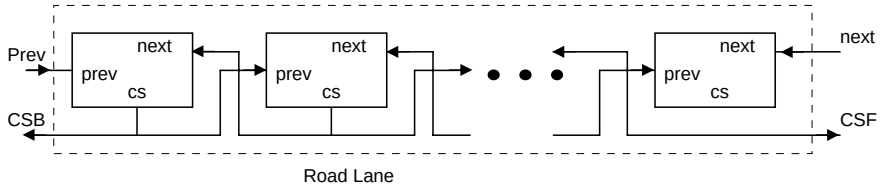



**Figure 4-9 Basic Road Cells form the Carriage Way**

The road section has prev and next inputs as before, but now has two outputs (CSF and CSB) representing the current state of the cells at the front (CSF) and the back (CSB) of the road section. A Circal function that implements a single lane carriageway is shown below:

```
1: Process RoadLane(Bool next,prev,CSB,CSF,int length){
2:   Process lane
3:   int i=0
4:   Bool states[length]
5:
6:   // 1st cell
7:   lane = BasicRoadCell (states[1], prev, states[0])
8:
9:   // compose middle cells together
10:  for (i=1; i<(length-1); i++)
11:    lane = lane * BasicRoadCell(states[i+1], states[i-1],states[i])
12:
13:  // last cell
14:  i = length-1
15:  lane = lane * BasicRoadCell(next, states[i-1],states[i])
16:
17:  lane = lane[CSB/states[0],CSF/states[i]]
18:
19:  for (i=0; i<(length-1); i++)
20:    lane = lane - states[i]
21:
22:  return lane
23:
```

Length is the number of road cells in the carriageway. It is possible to simulate carriageways that are any number cells in length.

#### 4.4.4.2 Multi Lane Road Sections

Multi-lane sections of carriageway can be constructed in arrays of processes. A section of multi-lane road, where there is no interaction between the lanes may be constructed as a number of single lane roads side by side. A Circal process for constructing this is shown below. The inputs and outputs for this section are arrays, with one entry for each lane.

```
1: Process BasicRoadCellBlock(Bool Prev[],Next[],CSB[],CSF[],int length){
2:
3:   n = sizeof Prev
4:   if (n<=1 || n!=sizeof Next || n!=sizeof CSB || n!=sizeof CSF)
5:     print "Bad argument(s) to BasicRoadCellBlock\n"
6:
7:   Process BRCB[n]
8:   for(i=0;i<=(n-1);i++)
9:     BRCB[i] = RoadLane(Next[i],Prev[i],CSB[i],CSF[i],length)
10:  return (*BRCB)
11:}
```

The program generates a process consisting of n identical single lane road sections. The inputs and outputs for this section are connected to arrays, with one entry for each lane. The size of the arrays is used to determine how many lanes to construct (line 3). The other arrays are checked to ensure they are the same size (lines 4, 5). An array of processes, one for each lane, is then constructed (lines 8, 9). The composition of all lanes is then returned (line 10). In order to make it possible for the multi-lane road sections to interact and vehicles to swap between the lanes it is necessary to develop the inter-lane functions that permit communication between the carriageways. These inter-lane processes will consist of a ‘front’ and a ‘back’ behaviour process (see 4.3.3), composed with a basic road cell.

### 4.4.5 Overtaking Cell

As shown in Figure 4-2 it is necessary to define the behaviour for the front and back of a cell. The cells behaviour is outlined in section 4.3.4. The behaviour of these cells was modelled as Process OvertakeFront and Process OvertakeBack from the behaviour, using the same technique as for the basic road cell. The logic descriptions developed which implement the behaviour. The behaviour and implementation were proven to be equivalent. The Circal representation for digital logic is shown below:

```
1: Process OvertakeFrontLogic(Bool Front,FrontR, Right, Next) {
```

```
2:   static Bool front, frontR, right, next, int1
3:   static Process C
4:   C = (OR(frontR,right,int1) * AND(int1,front,next))
5:   return (C[Front/front, FrontR/frontR, Right/right, Next/next]-int1)
6: }

1: Process OvertakeBackLogic (Bool Back, Left, BackL, Prev) {
2:   static Bool back, left, backL, prev, int1
3:   static Process C
4:   C = (AND(left,backL,int1) * OR(int1,back,prev))
5:   return (C[Back/back, Left/left, BackL/backL, Prev/prev] - int1)
6: }
```

The circuit diagram for the overtaking cell processes, back, front, basic road cell and time is shown in

Figure 4-10.

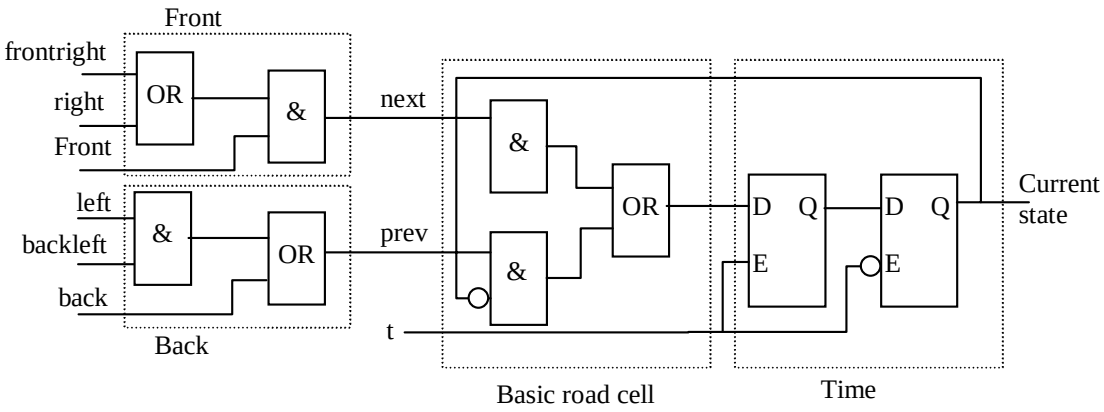



**Figure 4-10 Logic for an Overtaking Cell**

The overtaking road cell, composed with a basic road cell, has been constructed into a tile 3 cells wide. A multi-lane road that permits overtaking may be constructed using the overtaking cell and the basic road cells. This is performed by the following Circal process:

```
1: Process OvertakeBlock (Bool Prev[],Next[],CSB[],CSF[]){
2:
3:   n = sizeof Prev
4:   if (n <= 1 || n != sizeof Next || n != sizeof CSB || n!=sizeof CSF)
5:     print "Bad argument(s) to OvertakeBlock\n"
6:
```

```
7:    Bool one, zero
8:    Process OTB[n]
9:
10:   OTB[0] = (OvertakeBack(Prev[0],one,zero,CSF[0]) *
11:               OvertakeFront(Next[0],Next[1],Prev[1],CSB[0]))
12:
13:   for(i=1;i<(n-1);i++)
14:     OTB[i] = (OvertakeBack(Prev[i],Next[i-1],Prev[i-1],CSF[i]) *
15:                OvertakeFront(Next[i],Next[i+1],Prev[i+1],CSB[i]))
16:
17:    i = n-1
18:    OTB[i] = (OvertakeBack(Prev[i],Next[i-1],Prev[i-1],CSF[i]) *
19:                OvertakeFront(Next[i],one,one,CSB[i]))
20:    return ((*OTB * ONE(one) * ZERO(zero)) - (one zero))
21: }
```

Note that the inside and outside lanes have terminating constants to prevent cars leaving or entering from outside the carriageway (lines 10 and 19 respectively).

### 4.4.6 Pulling in Cell

The pulling in cell was constructed in a similar manor to the overtaking processes described above. The cells behaviour is outlined in section 4.3.4. The behaviour was again modelled and logic descriptions developed which implement the behaviour. The behaviour and implementation were proven to be equivalent using the Circal system verification by equivalence proof. The Circal representation for digital logic is shown below:

```
1: Process PullInFrontLogic (Bool Front, FrontL, Left, Next) {
2:    static Bool front, frontL, left, next, int1
3:    static Process C
4:    C = OR(left, frontL, int1) * AND(int1, front, next)
5:    return (C[Front/front, FrontL/frontL, Left/left,Next/next] - int1 )
6: }
```

```
1: Process PullInBackLogic (Bool Back, BackR, Left, BackL, Prev) {
2:    static Bool back,backL,backR,left,prev,int1,int2,int3
3:    static Process C
4:    C = OR(backL, left, int1) * AND(int1, back, int2) *
5:              X1BARAND(back, backR,int3) * OR(int2, int3, prev)
6:    return (C[Back/back, BackR/backR, Prev/prev,
7:                Left/left, BackL/backL] ) - (int1 int2 int3)
```

```
8: }
```

The circuit diagram for the front and back processes of the pulling in cell are shown in Figure 4-11.

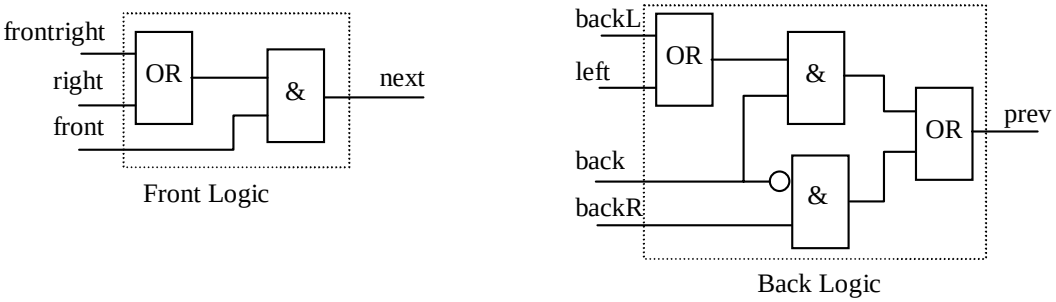



**Figure 4-11 Logic Circuit for the Front and Back of a Pulling in Cell**

A multi-lane road section may be constructed in the same way as for the overtaking cell.

## 4.4.7 On-Ramp

### 4.4.7.1 Overview

As described in section 4.3.5 different cell behaviour rules are required depending upon the position of the cell on the carriageway. Four different sets of behavioural processes are required:

1. One for the inside lane of the main carriageway which interacts directly with the on ramp,
2. One for the other lanes of the main carriageway,
3. One for the outside lane of the slip road which interacts directly with the main carriageway,
4. One for the other lanes of the slip road.

The following sections define the front and back behaviour and implementation processes for these cells. The front process is connected between to the output of the cell and determines whether the cell can empty into any of its neighbouring cells. The back process for a cell is connected to the input of the cell and determines whether the cell can be filled from any of its neighbouring cells.

### 4.4.7.2 Main Carriageway Inside Lane

Cars in the inside lane of the main carriageway do not have priority to move in any direction but they will move forward, which is their first preference, or forward and right, if it is clear.

The front process informs the proceeding cell that the cell can empty. The cell can empty (into the following cell) if the following cell is empty and the cell to its left (on the slip ramp) is empty (the slip ramp has priority). The cell can also empty (into the front right cell) if the front right cell is empty and the cell to its right is empty.

The back process informs the following cell if it will fill. There are two possible cells that could fill the following cell; back and back left. A full cell back left has priority on the following cell and will fill the following cell if it is empty. A full cell behind will also fill this cell (preferred direction of travel) if it is empty, and the cell back left is empty.

The inside lane of main carriageway front and back processes can each be implemented in logic. The logic for the front process is for when the cell will stay full, as this is the logic state for the 'next' input to the basic road cell, indicating that the vehicle is blocked from moving. Processes encapsulating this logic are shown below. These processes have been shown to be equivalent to the behavioural models using the Circal system verification by equivalence proof.

```
1: Process SlipInLane0FrontLogic(Bool Front, FrontR, Right, Left, Next) {
2: // Inside lane of main carriageway
3:   static Bool front, frontR, right, next, left, int1, int2, int3, int4
4:   static Process C
5:   C=(OR(frontR,right,int1) * AND(int1,front,int2) * INV(front,int3) *
6:          AND(int3,left,int4) * OR(int2,int4,next))
7:   return (C[Front/front, FrontR/frontR, Right/right, Left/left,
8:  Next/next] - (int1 int2 int3 int4))
9: }
```

```
1: Process SlipInLane0BackLogic (Bool Back, BackL, Prev) {
2: // Inside lane of main carriageway
3:   static Bool back, backL, prev
4:   static Process C
5:   C = (OR(back,backL,prev))
6:   return (C[Back/back, BackL/backL, Prev/prev] )
7: }
```

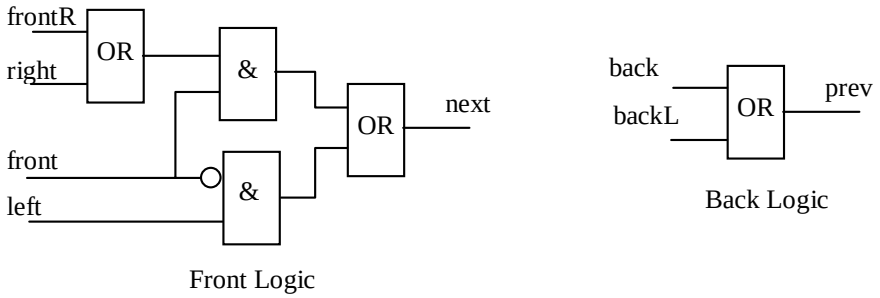



**Figure 4-12 Logic Circuit for the Front and Back of the Main Carriageway Inside Lane**

As stated in section 4.3.5 the other lanes of the main carriageway are simply overtaking processes.

#### 4.4.7.3 Outside Lane of the On Ramp

The outside lane of the on ramp has priority to move into the inside lane of the main carriageway, which is its first preference, and priority to move forward on the slip ramp.

The front process will allow the following cell to empty if either the cell to the front right, or the cell directly in front is empty.

The back process informs the following cell if it will fill. There are two possible cells that could fill the following cell; back and back left. A full cell behind has priority on this cell and will fill it if it is empty and the cell forward right is full. A full cell back left cause this cell to fill if the cell behind is empty (which has priority).

The outside lane of the slip road front and back processes are both implemented in logic. Processes encapsulating this logic are shown below. These processes have been shown to be equivalent to the behavioural models using the Circal system verification by equivalence proof.

```
1: Process SlipInSlip0FrontLogic (Bool Front, FrontR, Next) {
2: // Outside lane of sliproad
3:   static Bool front, frontR, next
4:   static Process C
5:   C = (AND(frontR,front,next))
6:   return (C[Front/front, FrontR/frontR, Next/next])
7: }
```

```
1: Process SlipInSlip0BackLogic(Bool Back, BackL, Right, Prev) {
2: // Outside lane of sliproad
4:   static Bool back, backL, right, prev, int1, int2
```

```
5:   static Process C
7:   C = (AND(back,right,int1) * X1BARAND(back,backL,int2) *
8:         OR(int1,int2,prev))
10:  return (C[Back/back,BackL/backL,Right/right,Prev/prev]-(int1 int2))
11:}
```

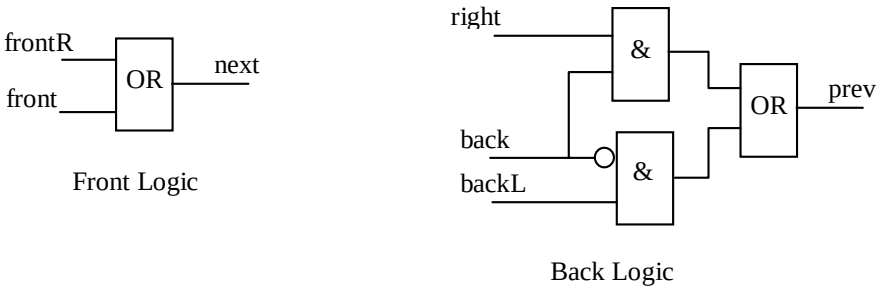



**Figure 4-13 Logic Circuit for the Front and Back of the Outside Lane of the On Ramp**

#### 4.4.7.4 Inside Lanes of the On Ramp

Cars in the inside lanes (that is all lanes except the outside lane) of the on ramp have priority to move forward but a preference to move forward and right, towards the main carriageway.

The front process will allow the following cell to empty if either the cell in front is empty or the cells to its right and forward right are empty.

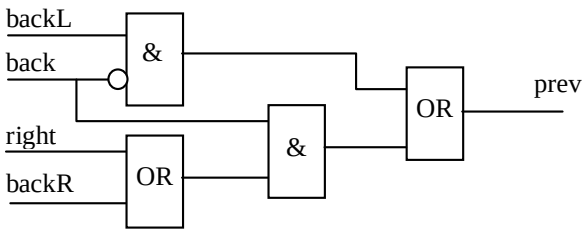
The back process informs the following cell if it will fill. There are two possible cells that could fill the following cell; back and back left. A full cell behind has priority on this cell and will fill it if it is empty and either of the cells to the right and forward right are full. A full cell back left cause this cell to fill if the cell behind is empty (which has priority).

The inside lanes of the slip road front and back processes are both implemented in logic. Processes encapsulating this logic are shown below. These processes have been shown to be equivalent to the behavioural models using the Circal system verification by equivalence proof.

```
1: Process SlipInSlipNFrontLogic(Bool Front,Right,FrontR,Next){
2: // Inside lane(s) of sliproad
3:   return OvertakeFrontLogic(Front, FrontR, Right, Next)
4: }
```

```
1: Process SlipInSlipNBackLogic (Bool Back, BackL, Right,BackR, Prev) {
2: // Inside lane(s) of sliproad
3:   static Bool back, backL, right, backR, prev, int1, int2, int3
```

```
4:   static Process C
5:   C = (X1BARAND(back, backL, int1) * OR(right,backR,int2) *
6:         AND(back,int2,int3) * OR(int1,int3,prev))
7:   return (C[Back/back, BackL/backL, Right/right, BackR/backR,
8:                    Prev/prev] - (int1 int2 int3))
9: }
```




Back Logic

**Figure 4-14 Logic Circuit for the Back of the Outside Lane of the On Ramp**

#### 4.4.7.5 Making the Carriageway

To construct a section of multilane carriageway with an on-ramp to the side the preceding processes must be composed together to form a function that spans the whole carriageway and slip road. This is performed by the following Circal process :

```
1: Process SlipInBlock (Bool Prev[],Next[],CSB[],CSF[],int lanes){
2:
3:   n = sizeof Prev
4:   if (lanes<=1 || n!=sizeof Next || n!=sizeof CSB ||
5:                                    n!=sizeof CSF || n<=lanes)
6:     print "Bad argument(s) to OvertakeBlock\n"
7:
8:   Bool one, zero
9:   Process SIB[n]
10:  Process ONe = ONE(one)
11:  Process ZERo = ZERO(zero)
12:
13:  SIB[0] = (SlipInLane0Back(Prev[0], Prev[lanes], CSF[0]) *
14:            SlipInLane0FrontLogic(Next[0], Next[1], Prev[1],
15:                Prev[lanes],CSB[0]))
16:  for(i=1;i<(lanes-1);i++)
17:    SIB[i] = (OvertakeBack(Prev[i],Next[i-1],Prev[i-1],CSF[i]) *
18:              OvertakeFront(Next[i],Next[i+1],Prev[i+1],CSB[i]))
19:
20:   i = lanes-1
```

```
21:   SIB[i] = (OvertakeBack(Prev[i],Next[i-1],Prev[i-1],CSF[i]) *
22:               OvertakeFront(Next[i],one,one,CSB[i]))
23:
24:   if(lanes == (n-1)) {
25:     SIB[lanes]=(SlipInSlip0Back(Prev[lanes],zero,Next[0],CSF[lanes]))
26:              * SlipInSlip0Front (Next[lanes],Next[0],CSB[lanes])
27:   }
28:   else {
29:     SIB[lanes] = (SlipInSlip0Back (Prev[lanes], Prev[lanes+1],
30:               Next[0], CSF[lanes])) *
31:               SlipInSlip0Front (Next[lanes],Next[0], CSB[lanes])
32:
33:     for(i=lanes+1;i<(n-1);i++){
34:        SIB[i]=(SlipInSlipNFront(Next[i],Prev[i-1],Next[i-1],CSB[i]) *
35:                    SlipInSlipNBack (Prev[i], Prev[i+1],
36:                     Next[i-1], Prev[i-1],CSF[i]))
37:     }
38:
39:     i=n-1
40:      SIB[i]=(SlipInSlipNFront(Next[i],Prev[i-1],Next[i-1],CSB[i]) *
41:             SlipInSlipNBack(Prev[i],zero,Next[i-1],Prev[i-1],CSF[i]))
42:   }
43:   return ((*SIB * ONE(one) * ZERO(zero)) - (one zero))
44:}
```

### 4.4.8 Off-Ramp

The off ramp consists of a pull-in block that spans the main carriageway and all lanes of the off-ramp. This is modified so that the outside lane of the off-ramp will block cars from moving onto it for a set percentage of the time. This is constructed using a counter that forces the cell to appear full, to the lane to its right, for some of its count.

### 4.4.9 Slow Cell

On certain sections of roadway the speed of traffic may be restricted. This may be a permanent restriction, such as a steep hill, or it may be temporary - for example, due to an accident. To model this phenomenon a cell, in which it takes a vehicle two clock ticks to pass through, is used. It therefore makes the vehicle travels at half of the maximum speed. The circuitry for the slow cell is shown in Figure 4-15.

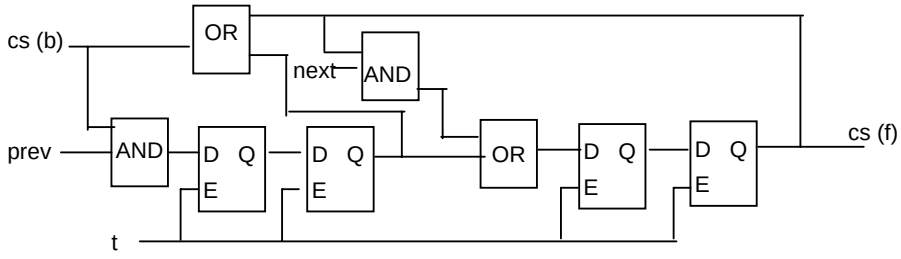



**Figure 4-15 Slow cell circuit**

### 4.4.10 Long Road Cells

The current road traffic cells model roads at a microscopic level. This produces a high fidelity model in which every car in the system is modelled. Unfortunately modelling traffic networks at this level consumes a lot of hardware, hence restricting the size of problem that can physically be placed on the computing surface. To address this problem road cells that behave as a series of basic road cells with a restricted set of internal values are constructed. The internal values chosen were all zeros (an empty section of road) 50% zeros / 50% ones, and all ones (stationary traffic). The road section to be simulated could be up to n road cells in length where n is an even number (to permit the 50/50 rule to be implemented). The new cell to simulate this road section, called ‘LongRoad’, is clocked by a signal that is 1/n the frequency of the clock that was clocking the basic road cells. The functionality of the road section is to simulate a corresponding section of roadway constructed from basic road cells over the limited number of states. To simulate this a truth table was constructed with the values of; the proceeding long cell, this long cell, and the next long cell. The truth table is shown in Figure 4-16 where ‘Next’ is the state of ‘Here’ at the next clock tick, 0 = all cells empty, 1 = 50/50 empty/full and 2 = all cells full.

| Back | Here | Fwd | Next |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 2 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 2 | 1 |
| 0 | 2 | 0 | 1 |
| 0 | 2 | 1 | 1 |
| 0 | 2 | 2 | 2 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 2 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 2 | 2 |
| 1 | 2 | 0 | 1 |
| 1 | 2 | 1 | 1 |
| 1 | 2 | 2 | 2 |
| 2 | 0 | 0 | 1 |
| 2 | 0 | 1 | 1 |
| 2 | 0 | 2 | 1 |
| 2 | 1 | 0 | 1 |
| 2 | 1 | 1 | 1 |
| 2 | 1 | 2 | 2 |
| 2 | 2 | 0 | 1 |
| 2 | 2 | 1 | 1 |
| 2 | 2 | 2 | 2 |

**Figure 4-16 Truth Table for Long Road**

The reasoning behind the calculated next states is:

1. The value of the cell will increase by 1 if the cell behind can pass cars into this cell.
2. The Value will decrease by 1 if this cell can pass cars onto the cell in front.
3. If this cell is empty then it cannot pass any cars forward.
4. If the cell in front if full then it cannot pass any cars forward.
5. It can always pass cars into an empty or half empty cell.

The cell behaves in a similar way to the basic road cell, and produces traffic flows that are similar to the basic road cell, and real traffic, that is as the traffic density increases, or there is an obstruction, the traffic tends to bunch up, when the bottleneck passes, the traffic tends to spread out on the road. The above rules can be applied to the cell behind to determine when it will pass cars into this cell. The long lane was implemented in logic. A Circal model of the logic is shown below, along with a circuit diagram. The Circal models were proven to be equivalent to the behavioural model (see Appendix 1 : Proof of Equivalence between Road Cells). The TimeLogic T is identical to the basic road cell time logic except that it is clocked by the slow clock, T.

Process LongLaneLogic(Bool Next1,Next0,Prev1,Prev0,CS1,CS0){

```
// Logic for a Long Lane of roadway

  static Bool
ns0,ns1,cs0,cs1,next0,next1,prev0,prev1,int1,int2,int3,int4,int5,int6,int7,
int8,int9,int10
  static Process C

   C = OR(prev0,prev1,int1) * X1BARAND(int2,int1,int3) *
   OR(int3,int4,ns0) * X1BARAND(int1,int8,int4) *
   AND(next1,int5,int2) * OR(cs0,cs1,int5) *
   AND(next1,cs0,int6) * X1BARAND(next1,cs1,int7) * AND(next1,cs1,int9) *
   OR(int10,int9,ns1) * AND(int1,int2,int10) * OR(int6,int7,int8)

       return ((C* TimeLogicT(ns0,cs0) * TimeLogicT(ns1,cs1))
[Next1/next1,Next0/next0,Prev1/prev1,Prev0/prev0,CS1/cs1,CS0/cs0] - (ns0
ns1 int1 int2 int3 int4 int5 int6 int7 int8 int9 int10))
```

```
}
```

## 4.5 A Graphical User Interface for the Space Machine

### 4.5.1 The Graphical Interface

A graphical user interface was created for the SPACE machine. In particular the interface functions as a design tool for road traffic (or any other type of CA) simulations on a reconfigurable computer. There are three reasons why such an interface is particularly important. This development was an important and necessary tool to observe the behaviour of the road traffic simulations. It permitted simulation of larger freeway models than would be practically possible using Circal alone, due to the decrease in performance of Circal as the number of states increases. It also permitted simulations to be quickly constructed, and allowed results from them to be saved.

Colour coding is used to indicate the state of each tile and the roadway was folded in a serpentine fashion across the computing surface. The results of a simulation can be saved to a file for later analysis as well as being dynamically viewed.

1. The interface permits *interactive* design and experimentation of any CA models. The interactive nature of the visualisation software is particularly valuable. This graphical user interface software makes it possible to run *interactive* experiments with the SPACE Machine. The tool makes it possible to initialise the states of cells, either before an experiment is started, or mid-way through an experiment. Tiles may also be chosen from a palette and interactively placed onto the SPACE FPGAs. It is also possible to construct cells that the user interface can modify the function of dynamically. This was used to construct cells inserted vehicles into the system according to a certain distribution. The generation of the vehicles was performed in software. The software then reconfigured the cell that was feeding the vehicles into the system, between clock cycles, to be either full or empty. Again this is an important experimental tool that aids the design of CA based simulations.
2. The interface also generalises to the visualisation of *other CA phenomena* and in this context is an extremely important development for the SPACE Machine series.

Figure 4-17 shows a screen from the experimental system running a short section of 4-lane freeway with 2 lanes of on-ramp. The coloured rectangles represent full cells whereas the white rectangles represent empty cells. The user may construct these road patterns by placing

appropriate CA cells onto the computing surface. A configuration is composed of basic building blocks including the basic road cell, slow cells, on ramp and off ramp cells. Importantly there must be a source and sink cell. Multi-lane bi-directional traffic may be modelled by appropriate design of the traffic system. . Cells will be placed on the grid to model a particular road pattern and reflect the empirically observed road conditions. As well as interactively designing the road network a user may load in pre-defined configurations.

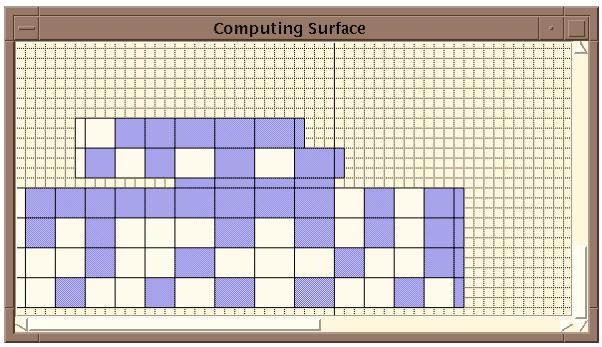



**Figure 4-17 Screen snapshot of traffic simulation showing Multi-Lane Merging Roads**

The interactive nature of the interface and the reconfigurable nature of the CA make it possible for the user to click on any of the cells and dynamically change its state. To do this the software reads the current state of the cell, reconfigures the circuit for the particular cell to force its state to the opposite state, and then reconfigures the cell back to its original functionality. The user can also drop a new cell onto the experiment, and the software reconfigures the area of the computing surface with the new cell, leaving the remainder computing surface as it was, preserving the existing configuration and cell states.

While running a simulation the results may be written to a text file for post simulation analysis.

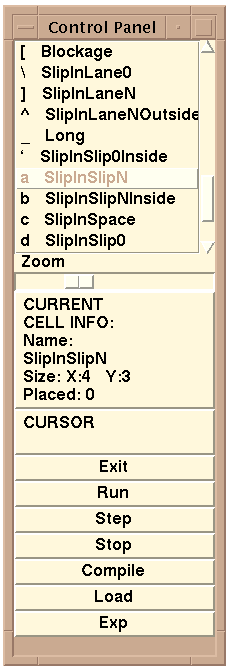



**Figure 4-18 The Controls provided by the Interactive Traffic Simulation**

Figure 4-18 illustrates some of the controls from which a user may select. Larger building blocks may also be defined such as junctions and complete sliproads.

### 4.5.2 Technical Challenges of Run Time Reconfiguration

The development of the Graphical Interface posed some interesting technical challenges. The interactive nature of the visualisation tool is achieved by reconfiguring the cells at every clock cycle to either logic 1 (denoting input a vehicle) or logic 0 (do not accept a vehicle) states. This solution allowed control of the variable parts of the experiment (input patterns) using software by reconfiguring the source cells every clock tick, while simulating the static part, the cell behaviour and road layout, in hardware.

Additionally, to maximise the use of the FPGAs the carriageways were zigzagged across the computing surface, that is the carriageway would first travel left to right across the bottom of the computing surface, then right to left in the next row up. This necessitated construction of two versions of each tile - one in which traffic flows from left to right, and a mirrored version catering for traffic flow in the opposite direction. The cells are connected at the ends using routing cells and the folding is automatically eliminated by the software before the network is presented for analysis. This method enables 1600 basic road cells to fit onto the SPACE machine. If each clock cycle represents 0.3 seconds of simulation time and set the length of each cell to be 9 metres (the distance a vehicle will travel in 0.3 seconds at 100 km/h), then this

represents approximately 14 km of single-lane roadway, or 4.8 km of 3-lane roadway. It was estimated that with SPACE 2 the simulated roadway capacity should increase to approximately 760 km of single-lane road, which is 85000 road cells updating in parallel at each clock tick (66 MHz maximum), which would be upto $20x10^6$ times real time (0.3 seconds of simulation per clock cycle).

### 4.5.3 Test Experiments

The purpose of these experiments is to discover the behaviour of various traffic cell configurations. The experiments are set up with a section of roadway, built from basic road cells, interspersed with 'test road' cells. The flow of traffic through this experimental roadway is monitored and becomes the result from the experiment. Each experimental cell configuration will be as a series of basic road cells connected together to form a length of road. The experimental cells may be followed by a complex cell, such as a slow cell, which will then be terminated by a cell that is always empty (sink). Complex cells may also be inserted between experimental cells. The experimental cells will be fed by a number of 'feed in cells', which form a input buffer so that any tail back that leaves the experimental road section does no affect the input source. The input to the feed in cells is controlled by software. This setup is shown in Figure 4-19.

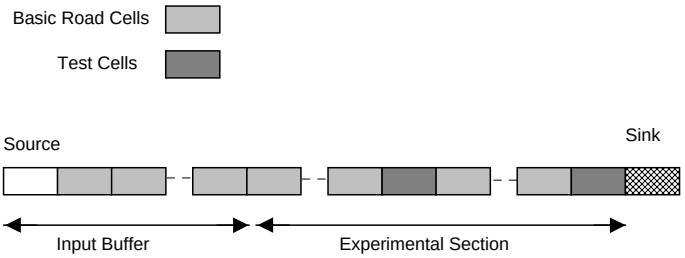



**Figure 4-19 Example Configuration of cells in a traffic simulation test**

The experimental section of the road was set to a pre-defined pattern of cars for each experiment. This is to allow simulations of various initial traffic densities. The initialisation patterns are a repeating series and are denoted by a series of ones (cell full) and zeros (cell empty), eg. {100} means a car in every third cell. The initialisation patterns will include; {1} all cells full (test outflow after a blockage), {0} all cells empty (test results of input pattern reaching the test cells), {10} alternate cells full (test the section under full flow), {100} one third full (test the section under medium flow) and {1000} one quarter full (light flow). The

Input Buffer will also be initialised to various patterns to simulate various input flow densities. The Source cell will generate the same pattern that was set into the input buffer. The input patterns shall include {0} no cars entering the road section (test the outflow from the section), {10} maximum flow into the section, {100} medium flow into the section, {1000} light flow into the section.

The results from each of the experiments are presented as rows of characters, where each row represents a time step and each character represents the state of a cell in the experiment. The first 30 characters represent the state of input buffer, a '.' represents an empty cell and an 'X' represents a full cell. The next approx. 40 characters represent the experimental road section where a ' ' represents an empty cell and a '1' represents a full cell. Active test cells are shown as ':' when empty and 'C' when full. The last approx. 9 characters are an output buffer to show the outflow from the experimental section. The output buffer in represented in the same way as the input buffer.

Figure 4-20 shows an example of this representation for two time steps.

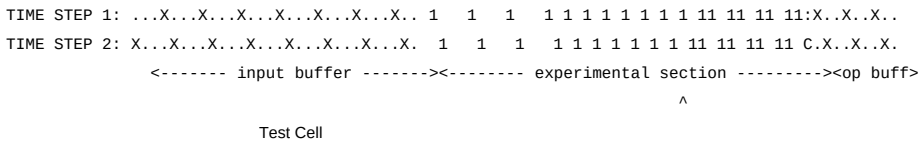



**Figure 4-20 Traffic Test time Steps**

Five experiments were performed. Experiment 1 demonstrates the inflow and outflow properties from a clear section of roadway. Experiment 2 demonstrates a lane of carriageway with a speed restriction at one end. Experiment 3 demonstrates a lane of carriageway that is blocked (e.g. a traffic light, temporary breakdown). Experiments 4 and 5 illustrate a lane of carriageway with two speed restrictions. The results may be viewed in Appendix 2: Results of Traffic Simulation Experiment

The experiments were repeated for all road cell types in various configurations and with different input and initialisation patterns. The results were consistent with empirical data as well as our knowledge and understanding of traffic flows (see Appendix 2: Results of Traffic Simulation Experiments for a summary of the findings). The results of five test experiments

were compared with reality and found to faithfully model the macroscopic behaviour of empirical data.

## 4.6 Summary of Traffic Simulation Experiments

This chapter has demonstrated the applicability of reconfigurable computing architectures such as SPACE to the problem of rapidly simulating highly complex chaotic systems. A family of non-homogeneous cellular automata traffic models were designed and realised on the SPACE FPGAs. The implementation was modelled and formally verified using a mature hardware verification system, the Circal System.

The significance of simple microscopic modelling rules achieving the macroscopic behaviour is important, together with the real time potentially large scale modelling of this complex system.

# 5. Cryptography with Cellular Automata on the SPACE Machine

## 5.1 Chapter Overview

A CA Cryptography system is used to further explore methods for modelling and implementing complex, and useful, cellular automata systems using Circal and the SPACE machine. This application is used to particularly explore the area of dynamically reconfiguring the circuits depending upon the inputs to the algorithm.

The chapter begins with an overview of cryptograph, including an explanation of the different types of cryptography. It then goes on to explain the connection between cryptography, complex systems, and CA. A cryptography algorithm, based on CA that is used in this chapter is introduced and explained. The general approach to implementing the algorithm is described, and this is followed by a discussion of the implementation in Circal and on the SPACE machine. Finally the results from the chapter are discussed and summarized.

## 5.2 Introduction to Cryptography

Cryptography is the art, or science, of keeping messages secret. The message is called plain text or clear text. Encoding the contents of the message in such a way that hides its contents from outsiders is called encryption, which produces the coded cipher text. The process of retrieving the plain text from the cipher text is called decryption. A system that performs encryption and decryption is called a cryptosystem. Encryption and decryption usually make use of a key. An algorithmic method of encryption and decryption is called a cipher. A provably unbreakable, practical cipher is not yet known [14] although many different encryption / decryption algorithms have been suggested. The effective measure of the quality of a cryptosystem remains in practice what it has always been; the longer a cryptosystem is in wide use without being broken, the better it is.

Some ciphers rely on the secrecy of the algorithms; such algorithms are only of historical interest and are not adequate for real-world needs. All modern algorithms use a key to control encryption and decryption; a message can be decrypted only if the key matches the encryption key. The key used for decryption can be different from the encryption key, or they may be the same. Up until the mid 1970’s, cryptography was an arcane science practised largely by government and military security experts. That situation changed dramatically following the

development of public key cryptography by Hellman and Diffie [39] in 1975. This development solved a major problem with most cryptographic systems - that of exchanging keys.

With increasing concerns with privacy and security, as will as ever increasing speeds in computer communications high speed secure cryptography is becoming more important.

### 5.2.1 Basic Cryptographic Algorithms

There are two classes of key-based algorithms, symmetric (or secret-key) and asymmetric (or public-key) algorithms. The difference is that symmetric algorithms use the same key for both encryption and decryption (or the decryption key is easily derived from the encryption key), whereas asymmetric algorithms use a different key for encryption and decryption, and the decryption key cannot be derived from the encryption key. Symmetric algorithms can be divided into stream ciphers and block ciphers. Stream ciphers can encrypt a single bit of text at a time, whereas block ciphers take a number of bits (typically 64 bits in modern ciphers), and encrypt them as a single unit. Asymmetric ciphers (also called public-key algorithms or generally public-key cryptography) permit one of the keys to be made public. Public-key cryptography can be used by either encrypting with the public or private key. A message encrypted using the public key can be sent to the owner of the private key, preventing anyone who intercepts the message from decrypting it. The private key can also be used to encrypt a message, proving to the recipient that the message has actually come from the owner of the key. Some cryptographic algorithms are designed to be executed by computers or specialized hardware devices. In most applications, cryptography is done in computer software. Generally, symmetric algorithms are much faster to execute than asymmetric ones. In practice, they are often used together, so that a public-key algorithm is used to encrypt a randomly generated encryption key, and the random key is used to encrypt the actual message using a symmetric algorithm.

A well-known cryptosystem is DES (Data Encryption Standard), developed in the 1970s, and made a standard by the US government [10]. DES is an example of an iterated symmetric cryptosystem where transformations are applied repeatedly to a message. The DES algorithm consists of 16 rounds of a transformation designed to fully mix message information together using the key information. The security of the DES has recently been seriously challenged especially when used with special-purpose hardware. A variant of DES, Triple-DES or 3DES is based on using DES three times (normally in an encrypt-decrypt-encrypt sequence with three different, unrelated keys). Many people consider Triple-DES to be much safer than plain DES.

RSA (Rivest-Shamir-Adelman) [10] is probably the best known asymmetric public key algorithm. It is generally considered to be secure when sufficiently long keys are used (512 bits is considered insecure, 768 bits is considered moderately secure, and 1024 bits is considered good). The security of RSA relies on the difficulty of factoring large integers. Dramatic advances in factoring large integers would make RSA vulnerable. Fast implementations of RSA on programmable hardware have been made by Shand et al. [85].

### 5.2.2 Cryptography and Complex Dynamic Systems

Gutowitz [29] linked complex chaotic systems with cryptology, explaining that as a complex system progresses in time it its initial state is “forgotten”, that is the new state, after enough time has passed, has no resemblance to the initial state. The future state of a (chaotic) dynamic system does however depend on its initial state and the dynamic system proceeds deterministically from the initial state to the future state. Encryption and decryption may be achieved by making the key of the cryptosystem the initial state of a complex system. The message is encrypted by continually combining it with an information stream generated by forward iteration of the system. The message could then be decrypted by combining the cipher text with an information stream generated by forward iteration of the system. This approach is promising in terms of security since such a complex dynamic system is unpredictable without access to the initial state. The evolution of the system over time bears little evident connection with the equations that define them, and anyone who does not know secret initial condition would not be able to recreate the development of the complex system.

#### 5.2.2.1 Reversible Dynamic Systems

Reversible dynamic systems may be run forward and backward. A message can be encrypted by encoding it as a state of the system. The system is run forward in time to produce the cipher text. To decrypt the cipher text, the system is inverse iterated the same number of time steps. The key for such a system is the equations that define the dynamic system itself. The ‘key’ operates directly on the message to encrypt and decrypt it. This contrasts with systems that only forward iterate, where the key is the initial condition of a fixed system, and the equations that define the dynamic system are fixed. One system that uses reversible dynamic systems for cryptography is that by Guan [28], which is a public key cryptography system. The public key for this system is a dynamical system that is the inverse to the private key. Guan uses a non-homogeneous CA in which the rule that updates a site's value depends on the site. Guan's system essentially relies on the difficulty of inverting a complicated system of polynomial

equations. Kari also used reversible dynamic system [42] but utilised translationally invariant, cellular automata. Kari bases much of his reasoning on a result of his, [41] which shows that even deciding whether a given cellular automaton, with more than one dimension, has an inverse is impossible.

#### 5.2.2.2 Irreversible Dynamic Systems

Habutsu et. al. [30] uses both forward and inverse iterations of a dynamical system but there is a very significant difference in that the dynamical system used is irreversible. An irreversible system is one in which some states have none, or more than one, antecedent state. Habutsu et al. uses irreversible dynamic systems to encrypt by i) encoding a message as a state of the system, and ii) running the system backward in time by choosing one of the two pre-images of this state at random. This process is repeated many times, resulting in a cipher text. A receiver of the cipher text who knows the secret key can decrypt the message by running the system forward in time the same number of iterations as was used in encryption.

## 5.3 A Cellular Automata Model for Encryption

### 5.3.1 Background

The following section explores a CA cryptography system proposed by Gutowitz [29]. Gutowitz linked cryptography with CA systems since a CA system may generate complexity. He states [29] "*Cryptography presents the challenge of finding irreversible rules under which all blocks have prior states that may be constructed rapidly*". The algorithm uses the inverse integrations of an irreversible CA for encryption, and the forward iterations for decryption.

### 5.3.2 The Encryption Algorithm with CA

The CA rules in the algorithm have, what Gutowitz describes as a 'toggle' property. The 'toggle' rules can be either 'right toggle', or 'left toggle' or both. This means that, for a one dimensional CA, changing the value of the extreme (left of right) neighbourhood site always changes the result of the function state at the next iteration. For example, the following rule is left toggle, as, changing the state of the left most bit (extreme left neighbour) will change the state of the cell at the next iteration (S(t+1)).

| Transition Rules for a 1-dimensional Right Toggle CA | | | |
|---|---|---|---|
| Neighbourhood & State S(t) | Cell State S(t+1) | Neighbourhood & State S(t) | Cell State S(t+1) |
| + - - | **+** | - - - | **-** |
| + - + | **-** | - - + | **+** |

| + + - | - | - + - | + |
|---|---|---|---|
| + + + | - | - + + | + |

**Figure 5-1 Example Right Toggle CA**

Toggle rules are important as they allow a pre-image (a state of the CA which would lead to the current state at the next step) for the current state to be rapidly constructed, as is explained below.

The encryption algorithm takes the plain text as the initial state of the 1-dimensional CA. It then finds a pre image, which is a CA state whose next state is this state. To find a pre image each cell is updated in sequence, starting at the right of the CA (assuming a left toggle rule is used). Firstly additional CA cells are added to the right of the CA (assuming a left toggle rule is used). The number of CA cells that are added is 2 times the CA rules radius. These cells can be set to a random state. For example, for the rule shown in the table above, which has a radius of 1 (one neighbour on either side), two cells would be added. Next, the state of the cell next to the added cells (e.g. the right most cell of the plain text for a left toggle rule) can be updated by applying the CA rule to the state of this cell with the added cells. This is then repeated; working left one cell at a time, until all the cells have had the rule applied to them.

This process is then repeated a number of times, each time adding random bits to the right (for a left toggle rule) of the message. This is shown, in the table below, for a single iteration, using a left toggle rule on a 4 bit plain text message. The function f(a,b,c) is the left toggle rule.

| **Plain Text**, $S^{t=0}$ | $\mathbf{X_0}$ | $\mathbf{X_1}$ | $\mathbf{X_2}$ | $\mathbf{X_3}$ | $X_4$ | $X_5$ |
|---|---|---|---|---|---|---|
| $S^{t=1}{}_i$ | $f(X^0{}_0, X^1{}_1, X^1{}_2)$ | $f(X^0{}_1, X^1{}_2, X^1{}_3)$ | $f(X^0{}_2, X^1{}_3, X^1{}_4)$ | $f(X^0{}_3, X^1{}_4, X^1{}_5)$ | Random | Random |

**Figure 5-2 Pre Image Generation**

Note that while the encryption is applied to one bit at a time this can perform in parallel over the iterations (see 5.3.3.2).

To decrypt the message the CA rule is repeatedly applied the correct number of times to the cipher text. An example of the encryption and decryption sequences is shown in the table below using the CA rule described in Figure 5-1. The top and bottom shaded rows are the plain text; the shaded middle row is the cipher text. The iterations of the

decryption have been shifted to the left one position to maintain the alignment with the encryption algorithm, e.g. the left most bit if from forward iterating the second bit from the right.

| Encrypt | Added Cells |
|---|---|
| 1 1 1 1 1 0 0 0 0 0 1 0 0 1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 | |
| 0 0 1 0 0 1 1 1 1 1 1 0 0 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 | 0 0 |
| 1 1 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0 0 1 0 0 1 0 0 1 | 1 1 0 1 |
| 1 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 0 0 | 0 1 0 0 0 1 |
| 0 1 1 1 0 0 1 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 1 1 1 | 1 0 1 1 1 0 1 0 |
| 1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 0 1 0 | 0 1 1 0 0 1 0 1 1 0 |
| Decrypt | |
| 0 1 1 1 0 0 1 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 1 1 1 | 1 0 1 1 1 0 1 0 1 0 |
| 1 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 0 0 | 0 1 0 0 0 1 0 1 1 0 |
| 1 1 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0 0 1 0 0 1 0 0 1 | 1 1 0 1 1 0 1 0 1 0 |
| 0 0 1 0 0 1 1 1 1 1 1 0 0 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 | 0 0 1 0 0 1 0 1 1 0 |
| 1 1 1 1 1 0 0 0 0 0 1 0 0 1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 | 1 1 1 1 1 0 1 0 1 0 |

**Figure 5-3 Example Encryption and Decryption**

As stated, the key for the system is the CA rule. The rule does not have to be the same at each iteration; in fact the rule can be different at each cell with the restriction that all cells in an iteration are either all right toggle rule or all left toggle rules. This allows a key to be used which selects a different rule for each cell and/or iteration. The key can also be used to alter the links between iterations (i.e. the order of the bits can be changed between iterations). This adds to the confusion of the data and ensures the effect of each bit in the plain text effects many other sites, which are small changes to the input (e.g. 1 bit difference) quickly change the state of many cells.

### 5.3.3 Pipelining the Algorithms

#### 5.3.3.1 Decryption

The use of one-dimensional CAs for cryptography has the advantage that it is possible to ‘pipeline’ the computations of many iterations of the CA. For the decryption case, the rules for each iteration are implemented on the computing surface. The output states ($S^{t+1}$) from the first iteration cells are connected to the inputs ($S^t$) of the next iteration and so on. If we define the number bits that are processed by each iteration as a “data set” then at the first clock tick the first iteration processes the first input data set. At the second clock tick the second iteration continues to processes the first input data set, while at the same time the first iteration is processing the second set of input data.

These results the throughput of the system being one data set processed at each iteration. In fact it is possible to increase the throughput to any level that is required by implementing several of these machines to run in parallel.

#### 5.3.3.2 Encryption

Pipelining the encryption is slightly more complicated. To demonstrate how to pipeline the encryption algorithm we will examine encrypting 4 bits of data over 4 iterations. We may consider that the inputs to the system is the data to be encrypted and the random data that is required at each iteration. Using the same format as Figure 5-3, this is shown in Figure 5-4 as cells with a "0" in (that is data that is available before iteration 1). Looking at Figure 5-4 it can be seen that given only the input data it is possible to calculate the new state the cells with a "1" in them. That is those cell states can be calculated at the first "clock tick", of the encryption. Again, given the cells that were calculated at the first iteration it is then possible to calculate the state of the cells with a 2 in them. This is then repeated until all the cell states for all encryption iterations have been calculated.

| Data | | | | Added Cells | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | | | | | | | |
| 4 | 3 | 2 | 1 | 0 | 0 | | | | | | |
| 6 | 5 | 4 | 3 | 2 | 1 | 0 | 0 | | | | |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 0 | | |
| 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 0 |

**Figure 5-4 Example of Order of Encrypting Cells**

While, for this example, it requires 10 clock ticks to fully encrypt the data through 4 encryption iterations, it is possible to start encrypting a new "data set" at each clock tick. To achieve this the cells can be laid out so that all cells that can be calculated at a certain clock tick are on the same row. Cells that cannot be calculated at a certain clock tick have a cell which just takes the value of the cell above and stores its value (e.g. a time process described in section 3.3.2). This is shown, for the example in Figure 5-4, in Figure 5-5. The cells with the numbers are the cells from the example above, in the same column but shifted to the correct row, the cells with the horizontal bars would contain a time process which moves the state from the cell above.

| Data | | | | Added Cells | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| \| | \| | \| | 1 | \| | 1 | \| | 1 | \| | 1 | \| | \| |
| \| | \| | 2 | \| | 2 | \| | 2 | \| | 2 | \| | \| | \| |
| \| | 3 | \| | 3 | \| | 3 | \| | 3 | \| | \| | \| | \| |

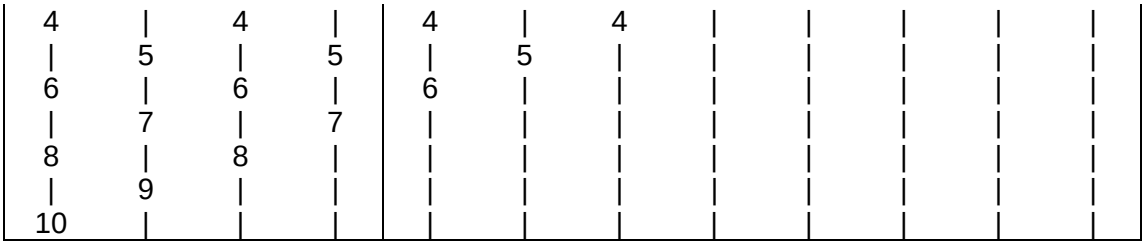



**Figure 5-5 Pipelined Encryption Example**

The use of pipelined one-dimensional CAs conveniently has the inputs and outputs naturally located at the edges of the implementation, which allows for simple I/O.

### 5.3.4 Design and Implementation

The crypto system was implemented on the SPACE 2 platform. The most efficient method to put a CA rule onto the machine is by implementing the update rule for every cell in logic, thus avoiding lookup tables. As stated above the key to the encryption is the update rules at each cell. To allow the key to be changed a skeleton circuit was generated. This circuit could then be modified before being placed on the machine. This also allowed a different approach to implementing CA on the machine to be investigated, that is reconfiguring the CA rules automatically before placing the circuit on the machine, giving the flexibility of a different custom configure circuit each time the key is changed. The skeleton for the system consisted of the basic algorithm described above being implemented for each iteration of the decryption and encryption algorithms, with a generic CA function, with a radius of one. The links between iterations could be modified but were limited to groups of 4 neighbouring cells to simplify the implementation.

The behaviour of the system was again modelled in Circal, and the logic implementation verified using the Circal system. The pipelined encryption and decryption behaviour was modelled by firstly constructing a behavioural model of a toggle rule. The encryption behaviour model was constructed by putting a toggle rule in each cell where a rule was required, and a time cell in the other positions. The Circal model that was constructed allowed any number of bits to be encrypted for any number of iterations. The Circal model for a toggle rule and the encryption is shown below. As can be seen, the first, middle and last sections of the encryption model are basically the same, the primary difference being that the first and last sections connect the first and last rows of the model to the plain and encrypted data, whereas the middle section only connects to internal states.

A Circal process for example toggle rule (the rule shown above):

```
Process LeftToggleRule1 (Bool Left, Middle, Right, Currentstate) {

  static Bool nextstate,currentstate,left,middle,right
  static Process C

  C <- (left.0 middle.0 right.0 nextstate.0) C +
       (left.0 middle.0 right.1 nextstate.1) C +
       (left.0 middle.1 right.0 nextstate.1) C +
       (left.0 middle.1 right.1 nextstate.1) C +
       (left.1 middle.0 right.0 nextstate.1) C +
       (left.1 middle.0 right.1 nextstate.0) C +
       (left.1 middle.1 right.0 nextstate.0) C +
       (left.1 middle.1 right.1 nextstate.0) C

  return ((C * Time(nextstate,currentstate))[Left/left, Middle/middle,
                 Right/right, Currentstate/currentstate] - nextstate)
```

The encryption process:

```
Process Encrypt (Bool plain[], crypt[], int width, iterations) {

  Process encrypt
  int r, c
  int rows=width-3 // internal rows (excludeing input & output)
  int cols=width
  Bool encStates[rows*cols]  // internal states, model a 2d array in 1d
array

  // connect up the inputs to the first internal row
  for (c=0; c<cols; c++) // process all columns
    if ((c%2) != 0 && c <= (cols-3) && c >= (cols-(2*iterations)-1)) {
      // these are the cells that have the rule in them
      encrypt = encrypt * LeftToggleRule1(plain[c], plain[c+1], plain[c+2],
                        encStates[c])
    } else {
      // these are the stright through cells
      if (c==0) { // first process, cannot compose
        encrypt = Time(plain[c], encStates[c])
      } else {
        encrypt = encrypt * Time(plain[c], encStates[c])
      }
```

```
    }

  // compose middle cells together
  for (r=1; r<rows; r++) // process all rows except input
    for (c=0; c<cols; c++) // process all columns
      if (((r+c)%2) != 0 && c < (cols-2-r) && c >= (cols-(2*iterations)-r-
1)) {
        // these are the cells that have the rule in them
        encrypt = encrypt * LeftToggleRule1(encStates[((r-1)*cols)+c],
                  encStates[((r-1)*cols)+c+1], encStates[((r-1)*cols)+c+2],
                  encStates[(r*cols)+c])
      } else {
        // these are the stright through cells
          encrypt = encrypt * Time(encStates[((r-1)*cols)+c],
                       encStates[(r*cols)+c])
      }

  // connect up the outputs to the last internal row
  // note there is only a rull cell in the left most column
  encrypt = encrypt * LeftToggleRule1(encStates[(rows-1)*cols],
                encStates[((rows-1)*cols)+1],encStates[((rows-1)*cols)+2],
                crypt[0])

  // the rest are the stright through time cells
  for (c=1; c<cols; c++) // process rest of columns
    encrypt = encrypt * Time(encStates[((rows-1)*cols)+c], crypt[c])

  return encrypt
}
```

The decryption model is simply a number of single dimension CAs connected together to form a two dimensional CA. The Circal model for decryption is shown below, note that once again the model is in three similar parts to allow the inputs (encrypted data) and outputs (decrypted data) to be connected.

The decryption process

```
Process Decrypt (Bool crypt[], decrypted[], int width, iterations) {

  Process decrypt
  int r, c
  int rows=iterations-1 // internal rows, last iteration is decrypted[]
  int cols=width
```

```
    Bool decStates[rows*cols] // internal states model a 2d array in 1d array

    // connect up the inputs
    decrypt = LeftToggleRule1(crypt[0], crypt[1], crypt[2], decStates[0])
    for (c=1; c<cols-2; c++)
      decrypt = decrypt * LeftToggleRule1(crypt[c], crypt[c+1], crypt[c+2],
                decStates[c])

    for (c=cols-2; c<cols; c++)
       decrypt = decrypt * Time(crypt[c], decStates[c])

    // compose middle cells together
    for (r=1; r<rows; r++) { // process all rows except input
      for (c=0; c<cols-2; c++) // process all columns
        decrypt = decrypt * LeftToggleRule1(decStates[((r-1)*cols)+c],
                                            decStates[((r-1)*cols)+c+1],
                                            decStates[((r-1)*cols)+c+2],
                                            decStates[(r*cols)+c])
      for (c=cols-2; c<cols; c++)
        decrypt = decrypt * Time(decStates[((r-1)*cols)+c],
                                 decStates[(r*cols)+c])
    }

    // and connect up the output
    for (c=0; c<cols-2; c++) // process rest of columns
      decrypt = decrypt * LeftToggleRule1(decStates[((rows-1)*cols)+c],
                                          decStates[((rows-1)*cols)+c+1],
                                          decStates[((rows-1)*cols)+c+2],
                                          decrypted[c])

    for (c=cols-2; c<cols; c++)
       decrypt = decrypt * Time(decStates[((rows-1)*cols)+c], decrypted[c])

    return decrypt
  }
```

The Circal models were used to show that the system could encrypt and decrypt data. This was done by composing the encrypt and decrypt processes together and then clock some test data through the system, this is shown below for a small system. The trace statement outputs the states of the system at each time step allowing the encryption and decryption to be verified.

```
  int width = 6
  int iteration = 2
```

```
Bool plain[width]
Bool crypt[width]
Bool decrypted[width]

Process enc = Encrypt(plain, crypt, width, iteration)
Process dec = Decrypt(crypt, decrypted, width, iteration)

Process TS
  TS <- (t.1 plain[0].1 plain[1].1 plain[2].0 plain[3].1 plain[4].1
          plain[5].1) //
        (t.1 plain[0].0 plain[1].0 plain[2].0 plain[3].0 plain[4].0
          plain[5].0) //
        (t.1 plain[0].0 plain[1].1 plain[2].1 plain[3].1 plain[4].1
          plain[5].0) //
        (t.1 plain[0].1 plain[1].0 plain[2].1 plain[3].0 plain[4].1
          plain[5].1) //
        (t.1 plain[0].1 plain[1].1 plain[2].0 plain[3].1 plain[4].1
          plain[5].1) //
        (t.1 plain[0].0 plain[1].0 plain[2].0 plain[3].1 plain[4].1
          plain[5].1) //
        (t.1 plain[0].1 plain[1].1 plain[2].0 plain[3].1 plain[4].1
          plain[5].1) //
     /\end

print "\nTest 1\n"
trace ~(TS * enc * dec * CLOCK)
```

The system was implemented by generating a “skeleton” circuit manually for the SPACE 2 machine. The skeleton circuit was a complete encryption and decryption with the same rule in each cell (as described above). The circuits for the skeleton cryptosystem are in ‘Appendix 3: Circuit Diagrams of the Cryptography System’. The key for the crypto system is the toggle rule at each site, plus the interconnections between iterations. The key for decryption is the same as the key for encryption, where the cells are changed in the first iteration of encryption they need to be changed in the last iteration of decryption. To implement the key a program, running on the host, chooses which cells need to be modified to what toggle rule (based on the key). It then modifies a copy the circuit list (a textural representation of the circuit that can be compiled ready for placing on the SPACE machine) of the skeleton circuit, changing the toggle rule circuits to the new toggle rule circuits. Following this the key is used to modify the interconnections between the iterations. This is again achieved by the program modifying the

circuit list. Once the circuit list has been modified for the particular key it is routed (compiled) and downloaded onto the SPACE machine. For the experiments both the encryption and decryption circuits were downloaded to the SPACE machine. The plain text was then written to the input of the encryption circuit on the board and the cipher text read from its output. The cipher text could then be written to the decryption circuit to be decrypted. The system was tested by connecting the output from the encryption section to the input of the decryption section on the SPACE machine. Data was then written to the input of the encryption section. Both the encrypted and decrypted text was read from the outputs of the encryption and decryption sections respectively.

## 5.4 Results

The cryptographic system as implemented encrypted and decrypted a byte of information at each clock cycle. The implementation could be scaled to encrypt any number of bits per clock cycle, assuming the hardware was available. This would allow very fast encryption and decryption of data to be implemented.

The need to route and layout the circuit before running the system did take a significant time. This was mainly due to the proprietary routing software that was used did not provide a modifiable final circuit description, which could have been modified directly without the need to re-route the circuit. This setup time could have been significantly reduced by writing a tool to manually place circuits onto the machine that could be simply modified by software before downloading to the machine. (Note that this is how the circuits were placed onto the original SPACE machine). While this is technically possible it was not considered that effort involved in developing such a tool would significantly add to the knowledge gained through these experiments. An alternative approach to having to having to route and layout the circuit each time before running the system would be to develop the system in a high level design language such as Pebble [45] that allows parameterised inputs. It is then possible to generate customized layouts at run-time much faster than having to re-layout the entire design again [15].

The algorithm chosen allowed the use of non- homogeneous CAs, (with a different update rule at each site), as well as homogeneous CAs. This allowed the confusion of the system to be increased for a given neighbourhood and number of iterations.

## 5.5 Summary

This chapter has demonstrated the applicability of reconfigurable computing architectures to the real life problem of efficiently encrypting and decrypting data. The concept of generating unique logic circuits for each instantiation of the program was explored and implemented on the SPACE 2 hardware. The concept of changing the encryption rules in hardware before each execution of the program could be used to increase the security of the system as it could be made very difficult to inspect the system as it was running.

# 6. Future Directions

## 6.1 Chapter Overview

This thesis has developed a number of techniques for developing CA models using Circal as a modelling language and the SPACE machine as a platform to develop these models on. It has discussed the need for high performance simulations and has shown that using cellular automata to model complex systems and using the inherent concurrency of digital electronics of a reconfigurable computer to implement these models, high performance computing solutions can be produced.

This chapter looks ahead at areas of research that could be explored to further provide modelling solutions for constructing CA models using reconfigurable computing platforms. Possible future areas of research for both traffic modelling and cryptography are discussed. The chapter finishes with the final conclusions drawn from the thesis.

## 6.2 Future directions of the CA paradigm on the SPACE Machine

### 6.2.1 Comparative Homogeneous and Non-homogeneous Modelling

The implementation of CA models on the SPACE Machine is significant since the naturally concurrent architecture of this reconfigurable computer can be directly mapped to the concurrent, distributed, local, regular, nature of CA. While other hardware does exist for CAs, the potential of being able to configure different areas of the computing surface for different processes is exciting. One of the things it enables is to make possible the modelling of non-homogeneous CA, where different types of cell may be placed on the computing surface. The non-homogeneous CA can potentially be used for modelling complex physical systems, taking into account the actual non-homogeneous composition of the system (e.g. interactions of different types of fluid molecules or different areas of a physical space where particles being modelled change their behaviour because of their location in the physical space). Future research following on from that reported in this thesis would be in comparative modelling of homogeneous and non-homogeneous CAs, utilising the reconfigurable nature of a computing surface to make both a theoretical and empirical investigation of the most appropriate modelling techniques for complex systems, based on the homogeneity and non-homogeneity of the system.

### 6.2.2 Run Time Reconfigurability

Another possible future direction of this thesis also stems from exploring the reconfigurable nature of the SPACE Machine. One possible area of research is that of allowing CA cells of non-homogeneous CA to move on the CA lattice at run time. This would allow the modelling of various real life systems, where the item being modelled by the cell actually moves in the system, e.g. the interaction of different gas particles as they mix.

The SPACE Machine also makes possible: i) the interactive development of CA models and ii) run time modification of the CA model. It would be valuable to explore and exploit these two avenues of reconfigurability enabled by the SPACE Machine.

Firstly, the interactive development of CA models becomes particularly important in the non-homogeneous CA where there may be the need either: i) to experiment with different configurations of the automata to achieve the optimal design or ii) to layout a particular known 'starting configuration' of a non-homogeneous system. The interactive interface developed in this thesis supports just such dynamic experimental modelling with visual feedback and the possibility of changing the cells of the automaton. The road traffic simulations enabled such modelling and other non-homogeneous physical systems may also be modelled in this fashion.

Secondly, the possibility of modifying the CA during run time is another avenue of reconfigurability that may be explored in both homogeneous and non-homogeneous CA architectures. It would be possible to model systems whose evolutionary nature actually *changed* the rules of the automaton. That is, after a certain number of iterations, or after a certain degree of complexity had been reached in the configuration of the system, the rules themselves would alter. For example, this might be illustrated in the Game of Life by changing the rule for survival according to the number of living or dead cells on the board. This is valuable since there is potential for empirically modelling complex systems in a truly dynamic fashion - where the rules evolve over time to account for the behaviour of the system.

### 6.2.3 Multi-Dimensional Modelling

This thesis also raises the interesting question of how multi-dimensional CA modelling might be achieved. The potential of establishing a 3-dimensional assembly of reconfigurable devices exists in order to make a very direct mapping of 3-dimensional physical systems onto a 3-dimensional architecture. Many complex modelling problems exist in 3-dimensional space (e.g.

weather systems) and it would be interesting to explore whether a series of 3-dimensional CA rules, on a 3-dimensional architecture, would provide any benefit in modelling the complex systems.

An alternative path towards 3-dimensional modelling would be to investigate how such automaton rules might be simulated on the 2-dimensional surface of the SPACE Machine. The simulation approach (of using 2-dimensional boards) would theoretically also enable higher multi-dimensional modelling to be investigated.

## 6.3 Future Directions for Road Traffic Modelling

### 6.3.1 Microscopic and Macroscopic Modelling

This thesis has explored the microscopic traffic-modelling problem where the system being modelled had very fine granularity (i.e. 5 metres of road). Various detailed traffic networks describing the structure of freeway systems have been simulated and shown to conform to reality, that is, to the behaviour of the physical system in question. The microscopic modelling approximated very well the macroscopic behaviour of real road systems and related research has also demonstrated the reality of the CA models with actual traffic networks [81].

However, limitations to the physical size of the traffic network under investigation have been encountered. This is due to the scale of the SPACE Machine. There is potential to investigate a coarser grained model. One such trail consisted of 10 vehicles per 100 metres of road and this became the basic road cell. This increased the size of the physical system that could be modelled although further examination is required to determine whether such a coarse-grain model is acceptable and affect its physical realism. Another solution could be to “swap” sections of a large model onto and off of the computing surface. Each section could be run for a number of clock cycles. The outputs from one section could the be used as inputs to another section.

### 6.3.2 Hardware Approaches to Improving Simulation Speed

Simulation speed is an important aspect to modelling traffic situations, which is, achieving rapid response to different scenarios for on-line traffic management. To this extent, it may be advantageous to further explore reconfigurable computing solutions to improving the performance of CA modelling. One question for future research is what is the most appropriate

reconfigurable hardware for such complex physical system simulations, given that speed is crucial.

One particularly promising technology is dynamically-programmable gate arrays, or DPGAs [13], which are gate arrays containing multiple configuration contexts that can be switched between at the frequency of the system clock. An FPGA can be thought of as a special case DPGA, having a single context. Systems that are too large to simulate in their entirety on our physical gate array can be partitioned into adjoining sub blocks, each of which is placed in its own DPGA context. Sub blocks are evaluated in turn on the hardware, with the boundary state of each context communicated to its neighbours via I/O registers. The substantial performance advantage of gate array simulations over software approaches lessens the impact of the speed reduction introduced by evaluating a system of n sub blocks in n clock ticks.

The experiments performed by this thesis suggest that fine-grained FPGA architecture gives a high array utilisation in CA applications. CAs lend themselves well to localised routing, which suggests that the hierarchical routing available in some FPGAs can be dispensed with in a gate array designed specifically for CA simulation. Any saving in routing resources translates to significant density improvements, given that routing tends to dominate the silicon budget of FPGA chips [12].

## 6.4 Future Directions for Cryptography

### 6.4.1 Developing CA Solutions for Public Key Encryption

This thesis demonstrated just one CA solution to encryption and decryption. There is potential to explore alternative CA paradigms to establish secure encryption schemes. The area of public key encryption has not been explored here using CAs.

There is the potential of theoretically and practically exploring reversible and irreversible CAs for cryptography. There is also the possibility of exploiting the dynamically reconfiguration (see section 6.2.2) of the CA cell rules, possibly during the running of the system, in order to achieve the irreversible system.

With the introduction of new hybrid devices being developed with micro-controllers and FPGA technology on the same chip the possibilities for reconfigurable systems-on-chip [2] there is the possibility of incorporating embedded cryptography directly into such systems.

### 6.4.2 Theoretical Investigation of Cipher Security

There is also potential to develop theoretical solutions to the question of how secure is a particular cipher. The CA on the border of chaotic (see section 2.2.5) behaviour may be particularly interesting to explore since these lead to very unpredictable systems and may be more secure than the CA models producing more regular patterns. There may be the potential of giving some guarantee of cipher security based on the complex behaviour that will be demonstrated by a particular CA model, and of evaluating current cipher algorithms to determine their degree of soundness.

### 6.4.3 Increased Efficiency in Code Breaking

From the code breaking angle there is potential to exploit the efficiency of hardware to discover the keys to encrypted messages and so use the CA as a code breaker.

### 6.4.4 Run Time Reconfiguration

It is possible for algorithms running on reconfigurable computing platforms to reconfigure themselves as they run. This feature could be exploited in CA based cryptography. It is possible that the CA updating rules at all, or some, cells could be changed by the algorithm as it executes.

## 6.5 Conclusion

This thesis concludes that a paradigm for reconfigurable CA modelling complex systems on the SPACE machine has been shown to be of practical value in both road traffic modelling and cryptography, as well as of theoretical interest in terms of modelling CA as processes, how these might be practically engineered onto a reconfigurable computing surface and the potential for using the reconfigurability for non-homogeneous modelling.

The SPACE Machine series of reconfigurable computers is currently unique in its reconfigurable architecture which lends itself to CA simulation. However, there is potential of transferring the reconfigurable, non-homogeneous CA modelling onto other reconfigurable platforms. The principles established in this thesis are not hardware specific although the speed achieved by the architecture of the SPACE Machine series is not anticipated on other architectures due to the spatial configuration.

Finally, directions of future research have been identified in both theoretical and practical terms. These areas will utilise and build upon the contributions of this thesis and so support and demonstrate the importance and significant scope of the work completed.

# References


[1] A. Bailey, G McCaskill., and G. Milne “An exercise in the automatic verification of asynchronous designs,” *Formal Methods in System Design*, vol 4, no 3, pp.213-242, May 1994.

[2] N. W. Bergmann, "Enabling technologies for reconfigurable system-on-chip" *IEEE International Conference of Field-Programmable Technology, 2002* Proceedings. 2002

[3] J. A. Bergstra, J. W. Klop “Algebraic Specifications for Parametrized Data Types with Minimal Parameter and Target Algebras,” *ICALP 1982, Proceedings.* Lecture Notes in Computer Science, Vol. 140, Springer, 1982.

[4] D. Buell, J. Arnold, W. Kleinfelder editors, “Splash 2: FPGAs in a Custom Computing Machine,” IEEE Computer Society Press, May 1996.

[5] A. W. Burks, “Essays on Cellular Automata. Urbana,” University of Illinois Press, 1970.

[6] Algotronix, Ltd “The CAL1024 Data sheet,” Doc. AX501-005, Algotronix, Ltd, Edinburgh, Scotland, 1992.

[7] A. Cerone, A.J. Cowie, G.J. Milne, P.A. Moseley, “Modelling a Time Dependent Protocol using the Circal Process Algebra,” University of South Australia, Tec Rep TR CIS-96-010, Oct 1996.

[8] P. Cockshott, P. Shaw, P. Barrie, G. Milne “Scalable cellular array architecture,” *IEE Computing & Control Engineering Journal*, vol 3, no5, September 1992.

[9] P. Cockshott, G. McCaskill, and P. Barrie “Use of a high speed cellular automata machine to simulate road traffic,” University of Strathclyde Research Report HDV-27-93, May 1993.

[10] Cryptographic Algorithms [Online]. Available: http://www.ssh.fi/tech/crypto/algorithms.html.

[11] Dawood, A.; Bergmann, N. "Enabling technologies for the use of reconfigurable computing in space" *Proceedings of the Fifth International Symposium on Signal Processing and Its Applications*, Vol2, pp683-687, 1999

[12] A. DeHon “Reconfigurable architectures for general-purpose computing,” Massachusetts Institute of Technology, *PhD Thesis* (A.I. Technical Report No 1586), October 1996.

[13] A. DeHon. “DPGA-coupled microprocessor: commodity ICs for the early 21st century,” Proc. *2nd IEEE Workshop on FPGAs for Custom Computing Machines* pp. 31-39, April 1994.

[14] E. Denning, “Cryptography and Data Security,” Addison-Wesley, 1982.

[15] A. Derbyshire, W. Luk "Compiling run-time parametrisable designs" *IEEE International Conference on Field-Programmable Technology, 2002 Proceedings* 2002

[16] S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, D. Talia “CAMEL: A Parallel Cellular Tool for Interactive Simulation and Modelling,” *IEEE Computational Science & Engineering*, Vol. 3, No. 3, Fall 1996.

[17] T. Drayer, W. King, J. Tront, R. Conners, “MORRPH: A MOdular and Reprogrammable Real-time Processing Hardware,” in *Proc: IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, pp. 11-19, April 1995.

[18] J. D. Eckart “Cellang: Language Reference Manual,” Radford University, Radford, VA, April 1995.

[19] J. D. Eckart “A Cellular Automata Simulation System,” Radford University, Radford, VA, April 1995.

[20] N. Forbs “Evolution on a Chip: Evolvable Hardware Aims to Optimize Circuit Design,” in *IEEE Computing in Science and Engineering*, Volume 3, Number 3, pp 6-10, May/June 2001

[21] U. Frisch, B. Hasslacher, Y. Pomeau, “Lattice-gas automata for the Navier-Stokes equations,” *Phys. Rev. Lett*., 56:1505-1508,1986.

[22] M. Gardner “Mathematical Games, The fantastic combinations of John Conway’s new solitaire game ‘life’,” *Scientific American*, October 1970, pp 120-123.

[23] R. J. Gaylord, K. Nishidate "Excursions in Programming: A Cellular Automata Programming Toolkit", *Mathematica in Education and Research* Vol 5 Issue 3 1996

[24] D.C. Gazis, “Traffic Science,” Wiley, New York, 1973.

[25] D. George, B. K. Gunther and G.J. Milne "Modelling Concurrent Systems for Simulation on Reconfigurable Computers", *In Proc. 4th Annual Australasian Conference on Parallel and RealTime Systems, PART'97* , LNCS, SpringerVerlag, September 1997.

[26] D.L. Gerlough, "Proceedings of the 35th Annual Meeting," pp543, edited by Burggrat F. and Ward, Highway research board, Washington DC, 1956.

[27] P.G. Gipps, "MULTSIM : A Model for Simulating Vehicular Traffic on Multi-Lane Arterial Roads," pp292 - 298, edited by Newton P.W., Taylor M.A.P. and Sharpe R., Desktop Planning: in *Microcomputing Applications for Infrastructural Services Planning and Design*, Hargreen Publishing, Melbourne, 1986.

[28] P. Guan, "Cellular Automaton Public-Key Cryptosystems," *Complex Systems* Vol. 1, 1987.

[29] Gutowitz, Cryptography with Dynamical Systems [Online]. Available: http://www.santafe.edu/~hag/crypto/crypto.html.

[30] T. Habutsu, Y. Nishio, I. Sasase, and S. Mori, "A Secret Key Cryptosystem by Iterating a Chaotic Map," in *Proc of Eurocrypt '91* pp127-140, 1991

[31] C. Hochberger, R. Hoffmann & S. Waldschmidt "Compilation of CDL for Different Target Architectures" *Parallel Computing Technologies*, LNCS 964, Springer Verlag, 1995

[32] C. Hochberger, R. Hoffmann, K. Völkmann, S. Waldschmidt "Cellular Processing Environment " PARELEC 98, Bialystok, Poland 1998.

[33] C. Hochberger, R. Hoffmann, S. Waldschmidt "The Cells Start Walking: Moving Objects in CDL++," ACRI 98, Trieste, Italy, 1998

[34] C. Hochberger, R. Hoffmann, S. Waldschmidt "CDL++ for the Description of Moving Objects in Cellular Automata", PaCT99, , LNCS 1662, Springer Verlag, 1999

[35] R. Hoffman, "CEPRA Architecture Research," Technische Universität Darmstadt [Online] http://www.informatik.tu-darmstadt.de/MP/forschung/forschung_en.htm.

[36] R. Hoffmann, K. Völkmann, S. Waldschmidt, W. Heenes "GCA: Global Cellular Automata, A Flexible Parallel Model" *PaCT2001*, LNCS 2127, Springer Verlag 2001

[37] R. Hoffmann, B. Ulmann, S. Waldschmidt, K. Völkmann "The Machine CEPRA-S Configured for Stream Processing" in *FPGA 2001, 9th Int. Symposium. on Field Programable Gate Arrays*, 2001, Montery California

[38] R. Hoffmann, K. Völkmann. "CEPRA-8: A cellular processing machine". *Parallelism Hardware, Software and Applications*, Capri, 1994.

[39] W. Diffie, M. Hellman "New Directions in Cryptography," *IEEE Transactions on Information Theory* 1976.

[40] P. C. Johnson, R. Jackson, "Frictional-collisional constitutive relations for granular materials,with application to plane shearing," *Journal of Fluid Mechanics*, Vol. 176 pp 67-93, 1987.

[41] J. Kari "Reversibility of 2D Cellular Automata is Undecidable," *Physica D*, Vol 45 pp 379-385, 1990.

[42] J. Kari, "Cryptosystems based on reversible cellular automata," University of Turku, Finland preprint, April 1992.

[43] C. Langton "Computation at the edge of chaos : Phase transitions and emergent computation," *Physica D*, Vol 42, pp 12-37, 1990.

[44] C. Langton, and D. Hiebeler "Cellsim," [Online]. Available: http://liinwww.ira.uka.de/ca/software/cellsim/

[45] W. Luk, S. McKeever "Pebble: A Language for Parametrised and Reconfigurable Hardware Design," Field-Programmable Logic and Applications, LNCS 1482, Springer 1998

[46] N. Margolus "CAM-8: A Virtual Processor Cellular Automata Machine," *Parallel and Distributed Computing Systems: Proceedings of the ISCA International Conference*, Orlando, Florida, U.S.A., September 21-23, 1995.

[47] N. Margolus "Ultimate Computers," In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing* pp. 181- 186, Society for Industrial and Applied Mathematics, Philadelphia, 1995.

[48] N. Margolus "CAM-8: a computer architecture based on cellular automata," *Pattern Formation and Lattice-Gas Automata*, pp 167-187. American Mathematical Society, 1993.

[49] N. Margolus M. Toffoli, "STEP: A space time event processor. Software Reference," MIT Laboratory for Computer Science, Cambridge, MA 02139, 1995, [Online]. Available: www.ai.mit.edu/projects/im/cam8/ps/forth_ref.ps.

[50] I. Mathieson, J. Smith, N. Nguyen, "TRITRAM: Linking a Process Based Traffic Simulation to an Adaptive Traffic Controller,", in *Proc of the Application of New Technology to Transport Systems Conference, Melbourne*, Vol 1, pp 227-256 ITA Australia, 1995.

[51] D. McArthur, G.D.B. Cameron, M.D. While, B.J.N.Wylie "Paramics: Parallel microscopic traffic simulator," Kings Buildings Edinburgh.

[52] M. McDonal, M.A. Brackstone, "Proceedings of the 27th International Symposium on Automative Technology and Automation (ISATA)," Automotive Automation Ltd, Croydon , England, 1994

[53] G. McCaskill, G. Milne "Hardware Description and Verification Using the Circal-System" University of South Australia Research Report HDV-24-92, June 1992

[54] R. Milner. "Communication and Concurrency" Prentice Hall, 1989.

[55] G. Milne "CIRCAL: A Calculus for Circuit Description", *Integration, the VLSI Journal*, Vol. 1, Nos. 2 and 3, 1983.

[56] G. Milne "The Formal Description and Verification of Hardware Timing", *IEEE Transactions on Computers*, Vol. 40, No. 7, pp. 711-826, July 1991.

[57] G. Milne "The Formal Specification and Verification of Digital Systems", McGraw-Hill, 1994.

[58] G. Milne "CIRCAL and the Representation of Communication, Concurrency and Time," *ACM Transations on Programming Languages and Systems*, Vol 7, No 2, April 1985.

[59] G. Milne "Formal specification and verification of digital systems," McGraw-Hill, London 1994.

[60] G. Milne, P. Cockshott, G. McCaskill, P. Barrie. "Realising massively concurrent systems on the SPACE machine," in *Proc IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 26-32, 1993 and University of Strathclyde Research Report HDV-29-93,

[61] G. Milne D. George and B.K. Gunther "Experiments in Traffic Simulation using Reconfigurable Computing" *International Workshop on Reconfigurable Architectures*, Geneva 1997. IT Press Verlag, 1997.

[62] G. Milne, R. Milner " Concurrent Processes and their Syntax," *Journal of the ACM*, Vol. 26, No. 2, 1983

[63] G. Milne, P. Shaw. "A highly parrallel FPGA-based machine and its formal verification," *Lecture Notes in Computer Science*, No705. Springer-Verlag 1993.

[64] M. Mitchell, "Computation in Cellular Automata: A Selected Review, Non-standard Computation," edited by H.G. Schuster and T. Gramss T, V.C.H Weinheim, Verlagsgesellschaft, 1996.

[65] M. Mitchell, T. Haraber, J. Crutchfield, "Dynamics, Computation, and the 'Edge of Chaos': A Re-Examination," *Complexity: Metaphors, Models, and Reality, Santa Fe Institute Studies in the Sciences of Complexity, Proceedings* Volume 19, 497-513, Addison-Wesley, 1994.

[66] M. Mitchell, T. Haraber, J. Crutchfield, "Revisiting the Edge of Chaos: Evolving Cellular Automata to Perform Computations"

[67] M. Mitchell, J. Crutchfield, R. Das "Evolving Cellular Automata to Perform Computations" , *Handbook of Evolutionary Computation*, Oxford University Press.

[68] K. Nagel, M. Schreckenberg, "A cellular automation model for freeway traffic," *Journal of Physics. I France 2*, Vol 2, pp. 2221-2229, 1992.

[69] K. Nagel "Particle hopping models and traffic flow theory," *Physical Review E*, Vol 53, No 5, 1996.

[70] S. Nandi, B.K. Kar, Chaudhuri, "Theory and Applications of Cellular Automata in Cryptography", *IEEE Transactions on Computers*, Vol 43, No 12, pp 1346 – 1357December 1994.

[71] National Institute of Standards and Technology, FiniteState Machine definition, [Online]. Available: http://www.nist.gov/dads/HTML/finiteStateMachine.html

[72] C. Orovas "Cellular Associative Neural Networks" *PhD Thesis*, University of York, 1999

[73] J. Ousterhout, "Tcl and the Tk Toolkit," Addison-Wesley, 1993.

[74] I. J. Palmer, "Scamper," [Online]. Available: ftp://qilib.scn.rain.com/pub/simulation.

[75] "Paramics software suite" [Online]. Available: http://www.paramics.com/

[76] R. Porter, N Bergman "Evolving FPGA Based Cellular Automata" *Lecture Notes in Computer Science, No 1585*, Springer-Verlag 1999.

[77] K Preston, M. Duff, “Modern Cellular Automata : Theory and Applications,” Plenum Press, New York, 1985.

[78] M. Resnick “Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds,” Cambridge, MA: MIT Press, 1994

[79] A. Roscoe “The Theory and Practice of Concurrency” Prentice-Hall, 1997

[80] G. Russell, A. Cowie, J. McInnes, G. Milne, M. Bate “Simulating Vehicular Traffic Flows Using the Circal System,” University of Strathclyde Research Report/94/157, May 1994 and CONPAR-1-94

[81] G. Russell, N. Ferguson, G. Milne, J.McInnes. “The rapid simulation of urban traffic using field programmable gate arrays,” in *Proc. Application of New Technology to Transport Systems Conference*, Vol 1, Melbourne, Published by ITS Australia, pp.107-122, 1995

[82] G. Russell, A. Cowie, J. McInnes, G. Milne, M. Bate “Simulating Vehicular Traffic Flows Using the Circal System,” University of Strathclyde Research Report/94/157, May 1994

[83] G. Russell, P. Shaw, N. Ferguson, “Accurate Rapid Simulation of Urban Traffic using Discrete Modelling,” Department of Computer Studies, Napier University, Technical Report No. RR-96-1, Scotland, UK, 1996.

[84] A. Schadschneider and M. Schreckenberg “Cellular automation models and traffic flow,” *Journal of Physics. A, Mathematical and General*, Vol 8 No 1; pp. L679-L683, 1993.

[85] M. Shand, J .Vuillemin, “Fast Implementations of RSA Cryptography,” *IEEE Symposium on Computer Arithmetic, ARITH 11*, Windsor, ONT, Canada, 1993.

[86] P. Shaw, P. Cockshott, P. Barrie “Implementation of Lattice Gases Using FPGAs,” *Journal of VLSI Signal Processing*, November 1996, pp 51-66

[87] M. Sipper “Evolution of Parallel Cellular Machines,” *Lecture Notes in Computer Science, No 1194*, Springer-Verlag 1997

[88] G. Spezzano and D. Talia CARPET: “A Programming Language for Parallel Cellular Processing” *Proceedings of the European School on Parallel Programming Environments for HPC '96*, France, April 1996.

[89] I. Stephenson “Creature Processing, An Alternative Cellular Architecture,” University of York Technical Report ASEG92.04, February 1992

[90] I. Stephenson “Creatures, A Simulation Environment for Autonomous Behaviour,” University of York Technical Report ASEG92.16, June 1992

[91] S. R. Sternberg. “Bio-medical image processing.” *IEEE Computer*, pp 22-34, 1983.

[92] T. Junchaya and G.L. Chang “Exploring real-time traffic simulation with massively concurrent parallel computing architecture’” Transportation Research - C 1 (1), 57 - 76, 1993.

[93] S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, D. Talia “CAMEL: A Parallel Cellular Tool for Interactive Simulation and Modeling”, *IEEE Computational Science & Engineering*, Vol. 3, No. 3, Fall 1996.

[94] T. Toffoli, and N. Margolus “Cellular Automata Machines, A new environment for modelling,” The MIT Press, Cambridge, MA, 1987

[95] “Tritram (Traveller Information and Traffic Management),” [Online]. Available: http://volans.cbr.dit.csiro.au:8001/tritram/tritram.html (last accessed 20th January 1999)

[96] J. Von Neumann, “Theory of Self-Reproducing Automata,” edited and completed by A.W. Burks, Urbana, IL: University of Illinois Press, 1966.

[97] P. Wagner, K. Nagel, and D.E. Wolf “Realistic Multi-Lane Traffic Rules for Cellular Automata,” [Online]. Available: http://www.zpr.uni-koeln.de/GroupBachem/VERKEHR.PG/RESEARCH.P/papers/unpublished_papers multi-lane.ps.gz

[98] P. Wagner, “Traffic Simulations using cellular Automata: Comparison with Reality,” Center for Parallel Computing, University of Cologne, Report No 95. 214, 1995

[99] J. R. Weimar “JCASim: Cellular automata simulation system” [Online]. Available: http://www.jweimar.de/jcasim/

[100] G. Wilson “The life and times of cellular automata,” *New Scientist*, pp. 44-47 8 October 1988.

[101] S. Wolfram “Theory and applications of cellular automata,” *World Scientific Publishing Co. Ltd*, 1986.

[102] S. Wolfram “Computation Theory of Cellular Automata,” *Communications in Mathematical Physics*, pp 189 – 231, Springer-Verlag, 1984.

[103] S. Wolfram “Cellular Automata as models of complexity,” *Nature* Vol 311, pp 419-424 October 1984.

[104] S. Wolfram “Cellular Automata,” *Los Alamos Science*, pp 2-21, September 1983.

# Appendix 1 : Proof of Equivalence between Road Cells Types

To prove the equivalence between the long cell, described in the truth table in Figure 4-16, and a series of basic road cells both systems were modelled in Circal. Timing for the two models was provided by a two phase clock. This clock has two signals; t which clocks the cars through the basic road cells, and T which runs at 1/n the frequency of t and is used to clock blocks of cars through the long cells. These two signals are generated by the following Circal code.

```
Bool t,SlowT  // Global Clocks
int n=10      // length, must be even

Process CLOCK(t SlowT), /\end(t,SlowT)
// Process CLOCK
 CLOCK <- (SlowT.1 t.0) ((SlowT.0 t.1)n) CLOCK
```

The Long Road cell was modelled in a similar way to the basic road cells. As this cell can have three states (0,1,2) two bits are needed to encode these states. The cell state names used for the basic road cell were appended with a ‘0’ or ‘1’ to allow for two bits of data. The Circal code that describes the behaviour of this cell is then:

```
Process LongLane(Bool Next1,Next0,Prev1,Prev0,CS1,CS0){

// Creates a Long Lane of roadway

  static Bool ns0,ns1,cs0,cs1,next0,next1,prev0,prev1
  static Process C

  C <- (prev1.0 prev0.0 cs1.0 cs0.0 next1.0 next0.0 ns1.0 ns0.0) C +
       (prev1.0 prev0.0 cs1.0 cs0.0 next1.0 next0.1 ns1.0 ns0.0) C +
       (prev1.0 prev0.0 cs1.0 cs0.0 next1.1 next0.0 ns1.0 ns0.0) C +
       (prev1.0 prev0.0 cs1.0 cs0.1 next1.0 next0.0 ns1.0 ns0.0) C +
       (prev1.0 prev0.0 cs1.0 cs0.1 next1.0 next0.1 ns1.0 ns0.0) C +
       (prev1.0 prev0.0 cs1.0 cs0.1 next1.1 next0.0 ns1.0 ns0.1) C +
       (prev1.0 prev0.0 cs1.1 cs0.0 next1.0 next0.0 ns1.0 ns0.1) C +
       (prev1.0 prev0.0 cs1.1 cs0.0 next1.0 next0.1 ns1.0 ns0.1) C +
       (prev1.0 prev0.0 cs1.1 cs0.0 next1.1 next0.0 ns1.1 ns0.0) C +

       (prev1.0 prev0.1 cs1.0 cs0.0 next1.0 next0.0 ns1.0 ns0.1) C +
       (prev1.0 prev0.1 cs1.0 cs0.0 next1.0 next0.1 ns1.0 ns0.1) C +
       (prev1.0 prev0.1 cs1.0 cs0.0 next1.1 next0.0 ns1.0 ns0.1) C +
       (prev1.0 prev0.1 cs1.0 cs0.1 next1.0 next0.0 ns1.0 ns0.1) C +
```

```
        (prev1.0 prev0.1 cs1.0 cs0.1 next1.0 next0.1 ns1.0 ns0.1) C +
        (prev1.0 prev0.1 cs1.0 cs0.1 next1.1 next0.0 ns1.1 ns0.0) C +
        (prev1.0 prev0.1 cs1.1 cs0.0 next1.0 next0.0 ns1.0 ns0.1) C +
        (prev1.0 prev0.1 cs1.1 cs0.0 next1.0 next0.1 ns1.0 ns0.1) C +
        (prev1.0 prev0.1 cs1.1 cs0.0 next1.1 next0.0 ns1.1 ns0.0) C +

        (prev1.1 prev0.0 cs1.0 cs0.0 next1.0 next0.0 ns1.0 ns0.1) C +
        (prev1.1 prev0.0 cs1.0 cs0.0 next1.0 next0.1 ns1.0 ns0.1) C +
        (prev1.1 prev0.0 cs1.0 cs0.0 next1.1 next0.0 ns1.0 ns0.1) C +
        (prev1.1 prev0.0 cs1.0 cs0.1 next1.0 next0.0 ns1.0 ns0.1) C +
        (prev1.1 prev0.0 cs1.0 cs0.1 next1.0 next0.1 ns1.0 ns0.1) C +
        (prev1.1 prev0.0 cs1.0 cs0.1 next1.1 next0.0 ns1.1 ns0.0) C +
        (prev1.1 prev0.0 cs1.1 cs0.0 next1.0 next0.0 ns1.0 ns0.1) C +
        (prev1.1 prev0.0 cs1.1 cs0.0 next1.0 next0.1 ns1.0 ns0.1) C +
        (prev1.1 prev0.0 cs1.1 cs0.0 next1.1 next0.0 ns1.1 ns0.0) C

        return ((C * TimeT(ns0,cs0)*
TimeT(ns1,cs1))[Next1/next1,Next0/next0,Prev1/prev1,Prev0/prev0,CS1/cs1,CS0
/cs0] - ns0 - ns1)
}
```

The time process for this cell is identical to that of the basic road cell except it is clocked by the slower T clock. This process is:

```
Process TimeT (Bool Nextstate, Currentstate) {

  static Bool nextstate,currentstate
  static Process F, E

  E <- (SlowT.1 nextstate.0 currentstate.0) E +
       (SlowT.1 nextstate.1 currentstate.0) F +
       (SlowT.0 nextstate.0 currentstate.0) E +
       (SlowT.0 nextstate.1 currentstate.0) E

  F <- (SlowT.1 nextstate.0 currentstate.1) E +
       (SlowT.1 nextstate.1 currentstate.1) F +
       (SlowT.0 nextstate.0 currentstate.1) F +
       (SlowT.0 nextstate.1 currentstate.1) F

  return ((E[Nextstate/nextstate,Currentstate/currentstate]))
}
```

To construct an equivalent process from basic road cells requires a number of road cells to be connected in series and then a number of constraints added to restrict the behaviour to that of the LongLane process.

A process that connects a number of basic road cells together is shown below. This process also provides at output the state of both end cells. It is possible from these two states to determine whether the block of cells are either all empty (both end cells will be empty), 50/50 (only one of the end cells will be full), or full (both end cells will be full). It is possible to deduce this information as one of the constraints (described later) restricts the input to alternate zeros and ones, or all zeros. The process is:

```
Process RoadLaneTest(Bool next,prev,CSB0,CSB1,CSF,int length){

  Process lane
  int i=0
  Bool states[length]

  // 1st cell
  lane = BasicRoadCell (states[1], prev, states[0])

  // compose middle cells together
  for (i=1; i<(length-1); i++)
    lane = lane * BasicRoadCell(states[i+1], states[i-1], states[i])

  // last cell
  i = length-1
  lane = lane * BasicRoadCell(next, states[i-1], states[i])

  lane = lane[CSB0/states[0],CSB1/states[i],CSF/states[i]]

   for (i=1; i<(length-1); i++)
     lane = lane - states[i]

  return lane
}
```

The process that converts the two end state signals of the basic road cells to the state values of the LongLane is shown below (described above) In this process CSB* are the end states of road cells, CS* are the interpreted states corresponding to the LongLane cell.

```
Process CSBtoCS (Bool CSB1,CSB0,CS1,CS0) {

  static Bool csb0, csb1, cs0, cs1
  static Process C

  C <- (csb1.0 csb0.0 cs1.0 cs0.0) C +
       (csb1.0 csb0.1 cs1.0 cs0.1) C +
       (csb1.1 csb0.0 cs1.0 cs0.1) C +
       (csb1.1 csb0.1 cs1.1 cs0.0) C

  return ((C[CSB1/csb1,CSB0/csb0,CS1/cs1,CS0/cs0]))
}
```

The state of the previous (long) cell, with the current state of this cell can be used to generate the input to the series of road cells. As stated above the Long road cell density will increase by on if there are cars in cell behind and this cell is not full. To increase the state value by one a car must flow into the section every other clock cycle. A process that generates a car every other clock cycle if the input conditions are met is shown below. CS* are the input states that the LongLane cell would expect from its proceeding cell, next is the output that feeds into the first cell of the basic road cells, CSX1 is bit one of the current state generated by CSBtoCS above.

```
Process CStoNext(Bool CS1,CS0,Next,CSX1){

  static Bool cs1, cs0, csx1, next
  static Process F,E

  E <- (t.1 cs1.0 cs0.0 csx1.0 next.0) E +
       (t.1 cs1.0 cs0.1 csx1.0 next.0) F +
       (t.1 cs1.1 cs0.0 csx1.0 next.0) F +
       (t.1 cs1.0 cs0.0 csx1.1 next.0) E +
       (t.1 cs1.0 cs0.1 csx1.1 next.0) E +
       (t.1 cs1.1 cs0.0 csx1.1 next.0) E +
       (t.0 cs1.0 cs0.0 csx1.0 next.0) E +
       (t.0 cs1.0 cs0.1 csx1.0 next.0) E +
       (t.0 cs1.1 cs0.0 csx1.0 next.0) E +
       (t.0 cs1.0 cs0.0 csx1.1 next.0) E +
       (t.0 cs1.0 cs0.1 csx1.1 next.0) E +
       (t.0 cs1.1 cs0.0 csx1.1 next.0) E

  F <- (t.1 cs1.0 cs0.0 csx1.0 next.0) E +
       (t.1 cs1.0 cs0.1 csx1.0 next.1) E +
```

```
        (t.1 cs1.1 cs0.0 csx1.0 next.1) E +
        (t.1 cs1.0 cs0.0 csx1.1 next.0) E +
        (t.1 cs1.0 cs0.1 csx1.1 next.0) E +
        (t.1 cs1.1 cs0.0 csx1.1 next.0) E +
        (t.0 cs1.0 cs0.0 csx1.0 next.0) F +
        (t.0 cs1.0 cs0.1 csx1.0 next.1) F +
        (t.0 cs1.1 cs0.0 csx1.0 next.1) F +
        (t.0 cs1.0 cs0.0 csx1.1 next.1) F +
        (t.0 cs1.0 cs0.1 csx1.1 next.1) F +
        (t.0 cs1.1 cs0.0 csx1.1 next.1) F

  return (E[CS1/cs1,CS0/cs0,Next/next,CSX1/csx1])
}
```

In the LongLane cell the input and output states are constrained to three values. A Circal process that can be used to restrain the inputs of the test roadway is shown below.

```
Process RestrictCS (Bool CSI1,CSI0) {

  static Bool csi1,csi0
  static Process C

  C <- (csi1.0 csi0.0) C +
       (csi1.0 csi0.1) C +
       (csi1.1 csi0.0) C
  return (C[CSI1/csi1,CSI0/csi0])
}
```

The above constraints were composed into one process. This process also has latches on the interface to the outside to constrain changes be synchronous with the slow clock (as for the LongLane). This composed process is shown below.

```
Process LongToBRC (Bool CSO1,CSO0,CSI1,CSI0,CSB1,CSB0,Next) {

  static Bool csi1,csi0,next,csI0,csI1,csb1,csb0,cso1,cso0,csO0,csO1
  static Process C

  C = CStoNext(csi1,csi0,next,csO1)
  C = C * CSBtoCS(csb1,csb0,cso1,cso0)
  C = C * CDLATCH(SlowT,cso0,csO0)
  C = C * CDLATCH(SlowT,cso1,csO1)
  C = C * CDLATCH(SlowT,csI0,csi0)
  C = C * CDLATCH(SlowT,csI1,csi1)
```

```
  C = C * RestrictCS(csI1,csI0)
  C = C - (csi1 csi0 cso1 cso0)

  return (C[CSO1/csO1,CSO0/csO0,CSI1/csI1,CSI0/csI0,CSB1/csb1,
            CSB0/csb0,Next/next])
}
```

The ‘next’ input to the last cell in the series of basic road cells can be connected the state bit one of the next LongLand cell. This is because a long lane cell will accept a maximum flow of cars if it is not full (i.e. state bit 1 = 0). The final process can then be generated by composing all of these processes together, the Circal process and a diagram showing the interconnection between the process is shown below.

```
Process TestBRCRoad = (RoadLaneTest(next,cs,CSB0,CSB1,CSF,n) *
                CDLATCH(SlowT,next1,next) *
                      LongToBRC(CSO1,CSO0,CSI1,CSI0,CSB1,CSB0,cs) *
                AND(next0,next1,t0)  *
                RestrictCS(next0,next1) *
                CLOCK) - (CSB1 CSB0 cs t0 t CSF next)
```

The basic road cells take a complete cycle of the clock process to load whereas the LongLane will load its state at the next slow clock tick (T). To overcome this the inputs to the LongLand were delayed by one clock tick using latches:

```
Process TestLongRoad = (LongLane(next1,next0,CSI1,CSI0,t1,t0) *
                CDLATCH(SlowT,t1,CSO1) *
                CDLATCH(SlowT,t0,CSO0) *
                 CLOCK) - (t t0 t1)
```

The two processes were compared using the Circal equivalence operator that showed that they were equivalent.

```
print "\nTestBRCRoad == TestLongRoad: "
print (TestBRCRoad==TestLongRoad)
print "\n"

Process TestLongRoad1 = LongLane(next1,next0,CSI1,CSI0,CSO1,CSO0)
Process TestLongRoad2 = LongLane(next1,next0,CSI1,CSI0,CSO1,CSO0) *
RestrictCS(CSO1,CSO0)
Process TestLongRoadLogic =
LongLaneLogic(next1,next0,CSI1,CSI0,CSO1,CSO0)*RestrictCS(next0,next1)*Rest
rictCS(CSI0,CSI1)
```

```
print "\nTestLongRoad1==TestLongRoadLogic: "
print (TestLongRoad1==TestLongRoadLogic)
print "\n"

print "\nTestLongRoad2==TestLongRoad (prove it can't produce illegal O/P):
"
print (TestLongRoad2==TestLongRoad1)
print "\n"
```

# Appendix 2: Results of Traffic Simulation Experiments

## Experiment 1

This experiment simulates a clear lane of carriageway and is used to demonstrate the inflow and outflow properties from a clear section of roadway.

## Input {0} Initial State {0}

This combination does not contain any cars and is therefore of no interest. It will not be included in further experiment results.

## Input {0} Initial State {1}

This result shows the outflow from a carriageway containing stationary cars and no input.

```
.............................11111111111111111111111111111111111111 X........
.............................111111111111111111111111111111111111 1.X.......
.............................1111111111111111111111111111111111 1 X.X......
.............................11111111111111111111111111111111 1 1.X.X.....
.............................111111111111111111111111111111111 1 1 X.X.X....
.............................1111111111111111111111111111111111 1 1 1.X.X.X...
.............................111111111111111111111111111111111 1 1 1 X.X.X.X..
.............................1111111111111111111111111111111111 1 1 1 1.X.X.X.X.
.............................11111111111111111111111111111111 1 1 1 1 X.X.X.X.X
                                       :
.............................111111 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
.............................11111 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
.............................1111 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
.............................111 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
.............................11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
.............................1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
............................. 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
.............................  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
.............................   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
.............................    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
.............................     1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
                                        :
.............................                                  1 1 1 X.X.X.X.X
.............................                                   1 1 1.X.X.X.X.
.............................                                    1 1 X.X.X.X.X
.............................                                     1 1.X.X.X.X.
.............................                                      1 X.X.X.X.X
.............................                                       1.X.X.X.X.
.............................                                        X.X.X.X.X
.............................                                         .X.X.X.X.
.............................                                          ..X.X.X.X
.............................                                           ...X.X.X.
```

```
.............................                                    ....X.X.X
```

## Input {0} Initial State {10}

This result shows the outflow from a carriageway containing the maximum number of moving cars and no input.

```
.............................  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.........
.............................   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X........
.............................    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.......
.............................     1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X......
.............................      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.....
.............................       1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X....
                                         :
.............................                                  1 1 1.X.X.X.X.
.............................                                   1 1 X.X.X.X.X
.............................                                    1 1.X.X.X.X.
.............................                                     1 X.X.X.X.X
.............................                                      1.X.X.X.X.
.............................                                       X.X.X.X.X
.............................                                        .X.X.X.X.
.............................                                        ..X.X.X.X
.............................                                        ...X.X.X.
```

each car travels forward one cell each clock cycle until all cars have left the system.

## Input {0} Initial State {100}

This result shows the outflow from a carriageway containing a medium number of moving cars and no input. The result for this experiment was the same as above except the cars were spaced two cells apart.

## Input {0} Initial State {1000}

This result shows the outflow from a carriageway containing a low number of moving cars and no input. The result for this experiment was the same as above except the cars were spaced two cells apart.

## Input {10} Initial State {0}

This result shows the inflow of the maximum number of moving cars into a empty carriageway.

```
.X.X.X.X.X.X.X.X.X.X.X.X.X.X                                     .........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1                                    .........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1                                   .........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1                                  .........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1                                 .........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1                                .........
                                         :
```

```
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   .........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  .........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 .........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.......
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X......
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.....
```

Cars enter the experiment and continue through, leaving on the opposite side.

## Input {10} Initial State {1}

Shows the effect of maximum flow traffic meeting a section of stationary traffic. The stationary traffic is able to leave the system at the same rate as cars are entering the system so a cars joins the end of the queues at the same rate cars are leaving the front of the queue, hence the queue moves backwards down the road by one traffic cell each clock tick.

```
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X1111111111111111111111111111111111111111 X........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.XX1111111111111111111111111111111111111111 1.X.......
.X.X.X.X.X.X.X.X.X.X.X.X.X.XXX1111111111111111111111111111111111111111 1 X.X......
X.X.X.X.X.X.X.X.X.X.X.X.X.XXXX1111111111111111111111111111111111111111 1 1.X.X.....
.X.X.X.X.X.X.X.X.X.X.X.X.X.XXXXX1111111111111111111111111111111111111111 1 1 X.X.X....
X.X.X.X.X.X.X.X.X.X.X.X.X.XXXXXX1111111111111111111111111111111111111111 1 1 1.X.X.X...
.X.X.X.X.X.X.X.X.X.X.X.X.X.XXXXXXX1111111111111111111111111111111111111111 1 1 1 X.X.X.X..
X.X.X.X.X.X.X.X.X.X.X.X.X.XXXXXXXX1111111111111111111111111111111111111111 1 1 1 1.X.X.X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.XXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 X.X.X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.XXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1.X.X.X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.XXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1 X.X.X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.XXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1 1.X.X.X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.XXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1 1 X.X.X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.XXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1 1.X.X.X.X.
                                                                              :
```

## Input {10} Initial State {10}

Shows cars enter a system containing the maximum number of moving cars at maximum flow rate. The results of this experiment are the same as the resulting system in Input {1 0} initial state {0} above.

## Input {10} Initial State {100}

As there are spaces between all cars in the system each car moves forward one cell at each time step.

## Input {10} Initial State {1000}

As there are spaces between all cars in the system each car moves forward one cell at each time step.

## Input {100} Initial State {0}

As for Input {10} initial state {0} above except car spacing is two.

## Input {100} Initial State {1}

Shows the effect of medium flow traffic meeting a section of stationary traffic. The stationary traffic is able to leave the queue at a faster rate (one car per clock cycle) than cars are entering the queue (one car every other clock cycle) The queues therefore moves backwards (cars leaving from the front) and gets shorter (every other clock cycle) until it has dissipated. Once the queue has dissipated the system is the same as for Input {100} initial state {10}.

```
X..X..X..X..X..X..X..X..X..X.X1111111111111111111111111111111111111111 X.......
.X..X..X..X..X..X..X..X..X..XX1111111111111111111111111111111111111111 1.X.....
..X..X..X..X..X..X..X..X..X.XX1111111111111111111111111111111111111111 1 X.X.....
X..X..X..X..X..X..X..X..X..XXX1111111111111111111111111111111111111111 1 1.X.X....
.X..X..X..X..X..X..X..X..X.XXX1111111111111111111111111111111111111111 1 1 X.X.X...
..X..X..X..X..X..X..X..X..XXXX1111111111111111111111111111111111111111 1 1 1.X.X.X..
X..X..X..X..X..X..X..X..X.XXXX1111111111111111111111111111111111111111 1 1 1 X.X.X.X.
.X..X..X..X..X..X..X..X..XXXXX1111111111111111111111111111111111111111 1 1 1 1.X.X.X.X
..X..X..X..X..X..X..X..X.XXXXX1111111111111111111111111111111111111111 1 1 1 1 X.X.X.X.X
X..X..X..X..X..X..X..X..XXXXXX1111111111111111111111111111111111111111 1 1 1 1 1.X.X.X.X.
                                        :
X..X.XXXXXXXXXXXXXXXX.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
.X..XXXXXXXXXXXXXXXX.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
..X.XXXXXXXXXXXXXXX.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
X..XXXXXXXXXXXXXXX.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
.X.XXXXXXXXXXXXXX.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
..XXXXXXXXXXXXXX.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
X.XXXXXXXXXXXXX.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
```

## Input {100} Initial State {10}

As there are spaces between all cars in the system each car moves forward one cell at each time step.

## Input {100} Initial State {100}

As there are spaces between all cars in the system each car moves forward one cell at each time step.

## Input {100} Initial State {1000}

As there are spaces between all cars in the system each car moves forward one cell at each time step.

## Input {1000} Initial State {0}

As for Input {10} initial state {0} above except car spacing is three.

## Input {1000} Initial State {1}

As for Input {100} initial state {1} above except now the cars are entering the queue at one third the rate of those leaving, hence the queue dissolves at a quicker rate.

```
...X...X...X...X...X...X..11111111111111111111111111111111111111 X.......
X...X...X...X...X...X...X.11111111111111111111111111111111111111 1.X.....
.X...X...X...X...X...X...X1111111111111111111111111111111111111 1 X.X.....
..X...X...X...X...X...X..X1111111111111111111111111111111111111 1 1.X.X....
...X...X...X...X...X...X.X1111111111111111111111111111111111111 1 1 X.X.X...
X...X...X...X...X...X...XX1111111111111111111111111111111111111 1 1 1.X.X.X..
.X...X...X...X...X...X..XX1111111111111111111111111111111111111 1 1 1 X.X.X.X.
..X...X...X...X...X...X.XX1111111111111111111111111111111111111 1 1 1 1.X.X.X.
...X...X...X...X...X...XXX1111111111111111111111111111111111111 1 1 1 1 X.X.X.X
                                    :
..X...X...X...XXXXXXXX.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
...X...X...X..XXXXXXX.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
X...X...X...X.XXXXXX.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
.X...X...X...XXXXXX.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
..X...X...X..XXXXX.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
...X...X...X.XXXX.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
X...X...X...XXXX.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
.X...X...X..XXX.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
..X...X...X.XX.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
...X...X...XX.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
X...X...X..X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
.X...X...X..X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
..X...X...X..X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
...X...X...X..X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
```

## Input {1000} Initial State {10}

As there are spaces between all cars in the system each car moves forward one cell at each time step.

## Input {1000} Initial State {100}

As there are spaces between all cars in the system each car moves forward one cell at each time step.

## Input {1000} Initial State {1000}

As there are spaces between all cars in the system each car moves forward one cell at each time step.

## Experiment 2

This experiment simulates a lane of carriageway with a speed restriction at one end. The speed restriction is simulated by a cell which requires two clock ticks for a car to pass through. The speed restricting cell is called slow .

## Input {0} Initial State {1}

This result shows the outflow from a carriageway containing stationary cars and no input.

```
.............................1111111111111111111111111111111111111111 C.........
.............................1111111111111111111111111111111111111111 1C........
.............................1111111111111111111111111111111111111111 11:X.......
.............................1111111111111111111111111111111111111111 11 C.X.....
.............................1111111111111111111111111111111111111111 11 1C..X.....
.............................1111111111111111111111111111111111111111 11 11:X..X....
.............................1111111111111111111111111111111111111111 11 11 C.X..X....
.............................1111111111111111111111111111111111111111 11 11 1C..X..X...
                                          :
.............................11111 11 11 11 11 11 11 11 11 11 11 11 11 1C..X..X..X
.............................1111 11 11 11 11 11 11 11 11 11 11 11 11 11:X..X..X..
.............................111 11 11 11 11 11 11 11 11 11 11 11 11 11 C.X..X..X.
.............................11 11 11 11 11 11 11 11 11 11 11 11 11 11 1C..X..X..X
.............................1 11 11 11 11 11 11 11 11 11 11 11 11 11 11:X..X..X..
.............................. 11 11 11 11 11 11 11 11 11 11 11 11 11 C.X..X..X.
.............................. 1 11 11 11 11 11 11 11 11 11 11 11 11 11 1C..X..X..X
..............................  11 11 11 11 11 11 11 11 11 11 11 11 11 11:X..X..X..
..............................  1 11 11 11 11 11 11 11 11 11 11 11 11 11 C.X..X..X.
..............................   11 11 11 11 11 11 11 11 11 11 11 11 11 11 1C..X..X..X
..............................   1 11 11 11 11 11 11 11 11 11 11 11 11 11:X..X..X..
..............................    11 11 11 11 11 11 11 11 11 11 11 11 11 C.X..X..X.
```

As can be seen a car leaves the section every third clock cycle (as opposed to every other clock cycle in the experiment above). This results in small blockages moving back down the road. This has the effect of slowing the traffic (on a macro scale) as each car only advances two cells every three clock cycles.

## Input {0} Initial State {10}

This result shows the outflow from a carriageway containing the maximum number of moving cars and no input.

```
.............................. 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1C........
..............................  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11:X.......
..............................   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 C.X.....
..............................    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1C..X.....
..............................     1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11:X..X.....
..............................      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 C.X..X....
```

```
.............................      1 1 1 1 1 1 1 1 1 1 1 1 11 11 1C..X..X...
.............................       1 1 1 1 1 1 1 1 1 1 1 1 11 11 11:X..X..X..
.............................        1 1 1 1 1 1 1 1 1 1 1 11 11 11 C.X..X..X.
.............................         1 1 1 1 1 1 1 1 1 1 11 11 11 1C..X..X..X
.............................          1 1 1 1 1 1 1 1 1 11 11 11 11:X..X..X..
.............................           1 1 1 1 1 1 1 1 11 11 11 11 C.X..X..X.
.............................            1 1 1 1 1 1 1 11 11 11 11 1C..X..X..X
.............................             1 1 1 1 1 1 1 11 11 11 11 11:X..X..X..
.............................              1 1 1 1 1 1 11 11 11 11 11 C.X..X..X.
.............................               1 1 1 1 1 11 11 11 11 11 1C..X..X..X
.............................                1 1 1 1 11 11 11 11 11 11:X..X..X..
.............................                 1 1 11 11 11 11 11 11 C.X..X..X.
.............................                  1 11 11 11 11 11 11 1C..X..X..X
.............................                   11 11 11 11 11 11 11:X..X..X..
.............................                   1 11 11 11 11 11 11 C.X..X..X.
.............................                    11 11 11 11 11 11 1C..X..X..X
.............................                    1 11 11 11 11 11 11:X..X..X..
.............................                     11 11 11 11 11 11 C.X..X..X.
```

The cars in this experiment bunch up as the reach the back of the tail back from the slow cell. The cars then advance at 2/3 speed as above

## Input {0} Initial State {100}

This result shows the outflow from a carriageway containing a medium number of moving cars and no input.

```
.............................. 1  1  1  1  1  1  1  1  1  1  1  1  1  C.........
..............................  1  1  1  1  1  1  1  1  1  1  1  1  1 C.........
...............................  1  1  1  1  1  1  1  1  1  1  1  1  1:X........
................................  1  1  1  1  1  1  1  1  1  1  1  1  C.X.......
.................................  1  1  1  1  1  1  1  1  1  1  1  1 C..X......
..................................  1  1  1  1  1  1  1  1  1  1  1  1:X..X.....
...................................  1  1  1  1  1  1  1  1  1  1  1  C.X..X....
....................................  1  1  1  1  1  1  1  1  1  1  1 C..X..X...
.....................................  1  1  1  1  1  1  1  1  1  1  1:X..X..X..
```

In this experiment the flow of cars in the system is the same as the flow through the slow cell so cars can flow at maximum speed until they reach the slow cell (i.e. the slow cell is always empty when a car arrives at it).

## Input {0} Initial State {1000}

This result shows the outflow from a carriageway containing a low number of moving cars and no input.

The result for this experiment was the same as above except the flow rate of the cars is less than the maximum flow rate of the slow cell.

## Input {10} Initial State {0}

This result shows the inflow of the maximum number of moving cars into a empty carriageway.

```
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  :.........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 :.........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1:.........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 C.........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1C.........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11:X........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 C.X.......
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1C..X......
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11:X..X.....
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 C.X..X....
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 1C..X..X...
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11:X..X..X..
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 C.X..X..X.
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 1C..X..X..X
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11:X..X..X..
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 C.X..X..X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 1C..X..X..X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 11:X..X..X..
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 C.X..X..X.
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 1C..X..X..X
```

Cars enter the experiment travel at maximum speed until the reach the tail back of slow cars, there speed in then reduced to 2/3 maximum. The length of the tail back grows by one car every clock cycle (speed of cars entering the system).

## Input {10} Initial State {1}

Shows the effect of maximum flow traffic meeting a section of stationary traffic. The stationary traffic is able to leave the jam at the same rate as cars are entering the system so a cars joins the end of the queues at the same rate cars are leaving the front of the queue, hence the queue moves backwards down the road by one traffic cell each clock tick. Cars can only leave the system at 2/3 maximum speed so the density of cars between the stationary queue and the slow cell is greater than for the same conditions in experiment 1

```
.X.X.X.X.X.X.X.X.X.X.X.X.X.X1111111111111111111111111111111111111111:X........
X.X.X.X.X.X.X.X.X.X.X.X.X.XX1111111111111111111111111111111111111111 C.X.......
.X.X.X.X.X.X.X.X.X.X.X.X.XXX1111111111111111111111111111111111111111 1C..X......
X.X.X.X.X.X.X.X.X.X.X.X.XXXX1111111111111111111111111111111111111111 11:X..X.....
.X.X.X.X.X.X.X.X.X.X.X.XXXXX1111111111111111111111111111111111111111 11 C.X..X....
X.X.X.X.X.X.X.X.X.X.X.XXXXXX1111111111111111111111111111111111111111 11 1C..X..X...
```

```
.X.X.X.X.X.X.X.X.X.X.XXXXXXX1111111111111111111111111111111 11 11:X..X..X..
X.X.X.X.X.X.X.X.X.X.XXXXXXXX1111111111111111111111111111111 11 11 C.X..X..X.
.X.X.X.X.X.X.X.X.X.XXXXXXXXX1111111111111111111111111111111 11 11 1C..X..X..X
X.X.X.X.X.X.X.X.X.XXXXXXXXXX1111111111111111111111111111111 11 11 11:X..X..X..
.X.X.X.X.X.X.X.X.XXXXXXXXXXX1111111111111111111111111111111 11 11 11 C.X..X..X.
X.X.X.X.X.X.X.X.XXXXXXXXXXXX1111111111111111111111111111111 11 11 11 1C..X..X..X
.X.X.X.X.X.X.X.XXXXXXXXXXXXX1111111111111111111111111111111 11 11 11 11:X..X..X..
X.X.X.X.X.X.X.XXXXXXXXXXXXXX1111111111111111111111111111111 11 11 11 11 C.X..X..X.
.X.X.X.X.X.X.XXXXXXXXXXXXXXX1111111111111111111111111111111 11 11 11 11 1C..X..X..X
X.X.X.X.X.X.XXXXXXXXXXXXXXXX1111111111111111111111111111111 11 11 11 11 11:X..X..X..
.X.X.X.X.X.XXXXXXXXXXXXXXXXX1111111111111111111111111111111 11 11 11 11 11 C.X..X..X.
```

## Input {10} Initial State {10}

Shows cars enter a system containing the maximum number of moving cars at maximum flow rate. The results of this experiment are the same as the resulting system in Input {1 0} initial state {0} above.

## Input {10} Initial State {100}

the cars from the initial state can leave the system at there current flow rate. The resulting system is then the same as for Input {1 0} initial state {1 0} above.

## Input {10} Initial State {1000}

the cars from the initial state can leave the system at there current flow rate. The resulting system is then the same as for Input {1 0} initial state {1 0} above.

## Input {100} Initial State {0}

All cars can leave the system at current flow rate.

## Input {100} Initial State {1}

Shows the effect of medium flow traffic meeting a section of stationary traffic. The stationary traffic is able to leave the queue at a faster rate (one car per clock cycle) than cars are entering the queue (one car every other clock cycle) The queues therefore moves backwards (cars leaving from the front) and gets shorter (every other clock cycle) until it has dissipated. Once the queue has dissipated the system is the same as for Input {100} initial state {10}

```
X..X..X..X..X..X..X..X..X.X1111111111111111111111111111111111111111 C.........
.X..X..X..X..X..X..X..X..XX1111111111111111111111111111111111111111 1C.........
..X..X..X..X..X..X..X..X.XX1111111111111111111111111111111111111111 11:X.......
X..X..X..X..X..X..X..X..XXX1111111111111111111111111111111111111111 11 C.X.......
.X..X..X..X..X..X..X..X.XXX1111111111111111111111111111111111111111 11 1C..X......
..X..X..X..X..X..X..X..XXXX1111111111111111111111111111111111111111 11 11:X..X.....
X..X..X..X..X..X..X..X.XXXX1111111111111111111111111111111111111111 11 11 C.X..X....
```

```
.X..X..X..X..X..X..X..X..XXXXX11111111111111111111111111111111 11 11 1C..X..X...
..X..X..X..X..X..X..X..X.XXXXX11111111111111111111111111111111 11 11 11:X..X..X..
X..X..X..X..X..X..X..X..XXXXXX11111111111111111111111111111111 11 11 11 C.X..X..X.
                                        :
.X..XXXXXXXXXXXXXXX.XX.XX.XX.11 11 11 11 11 11 11 11 11 11 11 11 11 1C..X..X..X
..X.XXXXXXXXXXXXXXX.XX.XX.XX.X1 11 11 11 11 11 11 11 11 11 11 11 11 11:X..X..X..
X..XXXXXXXXXXXXXXX.XX.XX.XX.XX 11 11 11 11 11 11 11 11 11 11 11 11 11 C.X..X..X.
.X.XXXXXXXXXXXXXX.XX.XX.XX.XX.11 11 11 11 11 11 11 11 11 11 11 11 11 1C..X..X..X
..XXXXXXXXXXXXXX.XX.XX.XX.XX.X1 11 11 11 11 11 11 11 11 11 11 11 11 11:X..X..X..
X.XXXXXXXXXXXXX.XX.XX.XX.XX.XX 11 11 11 11 11 11 11 11 11 11 11 11 11 C.X..X..X.
.XXXXXXXXXXXXX.XX.XX.XX.XX.XX.11 11 11 11 11 11 11 11 11 11 11 11 11 1C..X..X..X
.XXXXXXXXXXXX.XX.XX.XX.XX.XX.X1 11 11 11 11 11 11 11 11 11 11 11 11 11:X..X..X..
```

## Input {100} Initial State {10}

Cars leaving the initial condition cause a tail back until the initial condition has dissipated. The length of the tail back them remains constant as the rate of cars joining the system is the same as the rate of cars leaving the system.

```
X..X..X..X..X..X..X..X..X..X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1C.........
.X..X..X..X..X..X..X..X..X..X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11:X........
..X..X..X..X..X..X..X..X..X..X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 C.X.......
X..X..X..X..X..X..X..X..X..X..1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1C..X......
.X..X..X..X..X..X..X..X..X..X. 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11:X..X.....
..X..X..X..X..X..X..X..X..X..X  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 C.X..X....
X..X..X..X..X..X..X..X..X..X..1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 1C..X..X...
.X..X..X..X..X..X..X..X..X..X. 1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11:X..X..X..
..X..X..X..X..X..X..X..X..X..X  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 C.X..X..X.
X..X..X..X..X..X..X..X..X..X..1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 1C..X..X..X
.X..X..X..X..X..X..X..X..X..X. 1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11:X..X..X..
..X..X..X..X..X..X..X..X..X..X  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 C.X..X..X.
X..X..X..X..X..X..X..X..X..X..1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 1C..X..X..X
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 11:X..X..X..
..X..X..X..X..X..X..X..X..X..X  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 11 C.X..X..X.
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 1C..X..X..X
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 11:X..X..X..
..X..X..X..X..X..X..X..X..X..X  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 11 C.X..X..X.
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 1C..X..X..X
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 11:X..X..X..
..X..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 11 11 11 11 11 C.X..X..X.
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  11 11 11 11 11 11 11 11 1C..X..X..X
```

## Input {100} Initial State {100}

All cars can leave the system at there current flow rate.

## Input {100} Initial State {1000}

All cars can leave the system at there current flow rate.

## Input {1000} Initial State {0}

As for Input {10} initial state {0} above except car spacing is three.

## Input {1000} Initial State {1}

As for Input {100} initial state {1} above except now the cars are entering the queue at one third the rate of those leaving, hence the queue dissolves at a quicker rate. The resulting tail back is able to dissolve as the input to the system is less than the output from the system (i.e. a car enters every 4 ticks, one leaves every 3 ticks).

## Input {1000} Initial State {10}

As for Input {100} initial state {10} above except the queue builds slower and the resulting tail back is dissipated (as for Input {100} initial state {1} above).

```
...X...X...X...X...X...X.. 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1:X.......
X...X...X...X...X...X...X.  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 C.X.......
.X...X...X...X...X...X...X   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1C..X......
..X...X...X...X...X...X...1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11:X..X.....
...X...X...X...X...X...X.. 1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 C.X..X....
X...X...X...X...X...X...X.  1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1C..X..X...
.X...X...X...X...X...X...X   1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 11:X..X..X..
..X...X...X...X...X...X...1   1   1 1 1 1 1 1 1 1 1 1 1 1 1 11 11 C.X..X..X.
...X...X...X...X...X...X.. 1   1   1 1 1 1 1 1 1 1 1 1 1 11 11 1C..X..X..X
X...X...X...X...X...X...X.  1   1   1 1 1 1 1 1 1 1 1 1 11 11 11:X..X..X..
.X...X...X...X...X...X...X   1   1   1 1 1 1 1 1 1 1 1 11 11 11 C.X..X..X.
..X...X...X...X...X...X...1   1   1   1 1 1 1 1 1 1 1 11 11 11 1C..X..X..X
...X...X...X...X...X...X.. 1   1   1   1 1 1 1 1 1 1 11 11 11 11:X..X..X..
X...X...X...X...X...X...X.  1   1   1   1 1 1 1 1 1 1 11 11 11 11 C.X..X..X.
.X...X...X...X...X...X...X   1   1   1   1 1 1 1 1 1 1 11 11 11 11 1C..X..X..X
..X...X...X...X...X...X...1   1   1   1   1 1 1 1 1 11 11 11 11 11:X..X..X..
...X...X...X...X...X...X.. 1   1   1   1   1 1 1 1 11 11 11 11 11 C.X..X..X.
X...X...X...X...X...X...X.  1   1   1   1   1 1 1 11 11 11 11 11 1C..X..X..X
.X...X...X...X...X...X...X   1   1   1   1   1 1 11 11 11 11 11 11:X..X..X..
..X...X...X...X...X...X...1   1   1   1   1   1 11 11 11 11 11 11 C.X..X..X.
...X...X...X...X...X...X.. 1   1   1   1   1   11 11 11 11 11 11 1C..X..X..X
X...X...X...X...X...X...X.  1   1   1   1   1  1 11 11 11 11 11 11:X..X..X..
.X...X...X...X...X...X...X   1   1   1   1   1  11 11 11 11 11 11 C.X..X..X.
..X...X...X...X...X...X...1   1   1   1   1   1 1 11 11 11 11 11 1C..X..X..X
...X...X...X...X...X...X.. 1   1   1   1   1   1 11 11 11 11 11 11:X..X..X..
                                    :
.X...X...X...X...X...X...X   1   1   1   1   1   1   1   1 1 11 11:X..X..X..
```

```
..X...X...X...X...X...X...1   1   1   1   1   1   1   1  1 11 11 C.X..X..X.
...X...X...X...X...X...X.. 1   1   1   1   1   1   1   1   11 11 1C..X..X..X
X...X...X...X...X...X...X.  1   1   1   1   1   1   1   1  1 11 11:X..X..X..
.X...X...X...X...X...X...X   1   1   1   1   1   1   1   1  11 11 C.X..X..X.
..X...X...X...X...X...X...1   1   1   1   1   1   1   1   1 1 11 1C..X..X..X
...X...X...X...X...X...X.. 1   1   1   1   1   1   1   1   1 11 11:X..X..X..
X...X...X...X...X...X...X.  1   1   1   1   1   1   1   1   11 11 C.X..X..X.
.X...X...X...X...X...X...X   1   1   1   1   1   1   1   1  1 11 1C..X..X..X
..X...X...X...X...X...X...1   1   1   1   1   1   1   1   1  11 11:X..X..X..
...X...X...X...X...X...X.. 1   1   1   1   1   1   1   1   1 1 11 C.X..X..X.
X...X...X...X...X...X...X.  1   1   1   1   1   1   1   1   1 11 1C..X..X..X
.X...X...X...X...X...X...X   1   1   1   1   1   1   1   1   11 11:X..X..X..
..X...X...X...X...X...X...1   1   1   1   1   1   1   1   1  1 11 C.X..X..X.
...X...X...X...X...X...X.. 1   1   1   1   1   1   1   1   1  11 1C..X..X..X
X...X...X...X...X...X...X.  1   1   1   1   1   1   1   1   1 1 11:X..X..X..
.X...X...X...X...X...X...X   1   1   1   1   1   1   1   1   1 11 C.X..X..X.
..X...X...X...X...X...X...1   1   1   1   1   1   1   1   1   11 1C..X..X..X
```

## Input {1000} Initial State {100}

All cars can leave the system at there current flow rate.

## Input {1000} Initial State {1000}

All cars can leave the system at there current flow rate.

## Experiment 3

This experiment simulates a lane of carriageway which can be blocked using a signal. This setup could be used to simulate any type of temporary blockage, eg. a broken down car, traffic signals, etc. It could also be used to restrict the flow of traffic, as in the above experiment. The lane blockage cell is called block. The signal line to the block cell is driven with various patterns. The patterns chosen are (0 = blocked); {00000000001111111111} traffic signal, {01} simulation of a slow cell; {000001111111111} traffic signal with high duty cycle.

## Results: Signal {00000000001111111111}

## Input {0} Initial State {1}

This result shows the outflow from a carriageway containing stationary cars and no input.

```
.............................11111111111111111111111111111111111111111 X........
.............................11111111111111111111111111111111111111111 1.X.......
.............................11111111111111111111111111111111111111111 1 X.X......
.............................11111111111111111111111111111111111111111 1 1.X.X.....
.............................11111111111111111111111111111111111111111 1 1 X.X.X....
.............................11111111111111111111111111111111111111111 1 1 1.X.X.X...
.............................11111111111111111111111111111111111111111 1 1 1 X.X.X.X..
```

```
.............................11111111111111111111111111111111 1 1 1 1.X.X.X.X.
.............................11111111111111111111111111111111 1 1 1 1 X.X.X.X.X
.............................11111111111111111111111111111111 1 1 1 1 1.X.X.X.X.
.............................11111111111111111111111111111111 1 1 1 1 11..X.X.X.X
.............................11111111111111111111111111111111 1 1 1 1 111...X.X.X.
.............................11111111111111111111111111111111 1 1 1 1 1111....X.X.X
.............................11111111111111111111111111111111 1 1 1 1 11111.....X.X.
.............................11111111111111111111111111111111 1 1 1 1 111111......X.X
.............................11111111111111111111111111111111 1 1 1 1 1111111.......X.
.............................11111111111111111111111111111111 1 1 1 1 11111111........X
.............................11111111111111111111111111111111 1 1 1 1 111111111.........
.............................11111111111111111111111111111111 1 1 1 1 1111111111.........
.............................11111111111111111111111111111111 1 1 1 1 11111111111.........
.............................11111111111111111111111111111111 1 1 1 1 11111111111 X........
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1.X.......
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 X.X......
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 1.X.X.....
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 1 X.X.X....
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 1 1.X.X.X...
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 1 1 X.X.X.X..
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 1 1 1.X.X.X.X.
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 1 1 1 X.X.X.X.X
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 1 1 1 1.X.X.X.X.
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 1 1 1 11..X.X.X.X
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 1 1 1 111...X.X.X.
.............................11111111111111111111111111111111 1 1 1 1 11111111111 1 1 1 1 1111....X.X.X
```

## Input {0} Initial State {10}

This result shows the outflow from a carriageway containing the maximum number of moving cars and no input.

```
.............................. 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.........
..............................  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X........
..............................   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.......
..............................    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X......
..............................     1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.....
..............................      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X....
..............................       1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X...
..............................        1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X..
..............................         1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
..............................          1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
..............................           1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
..............................            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11..X.X.X.X
..............................             1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111...X.X.X.
..............................              1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1111....X.X.X
..............................               1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11111.....X.X.
```

```
.............................          1 1 1 1 1 1 1 1 111111......X.X
.............................           1 1 1 1 1 1 1 1 1111111.......X.
.............................            1 1 1 1 1 1 1 11111111........X
.............................             1 1 1 1 1 1 111111111.........
.............................              1 1 1 1 1 1111111111.........
.............................               1 1 1 1 1111111111 X........
.............................                1 1 1 1111111111 1.X.......
.............................                 1 1 1111111111 1 X.X......
.............................                  1 1111111111 1 1.X.X.....
.............................                   1111111111 1 1 X.X.X....
.............................                   111111111 1 1 1.X.X.X...
.............................                   11111111 1 1 1 X.X.X.X..
.............................                   1111111 1 1 1 1.X.X.X.X.
.............................                   111111 1 1 1 1 X.X.X.X.X
.............................                   11111 1 1 1 1 1.X.X.X.X.
.............................                   1111 1 1 1 1 11..X.X.X.X
.............................                   111 1 1 1 1 111...X.X.X.
.............................                   11 1 1 1 1 1111....X.X.X
.............................                   1 1 1 1 1 11111.....X.X.
.............................                    1 1 1 1 111111......X.X
.............................                     1 1 1 1111111.......X.
.............................                      1 1 11111111........X
.............................                       1 111111111.........
```

## Input {0} Initial State {100}

```
.............................        1  1  1  1  1  1  1  1  1  1  1..X..X..X
.............................         1  1  1  1  1  1  1  1  1  1  X..X..X..
.............................          1  1  1  1  1  1  1  1  1  1 .X..X..X.
.............................           1  1  1  1  1  1  1  1  1  1..X..X..X
.............................            1  1  1  1  1  1  1  1  1 1...X..X..
.............................             1  1  1  1  1  1  1  1  11....X..X.
.............................              1  1  1  1  1  1  1  1 11.....X..X
.............................               1  1  1  1  1  1  1  111......X..
.............................                1  1  1  1  1  1  1 111.......X.
.............................                 1  1  1  1  1  1  1111........X
.............................                  1  1  1  1  1  1 1111.........
.............................                   1  1  1  1  1  11111.........
.............................                    1  1  1  1  1 1111 X........
.............................                     1  1  1  1  1111 1.X.......
.............................                      1  1  1  1 111 1 X.X......
.............................                       1  1  1  111 1 1.X.X.....
.............................                        1  1  1 11 1 1 X.X.X....
.............................                         1  1  11 1 1 1.X.X.X...
.............................                          1  1 1 1 1 1 X.X.X.X..
.............................                           1  1 1 1 1 1.X.X.X.X.
.............................                            1  1 1 1 1 X.X.X.X.X
.............................                             1  1 1 1 1.X.X.X.X.
```

```
.............................                          1  1 1 11..X.X.X.X
.............................                           1  1 111...X.X.X.
```

## Input {0} Initial State {1000}

```
.............................       1   1   1   1   1   1   1   1   X...X....
.............................        1   1   1   1   1   1   1   1  .X...X...
.............................         1   1   1   1   1   1   1   1 ..X...X..
.............................          1   1   1   1   1   1   1   1...X...X.
.............................           1   1   1   1   1   1   1  1....X...X
.............................            1   1   1   1   1   1   1 1.....X...
.............................             1   1   1   1   1   1   11......X..
.............................              1   1   1   1   1   1  11.......X.
.............................               1   1   1   1   1   1 11........X
.............................                1   1   1   1   1   111.........
.............................                 1   1   1   1   1  111.........
.............................                  1   1   1   1   1 111.........
.............................                   1   1   1   1   1111.........
.............................                    1   1   1   1  111 X........
.............................                     1   1   1   1 11 1.X.......
.............................                      1   1   1   11 1 X.X......
.............................                       1   1   1  1 1 1.X.X.....
.............................                        1   1   1  1 1 X.X.X....
.............................                         1   1   1  1 1.X.X.X...
.............................                          1   1   1  1 X.X.X.X..
.............................                           1   1   1  1.X.X.X.X.
.............................                            1   1   1  X.X.X.X.X
.............................                             1   1   1 .X.X.X.X.
.............................                              1   1   1..X.X.X.X
.............................                               1   1  1...X.X.X.
```

## Input {10} Initial State {0}

Tail backs keep travelling back up road.

```
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11..X.X.X.X
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111...X.X.X.
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1111....X.X.X
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11111.....X.X.
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111111......X.X
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1111111.......X.
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11111111........X
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111111111.........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1111111111.........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1111111111 X........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1111111111 1.X.......
```

```
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1111111111 1 X.X......
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1111111111 1 1.X.X.....
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1111111111 1 1 X.X.X....
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1111111111 1 1 1.X.X.X...
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1111111111 1 1 1 X.X.X.X..
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1111111111 1 1 1 1.X.X.X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1111111111 1 1 1 1 X.X.X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1111111111 1 1 1 1 1.X.X.X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1111111111 1 1 1 1 11..X.X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1111111111 1 1 1 1 111...X.X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1111111111 1 1 1 1 1111....X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1111111111 1 1 1 1 11111.....X.X.
```

## Input {10} Initial State {1}

Tail backs keep travelling back up road.

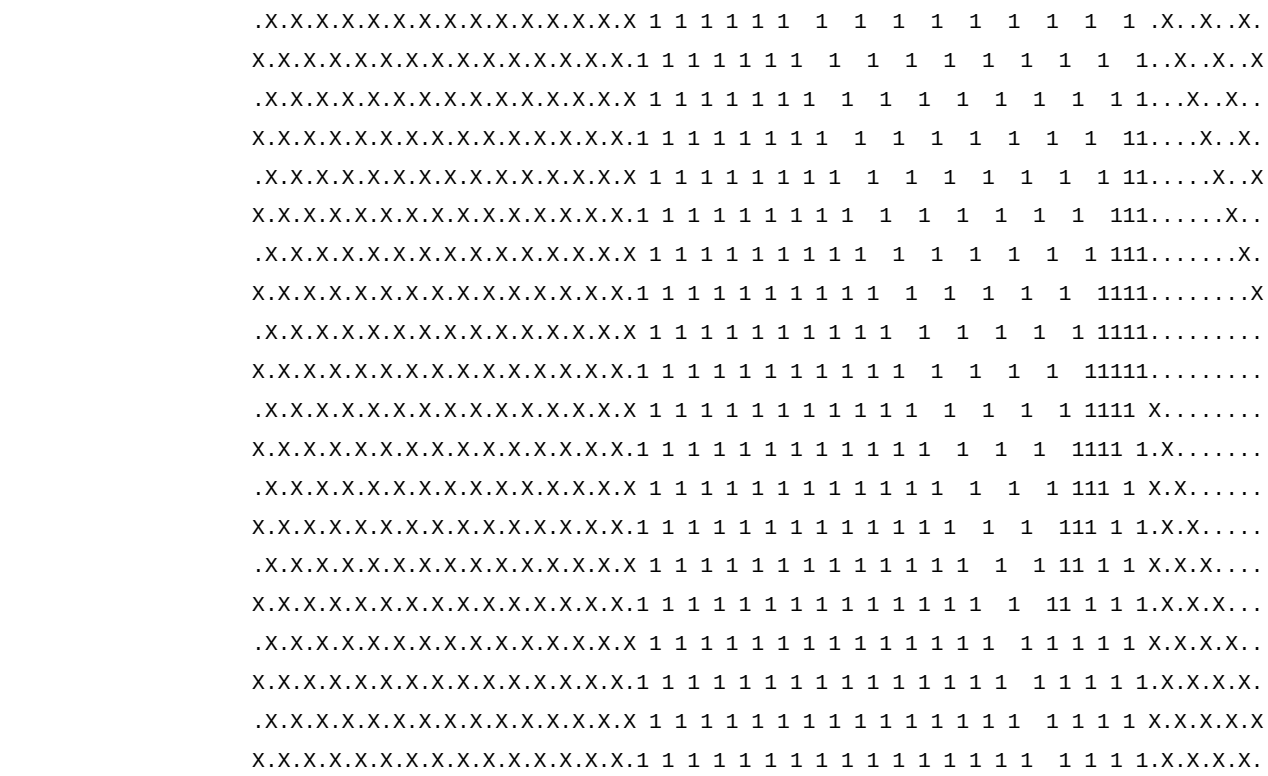
```
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X1111111111111111111111111111111111111111 X........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.XX1111111111111111111111111111111111111111 1.X.......
.X.X.X.X.X.X.X.X.X.X.X.X.X.XXX1111111111111111111111111111111111111111 1 X.X......
X.X.X.X.X.X.X.X.X.X.X.X.X.XXXX1111111111111111111111111111111111111111 1 1.X.X.....
.X.X.X.X.X.X.X.X.X.X.X.X.XXXXX1111111111111111111111111111111111111111 1 1 X.X.X....
X.X.X.X.X.X.X.X.X.X.X.X.XXXXXX1111111111111111111111111111111111111111 1 1 1.X.X.X...
.X.X.X.X.X.X.X.X.X.X.X.XXXXXXX1111111111111111111111111111111111111111 1 1 1 X.X.X.X..
X.X.X.X.X.X.X.X.X.X.X.XXXXXXXX1111111111111111111111111111111111111111 1 1 1 1.X.X.X.X.
.X.X.X.X.X.X.X.X.X.X.XXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 X.X.X.X.X
X.X.X.X.X.X.X.X.X.X.XXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1.X.X.X.X.
.X.X.X.X.X.X.X.X.X.X.XXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11..X.X.X.X
X.X.X.X.X.X.X.X.X.X.XXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 111...X.X.X.
.X.X.X.X.X.X.X.X.X.X.XXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1111....X.X.X
X.X.X.X.X.X.X.X.X.X.XXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111.....X.X.
.X.X.X.X.X.X.X.X.X.XXXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 111111......X.X
X.X.X.X.X.X.X.X.X.XXXXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1111111.......X.
.X.X.X.X.X.X.X.X.XXXXXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111111........X
X.X.X.X.X.X.X.X.XXXXXXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 111111111.........
.X.X.X.X.X.X.X.XXXXXXXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1111111111.........
X.X.X.X.X.X.X.XXXXXXXXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111111111.........
.X.X.X.X.X.X.XXXXXXXXXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111111111 X........
X.X.X.X.X.X.XXXXXXXXXXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111111111 1.X.......
.X.X.X.X.X.XXXXXXXXXXXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111111111 1 X.X......
X.X.X.X.X.XXXXXXXXXXXXXXXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111111111 1 1.X.X.....
```

## Input {10} Initial State {10}

As for Input {10} Initial State {0}

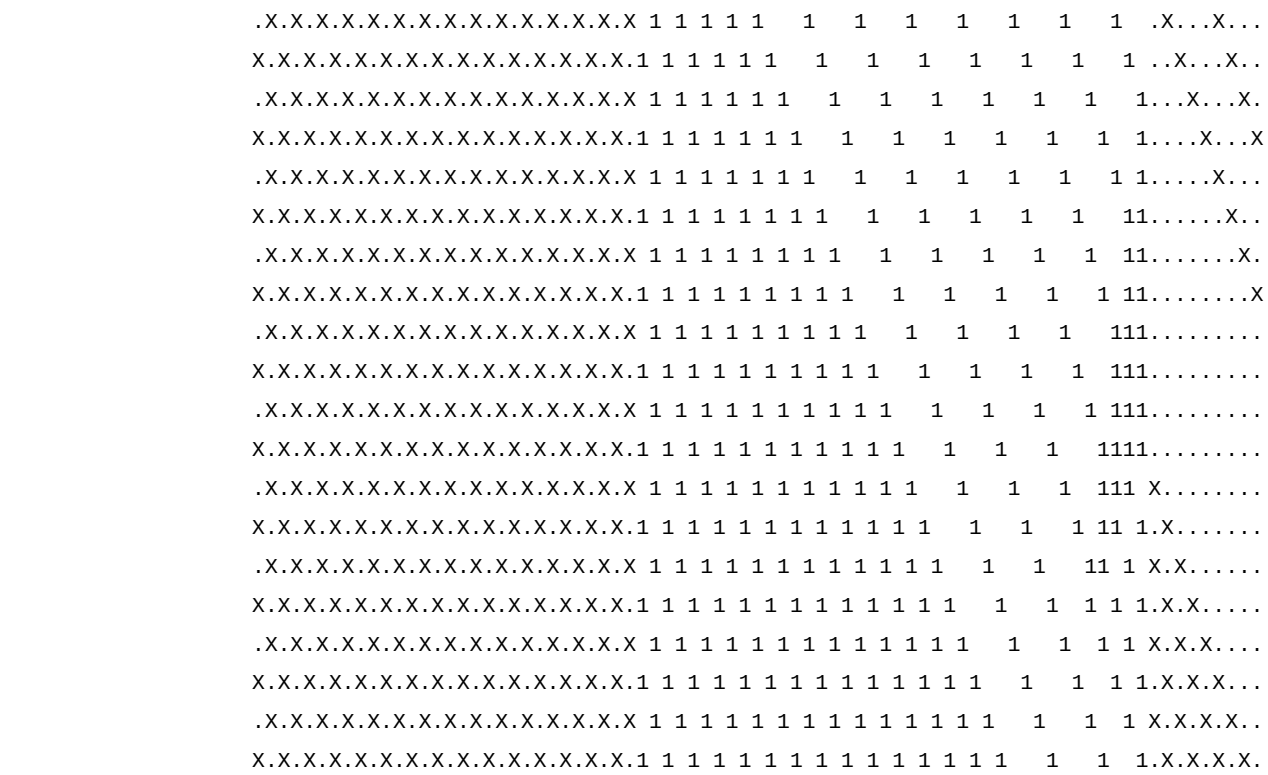
## Input {10} Initial State {100}

```
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1  1  1  1  1  1  1  1  1  1..X..X..X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1  1  1  1  1  1  1  1  1  1  X..X..X..
```



Then as for Input {10} Initial State {10}

## Input {10} Initial State {1000}



```
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1   1  X.X.X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1   1 .X.X.X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1   1..X.X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1  1...X.X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   1 1....X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   11.....X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  11......X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11.......X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111........X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1111.........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11111.........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111111.........
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111111 X........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111111 1.X.......
.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111111 1 X.X......
```

Then as for Input {10} Initial State {10}

## Input {100} Initial State {0}

```
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  1  1  1 .X..X..X.
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1  1  1  1..X..X..X
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  1  1  1 1...X..X..
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  1  1  11....X..X.
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1  1  1 11.....X..X
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  1  1  111......X..
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  1  1 111.......X.
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1  1  1111........X
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  1  1 1111.........
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  1  11111.........
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1  1 1111 X........
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  1  1111 1.X.......
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  1 111 1 X.X......
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1  111 1 1.X.X.....
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  1 11 1 1 X.X.X....
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  11 1 1 1.X.X.X...
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1 1 1 1 1 X.X.X.X..
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  1 1 1 1 1.X.X.X.X.
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  1 1 1 1 X.X.X.X.X
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1  1 1 1 1.X.X.X.X.
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  1  1 1 11..X.X.X.X
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  1  1 111...X.X.X.
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1  1  1111....X.X.X
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  1  1 1111.....X.X.
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  1  11111......X.X
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1  1 11111.......X.
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  1  111111........X
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  1 111111.........
```

```
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1111111.........
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  1 1111111.........
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1  1111111 X........
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1 111111 1.X.......
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  111111 1 X.X......
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1 11111 1 1.X.X.....
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  11111 1 1 X.X.X....
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1 1111 1 1 1.X.X.X...
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1111 1 1 1 X.X.X.X..
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1 111 1 1 1 1.X.X.X.X.
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  111 1 1 1 1 X.X.X.X.X
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1 11 1 1 1 1 1.X.X.X.X.
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  11 1 1 1 1 11..X.X.X.X
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1 1 1 1 1 1 111...X.X.X.
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1 1 1 1 1 1111....X.X.X
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1 1 1 1 11111.....X.X.
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1 1 1 111111......X.X
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1 1 1111111.......X.
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1 11111111........X
.X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1  1  1  111111111.........
..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1  1  1 111111111.........
X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1  1  1  1111111111.........
```

## Input {100} Initial State {1}

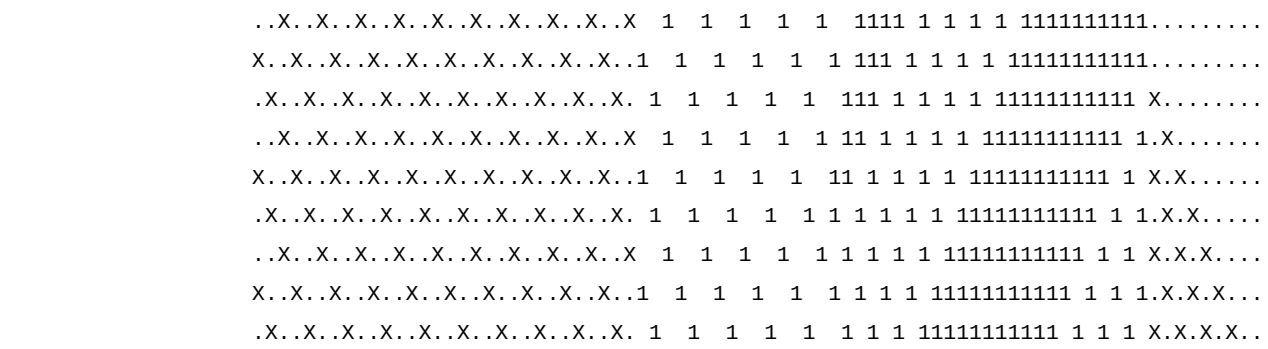
Left hand tail back gets shorter.

```
X..X..X..X..X..X..X..X..X..X11111111111111111111111111111111111111111 X........
.X..X..X..X..X..X..X..X..X..XX1111111111111111111111111111111111111111 1.X.......
..X..X..X..X..X..X..X..X..X.XX1111111111111111111111111111111111111111 1 X.X......
X..X..X..X..X..X..X..X..X..XXX1111111111111111111111111111111111111111 1 1.X.X.....
.X..X..X..X..X..X..X..X..X.XXX1111111111111111111111111111111111111111 1 1 X.X.X....
..X..X..X..X..X..X..X..X..XXXX1111111111111111111111111111111111111111 1 1 1.X.X.X...
X..X..X..X..X..X..X..X..X.XXXX1111111111111111111111111111111111111111 1 1 1 X.X.X.X..
.X..X..X..X..X..X..X..X..XXXXX1111111111111111111111111111111111111111 1 1 1 1.X.X.X.X.
..X..X..X..X..X..X..X..X.XXXXX1111111111111111111111111111111111111111 1 1 1 1 X.X.X.X.X
X..X..X..X..X..X..X..X..XXXXXX1111111111111111111111111111111111111111 1 1 1 1 1.X.X.X.X.
.X..X..X..X..X..X..X..X.XXXXXX1111111111111111111111111111111111111111 1 1 1 1 11..X.X.X.X
..X..X..X..X..X..X..X..XXXXXXX1111111111111111111111111111111111111111 1 1 1 1 111...X.X.X.
X..X..X..X..X..X..X..X.XXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1111....X.X.X
.X..X..X..X..X..X..X..XXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111.....X.X.
..X..X..X..X..X..X..X.XXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 111111......X.X
X..X..X..X..X..X..X..XXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1111111.......X.
.X..X..X..X..X..X..X.XXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111111........X
..X..X..X..X..X..X..XXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 111111111.........
X..X..X..X..X..X..X.XXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 1111111111.........
.X..X..X..X..X..X..XXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111111111.........
..X..X..X..X..X..X.XXXXXXXXXXX1111111111111111111111111111111111111111 1 1 1 1 11111111111 X........
```

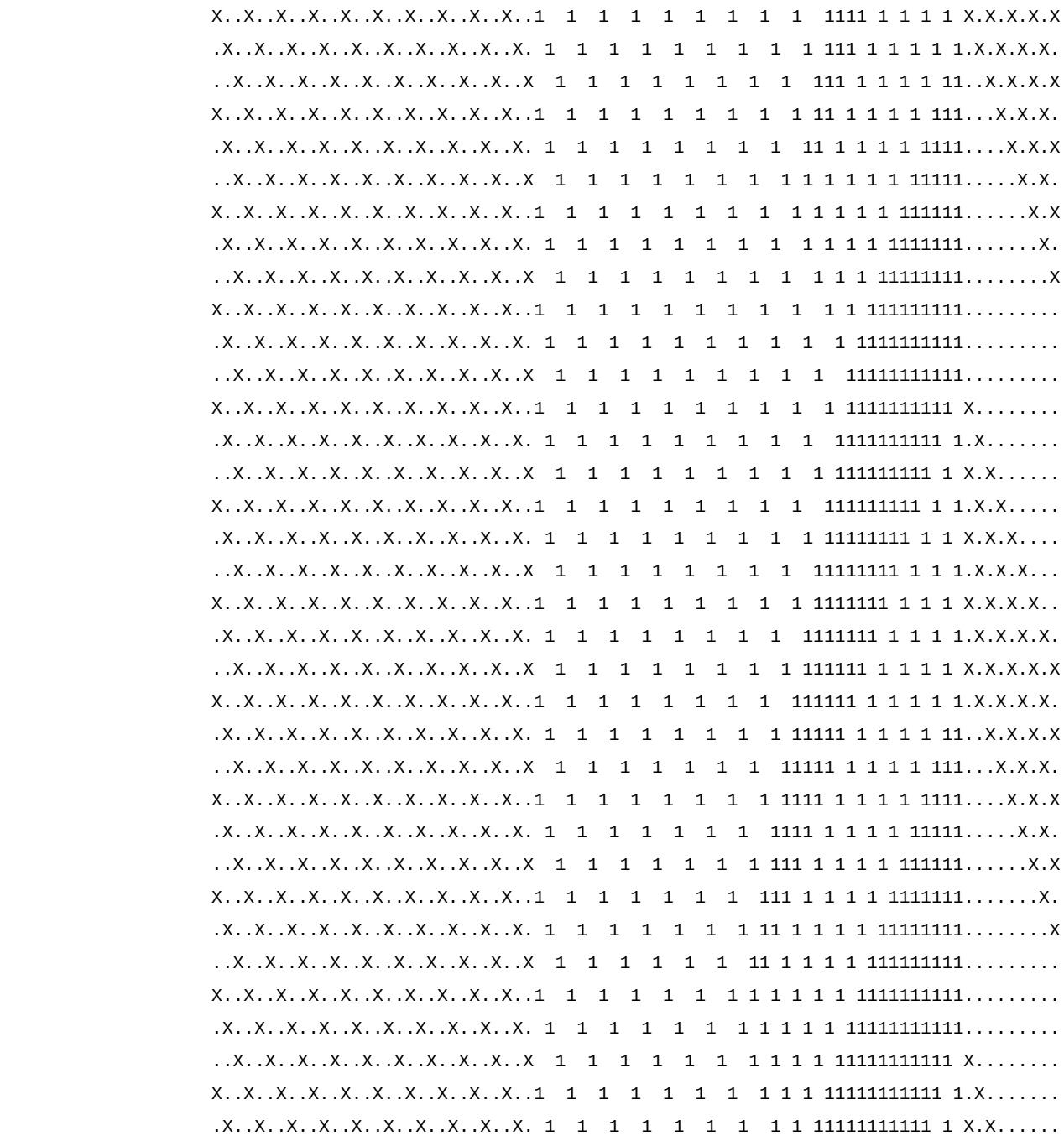
```
X..X..X..X..X..X..XXXXXXXXXXXX11111111111111111 1 1 1 1 11111111111 1.X.......
.X..X..X..X..X..X.XXXXXXXXXXXX11111111111111111 1 1 1 1 11111111111 1 X.X......
..X..X..X..X..X..XXXXXXXXXXXXX11111111111111111 1 1 1 1 11111111111 1 1.X.X.....
X..X..X..X..X..X.XXXXXXXXXXXXX11111111111111111 1 1 1 1 11111111111 1 1 X.X.X....
.X..X..X..X..X..XXXXXXXXXXXXXX11111111111111111 1 1 1 1 11111111111 1 1 1.X.X.X...
```

## Input {100} Initial State {10}

```
X..X..X..X..X..X..X..X..X..X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.........
.X..X..X..X..X..X..X..X..X..X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X........
..X..X..X..X..X..X..X..X..X..X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.......
X..X..X..X..X..X..X..X..X..X..1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X......
.X..X..X..X..X..X..X..X..X..X. 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.....
..X..X..X..X..X..X..X..X..X..X  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X....
X..X..X..X..X..X..X..X..X..X..1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X...
.X..X..X..X..X..X..X..X..X..X. 1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X..
..X..X..X..X..X..X..X..X..X..X  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
X..X..X..X..X..X..X..X..X..X..1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 X.X.X.X.X
.X..X..X..X..X..X..X..X..X..X. 1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1.X.X.X.X.
..X..X..X..X..X..X..X..X..X..X  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11..X.X.X.X
X..X..X..X..X..X..X..X..X..X..1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 111...X.X.X.
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1111....X.X.X
..X..X..X..X..X..X..X..X..X..X  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 11111.....X.X.
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 111111......X.X
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 1111111.......X.
..X..X..X..X..X..X..X..X..X..X  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1 11111111.......X
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 111111111.........
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1111111111.........
..X..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1 1111111111 X.......
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1111111111 1.X.......
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1111111111 1 X.X......
..X..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1 1111111111 1 1.X.X.....
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1111111111 1 1 X.X.X....
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  1 1 1 1 1 1 1 1 1 1 1111111111 1 1 1.X.X.X...
..X..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1  1111111111 1 1 1 X.X.X.X..
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  1 111111111 1 1 1 1.X.X.X.X.
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1  111111111 1 1 1 1 X.X.X.X.X
..X..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1 11111111 1 1 1 1 1.X.X.X.X.
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1  11111111 1 1 1 1 11..X.X.X.X
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  1 1111111 1 1 1 1 111...X.X.X.
..X..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1  1111111 1 1 1 1 1111....X.X.X
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  1 111111 1 1 1 1 11111.....X.X.
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1  111111 1 1 1 1 111111......X.X
..X..X..X..X..X..X..X..X..X..X  1  1  1  1  1  1 11111 1 1 1 1 1111111.......X.
X..X..X..X..X..X..X..X..X..X..1  1  1  1  1  1  11111 1 1 1 1 11111111.......X
.X..X..X..X..X..X..X..X..X..X. 1  1  1  1  1  1 1111 1 1 1 1 111111111.........
```



## Input {100} Initial State {100}

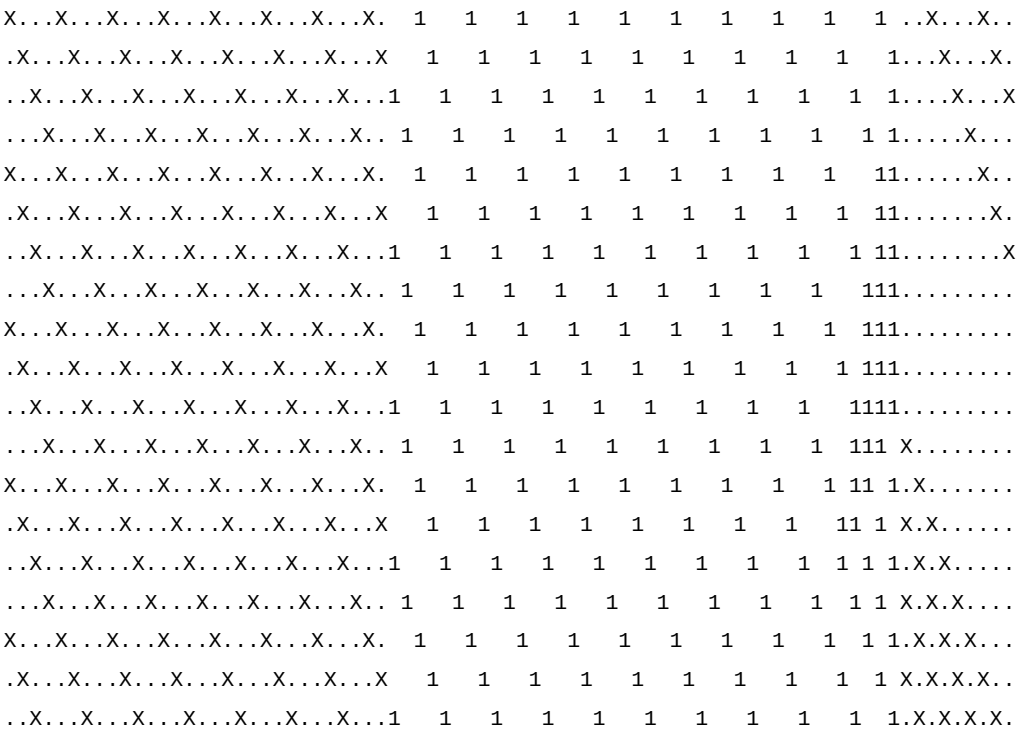


Tail backs working left.

## Input {100} Initial State {1000}

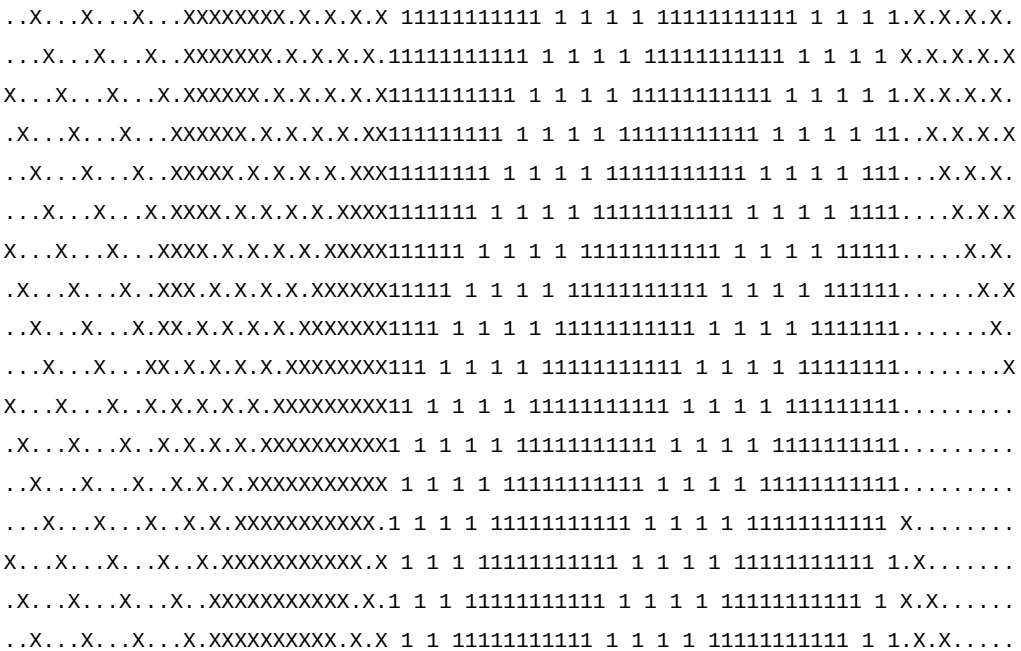
As above but slower.

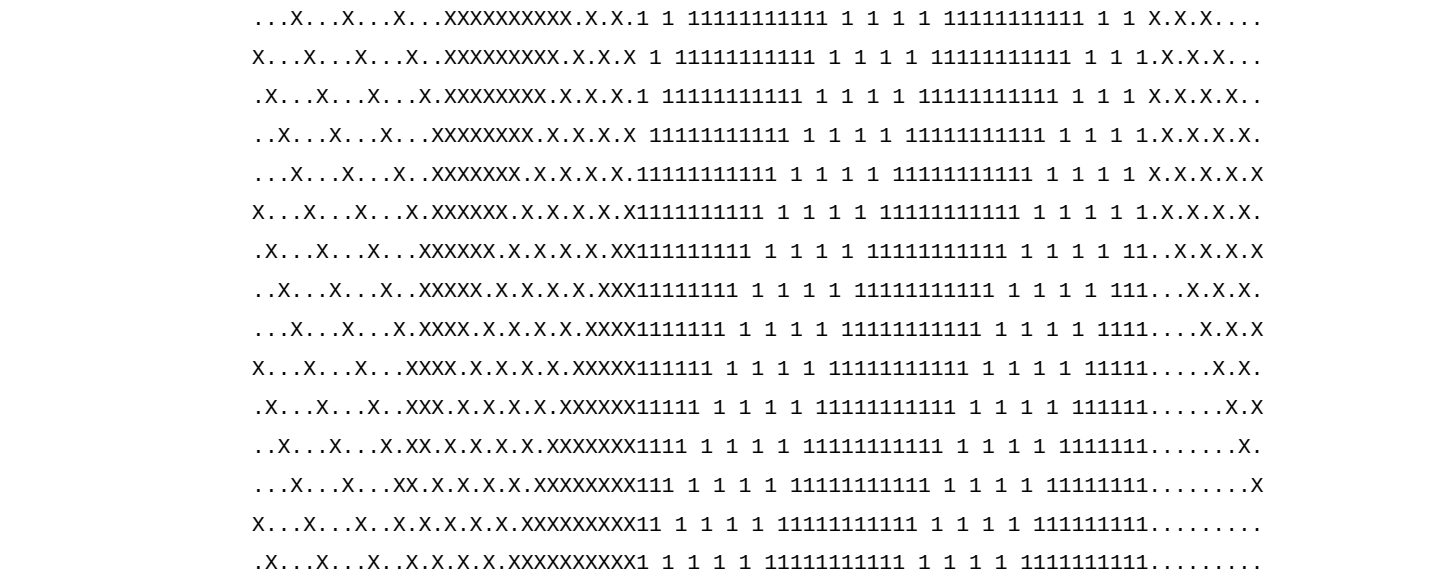
## Input {1000} Initial State {0}



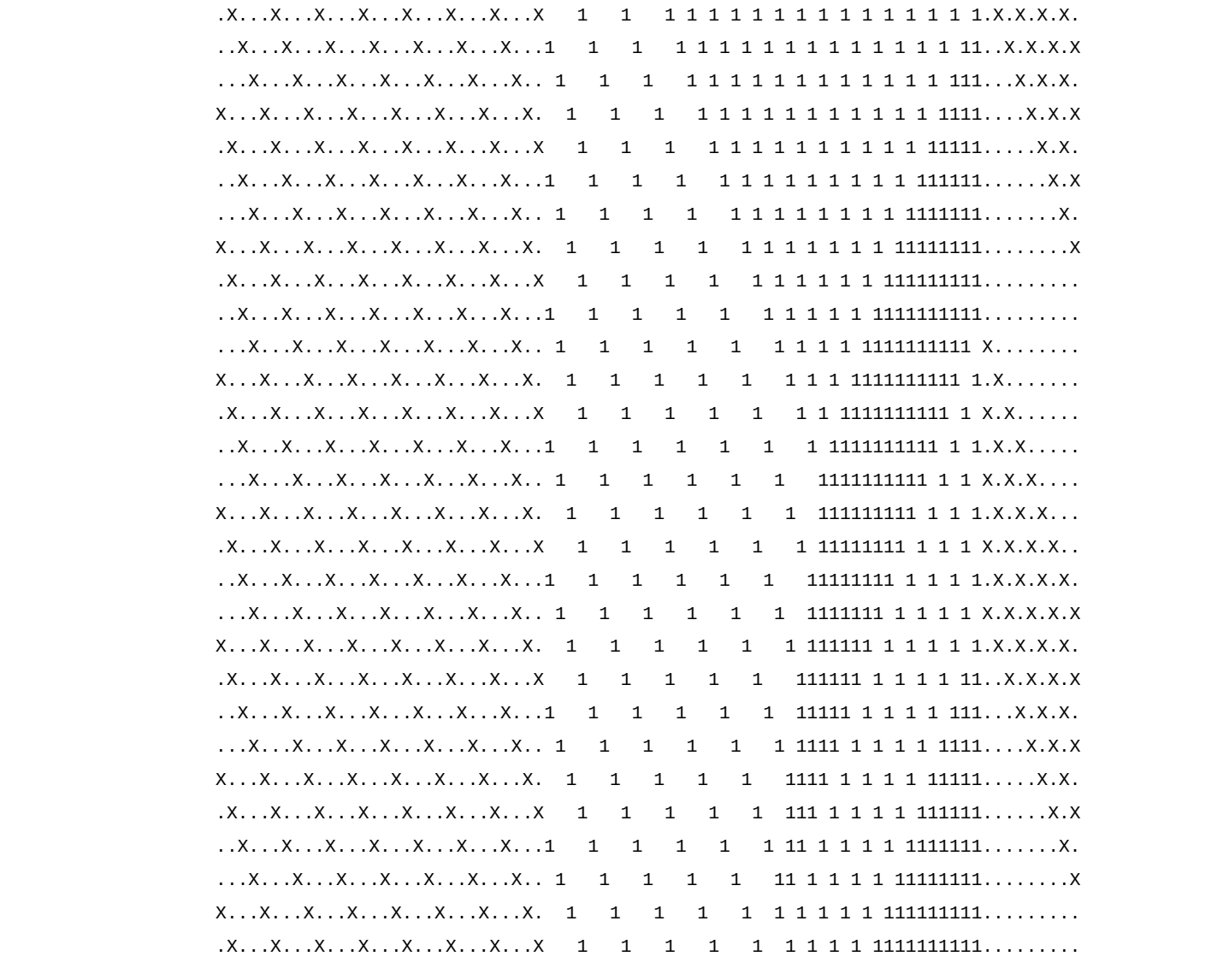
## Input {1000} Initial State {1}

As for Input {100} Initial State {1} except left tail back dissolves faster





## Input {1000} Initial State {10}



```
..X...X...X...X...X...X...1   1   1   1   1   1  1 1 1 11111111111.........
...X...X...X...X...X...X...X.. 1   1   1   1   1   1  1 1 11111111111 X........
X...X...X...X...X...X...X...X.  1   1   1   1   1   1  1 11111111111 1.X.......
```

## Input {1000} Initial State {100}

As Input {100} Initial State {0} then as Input {1000} Initial state {0}

## Results: Signal {000001111111111}

The results for this signal pattern were almost identical to those for Signal {00000000001111111111}. The main difference being that the tail backs were longer and therefore took longer to dissolve.

## Results: Signal {01}

The results for this test were similar to those of experiment 1. The only difference being if the first car arrived when the signal = blocked a two car tail back would travel back up the road as the cars re-aligned (only happens for maximum flow traffic).

```
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 X........
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1.X.......
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 X.X......
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1.X.X.....
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 X.X.X....
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1.X.X.X...
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 X.X.X.X..
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1.X.X.X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 X.X.X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1.X.X.X.X.
.X.X.X.X.X.X.X.X.X.X.X.X.X.X.X 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1 X.X.X.X.X
X.X.X.X.X.X.X.X.X.X.X.X.X.X.X.1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1 1.X.X.X.X.
```

## Experiment 4

This experiment simulates a lane of carriageway with two speed restriction. The first ‘slow’ cell slowed the traffic flow to the maximum that the second could take without causing a tail back. For experiments where the initial conditions caused a tail back between the two slow cells, and the outflow from the first slow cell was maximal, A stationary tail back would exist back from the second slow cell.

## Experiment 5

This experiment simulates a lane of carriageway with two speed restriction. The results from this experiment were very similar to those for experiment 3. The was very little interaction between the two ‘block’ cells as they both had the same net flow capacity.

# Appendix 3: Circuit Diagrams of the Cryptography System

This appendix to be the paper copies of the circuits from SPACE 2 schematic capture.