Asynchronous Logic Automata

by

David Allen Dalrymple

B.S. Mathematics, University of Maryland Baltimore County (2005)
B.S. Computer Science, University of Maryland Baltimore County (2005)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science in Media Technology

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

© Massachusetts Institute of Technology 2008. All rights reserved.

Author_____

Program in Media Arts and Sciences
May 9, 2008

Certified by_____

Neil A. Gershenfeld
Director, Center for Bits and Atoms
Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by_____

Prof. Deb Roy
Chair, Program in Media Arts and Sciences

Asynchronous Logic Automata

by

David Allen Dalrymple

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on May 9, 2008, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Technology

Abstract

Numerous applications, from high-performance scientific computing to large, high-resolution multi-touch interfaces to strong artifical intelligence, push the practical physical limits of modern computers. Typical computers attempt to hide the physics as much as possible, running software composed of a series of instructions drawn from an arbitrary set to be executed upon data that can be accessed uniformly. However, we submit that by exposing, rather than hiding, the density and velocity of information and the spatially concurrent, asynchronous nature of logic, scaling down in size and up in complexity becomes significantly easier. In particular, we introduce "asynchronous logic automata", which are a specialization of both asynchronous cellular automata and Petri nets, and include Boolean logic primitives in each cell. We also show some example algorithms, means to create circuits, potential hardware implementations, and comparisons to similar models in past practice.

Thesis Supervisor: Neil A. Gershenfeld
Title: Director, Center for Bits and Atoms, Professor of Media Arts and Sciences

Asynchronous Logic Automata

by

David Allen Dalrymple

The following people served as readers for this thesis:

Thesis Reader_____

Marvin Lee Minsky
Professor of Media Arts and Sciences
MIT Media Laboratory

Thesis Reader_____

Gerald Jay Sussman
Panasonic Professor of Electrical Engineering
MIT Computer Science and Artificial Intelligence Laboratory

Acknowledgments

I am grateful for the support of MIT's Center for Bits and Atoms and its sponsors.

I would like to thank my thesis supervisor, Neil Gershenfeld, whose suggested reductions to practice and ready-fire-aim methodology helped me move my ideas beyond vague speculation. Without his initiative in bringing me to MIT, I would not be here today.

I also thank my thesis committee: Gerald Sussman, who helped limit my goals for this Master's thesis, Marvin Minsky, who provided immensely valuable oral history, and unofficially, Ray Kurzweil, who taught me what all this computing will be good for in the future, and Tom Toffoli, who explained to me what computing really means.

I thank those who have worked with me on the Conformal Computing team and discussed these ideas with me, including Kailiang Chen, Kenny Cheung, Ahana Ghosh, Forrest Green, Mike Hennebry, Mariam Hoseini, Scott Kirkpatrick, Ara Knaian, Luis Lafeunte Molinero, Ivan Lima, Mark Pavicic, Danielle Thompson, and Chao You.

I wish to thank also these additional people who have taught me over the years: Andrew Alter, Vivian Armor, Tom Armstrong, Bianca Benincasa, Bruce Casteel, Richard Chang, Charles Choo, Donald Clark, R. Scott Cost, Marie desJardins, Robert Earickson, Sue Evans, William Feasley, Ted Foster, Karen Freiberg, Dennis Frey, Marianna Gleger, Matthias Gobbert, Muddappa Gowda, Kathleen Hoffman, Arnie Horta, Kostantinos Kalpakis, Jon Kelner, Chris Keaveney, John Kloetzel, James Lo, Greg Long, Yen-Mow Lynn, James McKusick, Susan Mitchell, Marc Olano, Pablo Parrillo, Art Pittenger, Florian Potra, Alan Sherman, Krishna Shivalingham, Michael Sipser, Verlie-Anne Skillman, Raymond Starr, Brooke Stephens, Michael Summers, Suzanne Sutton, Terry Viancour, Tad White, Terry Worchesky, En-Shinn Wu, Mohamed Younis, Tatia Zack, Robin Zerbe, Lynn Zimmerman, and John Zweck.

In addition, I thank those who have graciously given me work during my year of being an independent contractor: Greg Bean, Michael Speca, and Ron Vianu. Each was flexible when it came to my juggling hours between the three, supportive of my work, and willing to hire someone at an adult rate who was only 13–14 years old. I'd also like to thank my colleagues at these and other companies, especially Kurzweil Technologies, Inc., where I worked for three summers.

I thank all my great friends, colleagues, and administrators at MIT, including: Moin Ahmad, Jason Alonso, Kenneth Arnold, Anurag Bajpayee, Sarah Bates, Amy Brzezinski, Bill Butera, Kailiang Chen, Suelin Chen, Zhao Chen, Erik Demaine, David Farhi, Catherine Havasi, Eric Vincent Heubel, Jack Hill, Sarah Hadaway Hudson, Anika Huhn, Alicia Hunt, Nayden Kambouchev, Chris Kennedy, Siân Kleindienst, David Kopp, Dave Ta Fu Kuo, Sherry Lassiter, Steve Leibman, Dacheng Lin, Yuri Lin, Greg Little, Sandy Lugo, Kerry Lynn, Ian MacKenzie, Yael Maguire, William Michael Kaminsky, Pranav Mistry, Manas

Mittal, Bo Morgan, Ilan Moyer, Susan Murphy-Bottari, Ann Orlando, Terry Orlando, Neri Oxman, Linda Peterson, Emily Pittore, George Alex Popescu, Manu Prakash, Michael Price, Katya Radul, Mitch Resnick, Aaron Rosado, Stephen Samouhos, Beth Schaffer, Laura Schuhrke, Adam Seering, Gigi Shafer, Nur Shahir, Mikey Siegel, Jay Silver, Dustin Smith, Amy Sun, Tina Tallon, Christian Ternus, Naoto Tsujii, Erica Ueda, Noah Vawter, Dheera Venkatraman, Alexandra Vinegar, Eric Weese, Yang Zhang (and many others who cannot be listed on this page), as well as those who have supported me from elsewhere, including Jack Aboutboul, Amy Adler, Orion Anderson, Catherine Asaro, Mike Atamas, Daniel Bankman, Nic Babb, Roni Bat-Lavi, Noah Bedford, Tori Bedford, James Bell, Alex Bishop, Bryan Bishop, Richard Blankman, Max Boddy, Rebecca Rose Boddy, Tori Borland, Cathy Cannizzo, Bob Davidson, Jan Davidson, Matthew Dupree, Santiago Echeverry, Rachel Ellison, Andrew Fenner, Amber Fenner, Brittany Frey, Irina Fuzaylova, Paul Gardner, Chrissy Gregg, Sean Griffith, Michael Hakulin, Eric Hamilton, Freeman Hrabowski, Joe Howley, Matthew Itkin, John Kloetzli, Brian Krummel, Chris Krummel, Zachary McCord, Paige McKusick, Alex Menkes, Alanna Ni, Mark Nejman, Robert Rifkin, Demetri Sampas, Sebastian Sampas, Linus Schultz, Lucy Schultz, Karineh Shahverdi, Seth Shostak, Brian Stepnitz, Joshua Stevens, Corey Stevens, Stephen Stickells, Linda Stone, Barry Tevelow, Bill Trac, Sophie Weber, Jeanette Whitney, and Richard Saul Wurman, Wan-Hsi Yuan. I'd also like to thank all the members of the Paint Branch Unitarian Universalist Church Sunday morning discussion group (led by Lowell Owens) and the staff of the Davidson Institute for Talent Development for their support.

I must thank those remaining giants upon whose shoulders I have attempted to stand: Gottfried Wilhelm Leibniz and George Boole for binary logic; Kurt Gödel, Alonzo Church, and Alan Mathison Turing for formalizing the algorithm; Stanislaw Ulam, John von Neumann, Arthur Walter Burks, Edwin Roger Banks, and John Horton Conway for cellular automata; Claude Elwood Shannon for the digital circuit *and* information theory; Norbert Weiner for cybernetics; Carl Adam Petri for the theory of nets; Yves Pomeau, Uriel Frisch, and Brosl Hasslacher for lattice gases; Rolf Landauer for the relation between reversibility, information, and entropy; William Daniel Hillis, Richard Feynman, Carl Feynman, Tom Knight, Charles Eric Leiserson, Guy Lewis Steele Jr., Stephen Wolfram, and others for the Connection Machine; Richard Matthew Stallman and Linus Torvalds for GNU/Linux; Donald Ervin Knuth and Leslie Lamport for TEX and LATEX; Bram Moolenaar for `vim`; Kenneth Eugene Iverson, Brian Wilson Kernighan, Dennis MacAlistair Ritchie, John McCarthy, Steve Russell, Guy Lewis Steele Jr. (again), Peter Landin, Robin Milner, Simon Peyton-Jones, Paul Hudak and many more, for their significant contributions to programming languages; and Douglas Hofstadter, Paul Graham, Philip Greenspun, Peter Bentley, David Thomas, and Andrew Hunt, for prose that has changed the way I think; and Nicholas Negroponte for creating the wonderful research environment of the Media Lab.

Finally, I am grateful for the love and support of all my grandparents (Pythagoras and Mildred Cutchis, Lee and Esther Dalrymple), aunts and uncles, great-aunts and great-uncles, cousins of various types, and most importantly, my parents, Athena and Scott Dalrymple, without whom I would not be.

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Background

1.1 Introduction

To build a computer is to create initial conditions and define the interpretations of inputs, outputs, and states such that the dynamics of physics, within a limited spatial domain, corresponds exactly to that of a more easily useful (yet equally versatile) mathematical model. Various modes of physics have been employed to this end, including electromechanical (relays), thermionic (vacuum tubes), and even fluidic [32] and quantum [15]. In addition, many distinct families of models have been emulated, including pointer machines (in which all information is accessed by traversing a directed graph), Harvard architectures (in which instructions are separate from data), and dataflow architectures (in which there is no explicit flow of control). However, almost all computers for the past 50–60 years have made one specific choice: namely, the so-called "von Neumann" [47] model of computation (also known as Random Access Stored Program or RASP).

Physics inherently allows only local information transfer, and computation, like every other process, relies on physics. Thus, programming models which assume non-local

processes, such as data buses, random access memory, and global clocking, must be implemented at a slow enough speed to allow local interactions to simulate the non-local effects which are assumed. Since such models do not take physical locality into account, even local effects are limited to the speed of the false non-local effects, by a global clock which regulates all operations.

In computing today, many observers agree that there is a practical physical speed limit for the venerable von Neumann model (see for instance [33]), and that the bulk of future speed increases will derive from parallelism in some form. Chipmakers are currently working to pack as many processors as they can into one box to achieve this parallelism, but in doing so, they are moving even further from the locality that is necessary for a direct implementation as physics. At the other end of the abstraction spectrum, while sequential programming models can be generalized to use multiple parallel threads, such models are often clumsy and do not reflect the physical location of the threads relative to each other or memory.

In addition, research has long suggested that asynchronous (or "self-timed") devices consume less power and dissipate less heat than typical clocked devices [48]. However, traditional microarchitectures require significant book-keeping overhead to synchronize various functional blocks, due to the nature of their instructions, which must be executed in sequence. Most asynchronous designs to present have derived their performance benefits from clever pipelining and power distribution rather than true asynchrony – known as "globally asynchronous, locally synchronous" design – and often this is not enough to offset the overhead [14].

These shortcomings are accepted because of the tremendous body of existing code written in sequential fashion, which is expected to run on the latest hardware. However, by removing the assumption of backwards compatibility, there is an opportunity to create a new, disruptive programming model which is more efficient to physically implement. The trick is to expose the underlying physical limitations formally in the

model, instead of hiding them, and to bring engineering tradeoffs traditionally fixed in advance into the dynamic and reconfigurable realm of software. Such a model could scale favorably and painlessly to an arbitrary number of parallel elements, to larger problem sizes, and to faster, smaller process technologies.

Potentially, this may have eventual impact across the field of computing, initially in:

- high-performance computing, in which parallelization is the only way forward, and sequential algorithms are not scaling favorably;
- very large, high-resolution human-computer interfaces, which not only require parallelism but have a natural sense of spatial distribution;
- physical simulations and 3D rendering, which are volumetric in nature and could take advantage of this type of model if extended to 3D;
- as well as strong AI and the Singularity (see [23]), which requires massive parallelism to emulate the myriad functions of the human brain.

1.2 Past Work

The ideas discussed so far are not original: the history begins with the cellular automata (CAs) of von Neumann [46], designed to explore the theory of self-replicating machines in a mathematical way (though never finished). Note that this was some time after he completed the architecture for the Electronic Discrete Variable Automatic Computer, or EDVAC [47], which has come to be known as "the von Neumann architecture." Many papers since then can be found examining (mostly 2-state) CAs, and there are a few directions to prove simple CA universality – Alvy Ray Smith's [38], E. Roger Banks' [5], and Matthew Cook's more recent Rule 110 construction [10]. However, while interesting from the point of view of computability theory, classical

CAs clearly over-constrain algorithms to beyond the point of practicality, except in a certain class of problems related to physical simulation [13].

Norman Margolus and Tommaso Toffoli, among others, built special-purpose hardware called the Cellular Automata Machine 8 (CAM-8) for these and other applications [44, 25]. However, although the CAM-8 was optimized to simulate cellular automata, it was not physically embodied as an extensible cellular structure. Our work is also distinguished by constraints that are closer to physics than the rigid clocking of a classical cellular automaton.

Another related sub-field is that of field-programmable gate arrays (FPGAs). Gate arrays have evolved over time from sum-product networks such as Shoup's [37] and other acyclic, memory-less structures such as Minnick's [26] to the complex, non-local constructions of today's commercial offerings, yet skipping over synchronous and sequential, but simplified local-effect cells. Because neither FPGA type is strictly local, an ensemble of small FPGAs cannot easily be combined into a larger FPGA.

The tradition of parallel programming languages, from Occam [34] to Erlang [3] to Fortress [41] is also of interest. Although they are designed for clusters of standard machines (possibly with multiple processors sharing access to a single, separate memory), they introduce work distribution techniques and programming language ideas that are likely to prove useful in the practical application of our work. However, they are still based on primitive operations such as multiplication or message passing, which are far from primitive from a physical perspective.

Paintable Computing [8], a previous project at the Media Lab, and Amorphous Computing [1], a similar project in MIT's Project on Mathematics and Computation (Project MAC), share similar goals but still used stock von Neumann devices at each node, although they assumed much less about the density or relative position of nodes in space. Regular, tightly packed lattices are clearly more spatially efficient than random diffusion, and the only cost is that the "computing material" would come in

sheets or blocks rather than paint cans or powder ("smart dust"). In addition, our work requires far less capability to exist at the lowest level of hierarchy it describes, so it is not tied to a particular processor architecture or even process technology.

Finally, the Connection Machine [19] was designed with a similar motivation – merging processing and memory into a homogeneous substrate – but as the name indicates, included many non-local connections: "In an abstract sense, the Connection Machine is a universal cellular automaton with an additional mechanism added for non-local communication. In other words, the Connection Machine hardware hides the details." We are primarily concerned with exposing the details, so that the programmer can decide on resource trade-offs dynamically. However, the implementation of Lisp on the Connection Machine [40] introduces concepts such as xectors (spatially distributed, inherently parallel sequences of data) which are likely to be useful in the implementation of functional programming languages in our architecture.

To sum up, the key element of our approach that is not present in any of these models is that of formal conformance to physics:

- classical CAs are an "overshoot" – imposing too many constraints between space and time above those of physics;
- the CAM-8 machine, while specialized to simulate CAs, is not physically an extensible cellular structure;
- gate arrays have become non-local and are trending further away from local interactions, and cannot be fused together into larger gate arrays;
- practical parallel languages accept the architecture of commercial computers and simply make the best of it in software;
- Paintable Computing and Amorphous Computing assume a diffuse set of nodes

rather than a tight space-packing lattice, as well as building up from microprocessors rather than more primitive primitives; and

- the Connection Machine allows non-local communication by hiding physical details.

Also, at least as important as this is the fact that our model operates precisely without clocking, while the models above do not. This decreases power requirements and heat dissipation, while increasing overall speed.

We now discuss at a lower level the development and specifics of the Logic CA and Asynchronous Logic Automata.

Chapter 2

Models

2.1 Inspiration

We initially sought the simplest, most lightweight models we could to enable maximum hardware flexibility. Much past work has begun with the assumption of a certain processor (e.g. ARM9, PowerPC), but in the spirit of staying close to physics, we wanted a much more conceptually minimal universal computing element. Not only would this free us from the choice of architectures and programming languages, but if the primitive element is simple enough, it can be ported across physical modes (semiconductor, fluidic, molecular, mechanical, etc.). This naturally led to an examination of cellular automata: among the simplest computationally universal models known, and local by nature. Cellular automata make only local assumptions about communications, have limited state, and thus limited logic in each node. However, most cellular automata are considered mere theoretical curiosities due to the number of such minimalistic cells it takes to implement even elementary logical functions such as AND or XOR.

In 1970, Banks published a thesis supervised by Fredkin, Minsky, Sheridan and Payn-

ter in which he proved that a two-state cellular automaton communicating only with its four nearest edge neighbors is computationally universal. The automaton is governed by three simple rules: a "0" cell surrounded by three or four "1" cells becomes a "1", a "1" cell which is neighbored on two adjacent sides (north and west, north and east, south and west, or south and east) by "1" cells and on the other sides by "0" cells becomes a "0", and finally, that any other cell retains its previous value. Universality was proven largely by the implementation of a universal logic element, $B \wedge \neg A$, shown in Fig. 2-1. Other logic elements, such as $B \wedge A$, can be constructed using this one and similarly sized routing and fan-out components exhibited in the Banks thesis. Despite the simplicity of each individual cell, to perform interesting computation, a large number of such cells is needed.

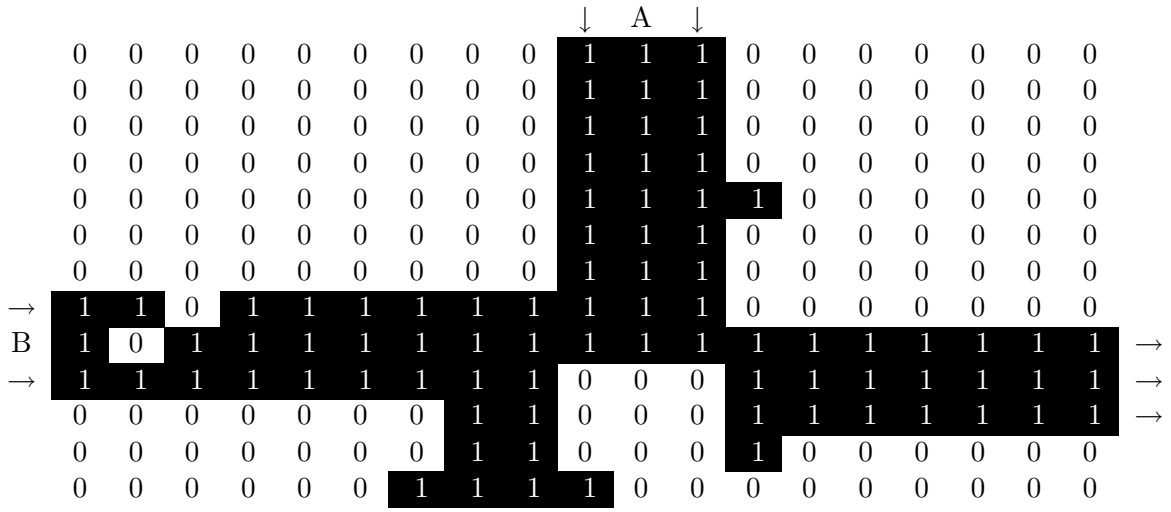


Figure 2-1: E. Roger Banks' universal logical element (computes $B \wedge \neg A$). A logical $TRUE$ is represented by a two-"0" pattern along three parallel lines of "1"s (seen here coming into input "B").

2.2 Logic CA

2.2.1 Motivation

The original model presented in this section improves the product of complexity per cell and the number of cells needed to achieve the design by incorporating Boolean logic directly in the cells instead of deriving it from ensembles of cells. For instance, although a Banks CA cell can be implemented in about 30 transistors, implementing a full adder would require over 3000 Banks cells (about 90,000 total transistors). Another way of looking at this is to say that we are trading off system complexity against design complexity, but in fact, in any given application, design complexity bounds total system complexity (enough cells are needed in the system to represent the design) so both are improved in practice. Note that this does impact the theoretical power of the model: a consequence of Turing-universality is that any universal CA can compute an equivalent class of functions given sufficient time and storage. What we are considering here is a means to decrease the specific storage requirement for a representative problem (in this case, the simple combinatorial circuit of the full adder). We considered some different ways to achieve this, and found that a relatively low complexity product is attained by a CA in which each cell is essentially a processor with one one-bit register and a set of one-bit instructions, which we call the "Logic CA". The basic concept of the Logic CA is to recognize that if making Boolean logic circuits is the goal, and if transistors (or other types of switches) are the physical substrate, then it is quite sensible to include Boolean logic functions directly in the definition of the CA. In addition, if crossovers are part of the goal, and are admissible in the substrate, then it is reasonable to allow diagonal connections between cells to cross each other. In this model, a full adder can be implemented in 6 cells, each consisting of 340 transistors, for a total of 2040 transistors, improving the complexity product by more than an order of magnitude. However, these concessions

do not only improve the numbers, but are also qualitatively beneficial for ease of design synthesis.

2.2.2 Details

The Logic CA consists of cells with 8 neighbors and 9 total bits of state. The state bits are divided into 8 configuration bits (specifying the "instruction" to be performed at every clock tick) and 1 dynamic state bit. The configuration bits are further divided into 2 gate bits which choose among the four allowed Boolean functions ($\mathcal{G} = \{AND, OR, XOR, NAND\}$) and 6 input bits which choose among the 36 possible pairs of (potentially identical) inputs chosen from the 8 neighbors ($\frac{1}{2} \cdot 8 \cdot (8-1)+8$). At each time step, a cell examines the dynamic state bit of its selected inputs, performs the selected Boolean operation on these inputs, and sets its own dynamic state to the result.

Mathematically, an instance of the Logic CA can be described as a series of global states S_t ($t \in \mathbb{N}_0$) each composed of local states $s^t_{(i,j)} \in \{0,1\}$ ($i,j \in \mathbb{Z}$) and a set of constant configuration elements

$$\begin{aligned} c_{(i,j)} \in \mathcal{C} &= (\mathcal{G} \times (\{-1,0,1\}^2 - \{(0,0)\})^2) \\ &= \mathcal{G} \times \{(1,0),(1,1),(0,1),(-1,1),(-1,0),(-1,-1),(0,-1),(1,-1)\} \\ &\quad \times \{(1,0),(1,1),(0,1),(-1,1),(-1,0),(-1,-1),(0,-1),(1,-1)\} \end{aligned}$$

(note that there is a bijection between $\mathcal{C}$ and $\{0,1\}^8$, 8 bits) such that given the definitions

$$\begin{aligned} g_{(i,j)} &= (c_{(i,j)})_1 \in \mathcal{G} && \text{(the selected gate)} \\ a_{(i,j)} &= (i,j) + (c_{(i,j)})_2 \in \mathbb{Z}^2 && \text{(the coordinates pointed to by input A)} \\ b_{(i,j)} &= (i,j) + (c_{(i,j)})_3 \in \mathbb{Z}^2 && \text{(the coordinates pointed to by input B)} \end{aligned}$$

we have the update rule:

$$s_{(i,j)}^{t+1} = \begin{cases} \text{if} \quad g_{(i,j)} = AND & s_{a_{(i,j)}}^{t} \wedge s_{b_{(i,j)}}^{t} \\ \text{if} \quad g_{(i,j)} = OR & s_{a_{(i,j)}}^{t} \vee s_{b_{(i,j)}}^{t} \\ \text{if} \quad g_{(i,j)} = XOR & s_{a_{(i,j)}}^{t} \oplus s_{b_{(i,j)}}^{t} \\ \text{if} \quad g_{(i,j)} = NAND & \neg(\; s_{a_{(i,j)}}^{t} \wedge s_{b_{(i,j)}}^{t} \;) \end{cases}$$

This description as a formal CA is cumbersome because the Logic CA sacrifices mathematical elegance for practical utility, but be assured that it represents the same concept as the paragraph of prose at the beginning of the section.

In pictures of the Logic CA such as Fig. 2-2, the smaller squares outside the large squares representing each cell indicate the selected input directions for a given cell, the central glyph indicates the selected Boolean function, and the color of the glyph indicates the dynamic state (the shade on the farthest left cell represents 0, and the shade on the farthest right represents 1).

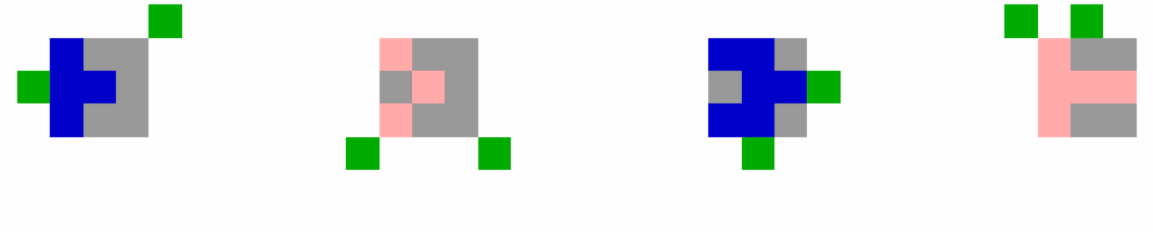


Figure 2-2: Logic CA gates (from left: AND, OR, XOR, NAND)

2.3 Asynchronous Logic Automata

2.3.1 Motivation

There are a few driving factors that push us to consider a version of the Logic CA which is not driven by a global clock (i.e. not all cells update simultaneously). The first is simply the cost of distributing the global clock without losing phase over an indefinitely sized array. Even if it may be possible to correct phase and distribute the clock locally using a phase-lock loop [17] or similar strategies, clock distribution will still cost power. More importantly, updating all cells simultaneously wastes power since some areas of the array may have more work to do than others at any given time, and ought to be updated more because of it.

By "asynchronous" here, we do not simply mean that each element might perform its designated operation at any moment, nor do we mean that there are a few clocked elements which each have an independent clock. Rather, there are strict conditions on when operations can be performed based on data dependencies (the inputs must be ready to provide input, and the outputs must be ready to receive output). These conditions cannot be reversed except by the operation, and the operation can be delayed any amount of time without causing the overall system to have a different result (the result would simply appear later). When traditional architectures are made to run without a global clock [48], the data dependencies in these architectures can be very complex, and a huge amount of design effort is required to make sure they are always resolved. The trick here that makes asynchronous design easy is that because each individual bit is a processor, and each processor is only dealing with $O(1)$ bits at any given time, the constraints necessary to make an asynchronous circuit work can be enforced by the substrate itself, below the level of Logic CA circuit design.

Another benefit of making the Logic CA asynchronous is the elimination of "delay

lines". These are parts of the circuit which exist solely to make the length of two convergent paths equal so that their signals arrive at the same time. In an asynchronous version, these become unnecessary, since the merging point will have to wait for both signals to be ready. This saves space, time, and power, and makes creating valid algorithms simpler.

Also, as we will see, the changes needed to make the Logic CA work asynchronously without explicit acknowledgment signals also make the application of charge-conserving logic possible. Traditional complementary metal-oxide-semiconductor (CMOS) logic sinks charge at every gate input and sources it again at the output, dissipating and consuming energy. Charge-conserving logic ejects the same electrons at the output as entered the input (along with some extra to restore those which left due to thermal factors). This also saves a large amount of power.

In short, power is saved by:

- not distributing a clock signal over a large physical space,
- not consuming power in areas which are idle,
- and not dissipating $\frac{1}{2}CV^2$ with every operation.

In addition, speed is improved since cells which are only passing data from input to output, and not computing, may move data at gate propagation speed, (about 1 millimeter per nanosecond) which is a couple orders of magnitude slower than light, but a few orders of magnitude faster than a synchronous Logic CA would be.

2.3.2 Details

Asynchronous Logic Automata (ALA) are modification of the Logic CA, inspired by both lattice-gas theory [43] and Petri net theory [31], that realizes the benefits

described above.

By "lattice gas", we mean a model similar to cellular automata in which the cells communicate by means of particles with velocity as opposed to broadcasted states. Practically, this means that the information transmitted by a cell to each of its neighbors is independent in a lattice gas, where in a cellular automaton these transmissions are identical. By convention, a lattice gas also has certain symmetries and conservation properties that intuitively approximate an ideal gas [18], and in some cases, numerically approximate an ideal gas [13].

Meanwhile, Petri nets are a broad and complex theory; we are primarily concerned with the subclass known as "marked graphs" (a detailed explanation can be found in [29]). In short, a marked graph is a graph whose edges can be occupied at any given time by zero or more tokens. According to certain conditions on the tokens in edges neighboring a node of the graph, the node may be allowed to "fire" (at any time as long as the conditions are met) by performing some operations on the tokens (such as moving a token from one of its edges to another or simply consuming a token from an edge). Petri nets have been used for the explicit construction of asynchronous circuits in the past [11, 28], but not in combination with a cellular structure.

Asynchronous Logic Automata merge these with the Logic CA as follows. We remove the global clock and the bit of dynamic state in each cell, and replace the neighborhood broadcasts with a set of four edges between neighboring cells, each containing zero or one tokens, thus comprising a bit of state (see Fig. 2-3). Between each pair of cells, in each direction, we have a pair of edges, one to represent a "0" signal, and the other a "1" signal. Note that each pair of edges could be considered one edge which can carry a "0" token or a "1" token. Instead of each cell being configured to read the appropriate inputs, this data is now represented by an "active" bit in each edge. Then, each cell becomes a stateless node (except that it still maintains its own gate type) in this graph, which can fire on the conditions that all its active inputs

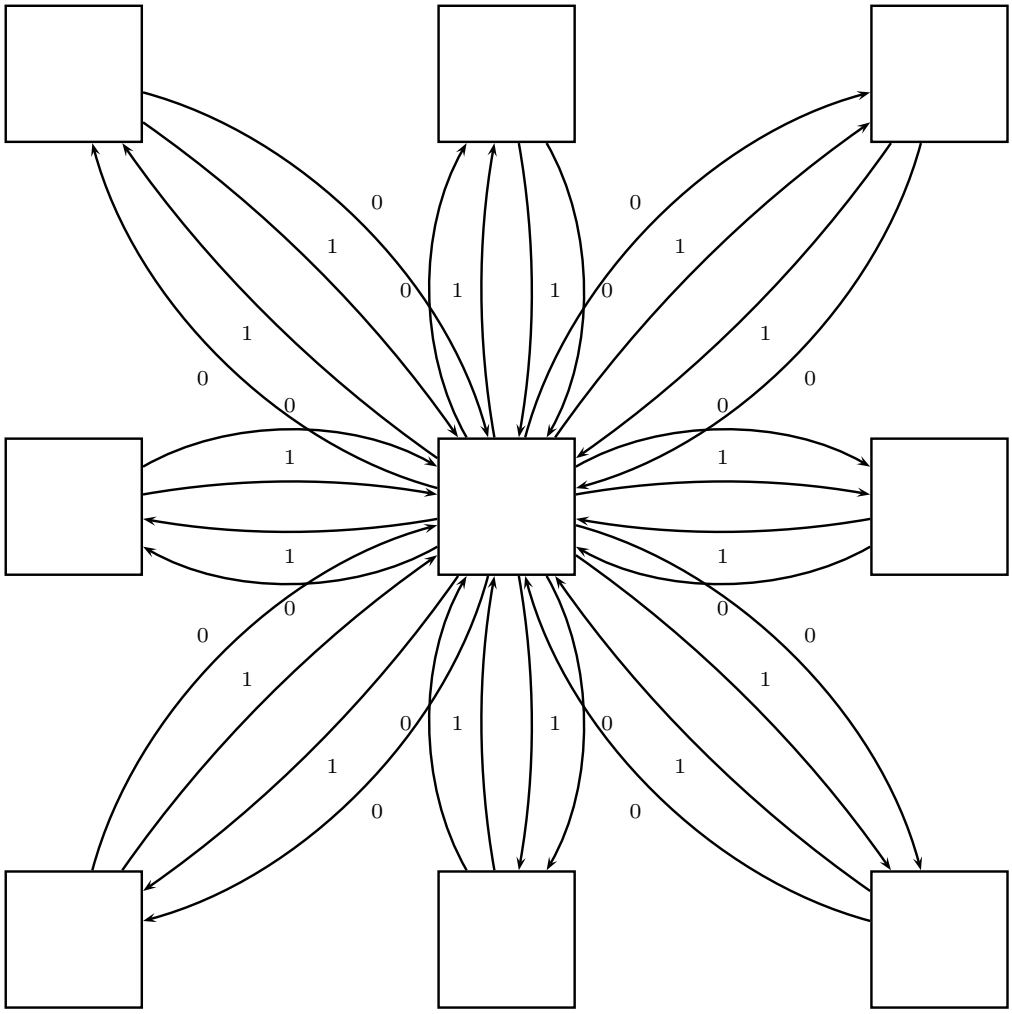
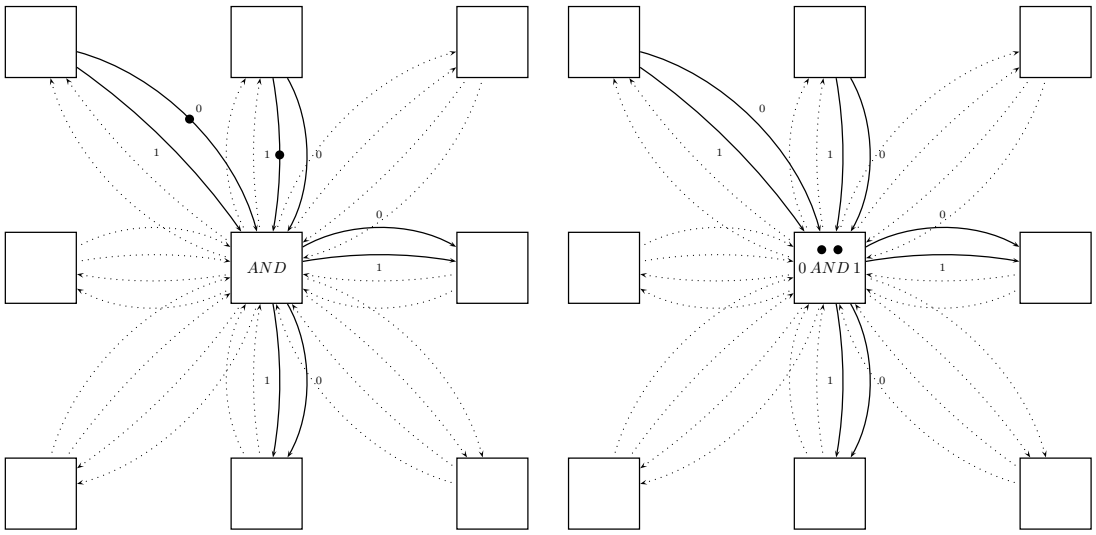


Figure 2-3: Edges of one ALA cell

are providing either a “0” token or a “1” token and that none of its active output edges is currently occupied by a token of either type. When firing, it consumes the input tokens (removing them from the input edges), performs its configured function, and deposits the result to the appropriate output edges (see Fig. 2-4 for an example of a 2-input, 2-output AND gate firing). As it is a marked graph, the behavior of this model is well-defined even without any assumptions regarding the timing of the computations, except that each computation will fire in some finite length of time after the preconditions are met.

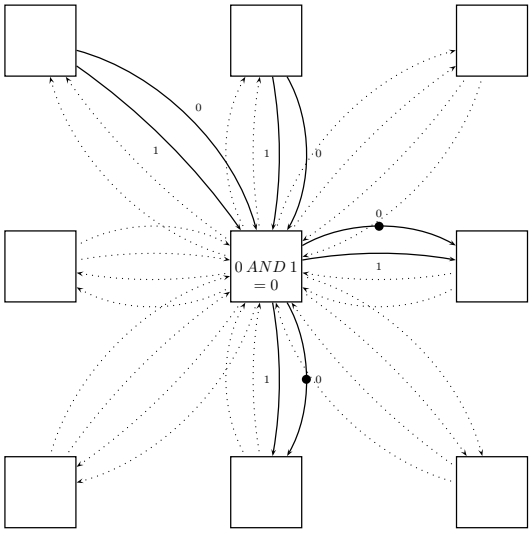
The model now operates asynchronously, and removes the need not only for a global clock, but any clock at all. In addition, the ”handshaking” mechanism is simply the charging of a capacitor to signal data ready (which capacitor is charged represents the value of the data) and the discharging of the same capacitor by a neighboring cell to represent data acknowledgment. As a final bonus, the charge removed from each capacitor can be placed by bucket-brigade logic [36] onto the output capacitor with minimal energy dissipation, lowering power consumption.

We have also introduced explicit accounting for the creation and destruction of tokens instead of implicitly doing both in every operation, as with traditional CMOS logic. For instance, in Fig. 2-4, since there are equally many inputs and outputs, no tokens must be created or destroyed. Only cells with more outputs than inputs must consume power, and only cells with more inputs than outputs must dissipate heat. While the model still uses the same irreversible Boolean functions, these functions can be thought of as being simulated by conservative logic which is taking in constants and dispersing garbage [12], enabling an easy pricing of the cost of non-conservatism in any given configuration.



(a) Firing conditions met

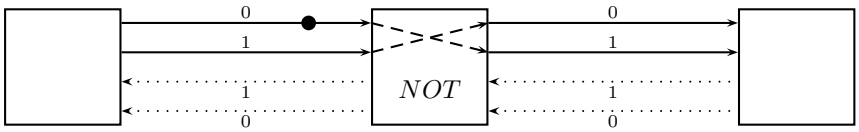
(b) Collect inputs; perform computation



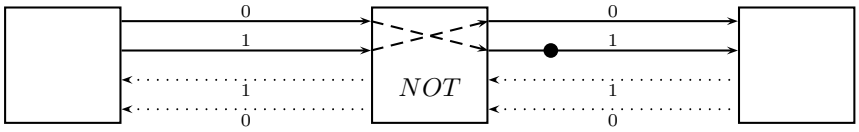
(c) Distribute result

Figure 2-4: An ALA cell firing; note that despite the loss of information (the inputs are not deducible from the outputs), tokens are conserved in this example

In addition, this model adapts much more easily to take advantage of adiabatic logic design. For instance, when a cell is being used only to ferry tokens from one place to another (e.g. an inverter, shown in Fig. 2-5), it can do so physically, instead of using a traditional, charge-dumping CMOS stage.



(a) A token is output by the cell to the left



(b) The token now passes to the right cell

Figure 2-5: A bit travels left to right through an inverting cell

Note the following possible ALA variations:

1. **No Diagonals.** Connections may only be present between vertically or horizontally adjacent cells, to simplify hardware layout.

2. **Alternate Lattices.** Indeed, any regular connection topology may be used, including alternate two-dimensional layouts such as hexagonal lattices or even three-dimensional structures such as the body-centered cubic lattice.

3. **More Functions.** The class of possible functions executed by each cell need not be limited to {AND, OR, XOR, NAND} but may include any function $f : \{0, 1, \emptyset\}^n \rightarrow \{0, 1, \emptyset\}^n$ (mapping from possible input states to possible output actions) where n is the number of neighbors of each cell. For $n = 4$, there are about 10^{61} functions on a fixed number of inputs; for $n = 6$, there are about 10^{687}. A cell executing function f may fire if f's present output is not $\emptyset^n$ (i.e. if the output has some non-empty elements) and every non-empty element of

the output points to either an inactive or empty set of output edges. Then each of those output edges would become populated with the value specified by f's output. There is a tradeoff between the number of functions allowed and the number of configuration bits in each cell needed to specify the function.

4. **Multiple Signals.** More than four token-storing edges may connect neighboring cells, allowing the conveyance of more parallel information in the same period of time. This could be used for special programming inputs, or for gates which act bitwise on multiple bits.

2.3.3 ALA Simulator

We wrote a simulator for Asynchronous Logic Automata using the C language, Guile (an embeddable Scheme interpreter), OpenGL, and freeglut. The code is listed in Appendix A. The high-level algorithm is to keep a list of cells whose firing conditions are met, and choose a random one to fire at every time step. The description of the configuration to simulate is generated at run-time by a Scheme program provided as input. Some examples of input code can be found in Appendix B. Figure 2-6 is a montage showing the first few frames of output from the circuit described by `lfsr.scm` (section B.2). The visualization used by the ALA simulator is different to that of the Logic CA: gate type is represented by color, and bits traveling through the gates have fading trails through these gates. This reflects where computation is happening (and where power is being dissipated).

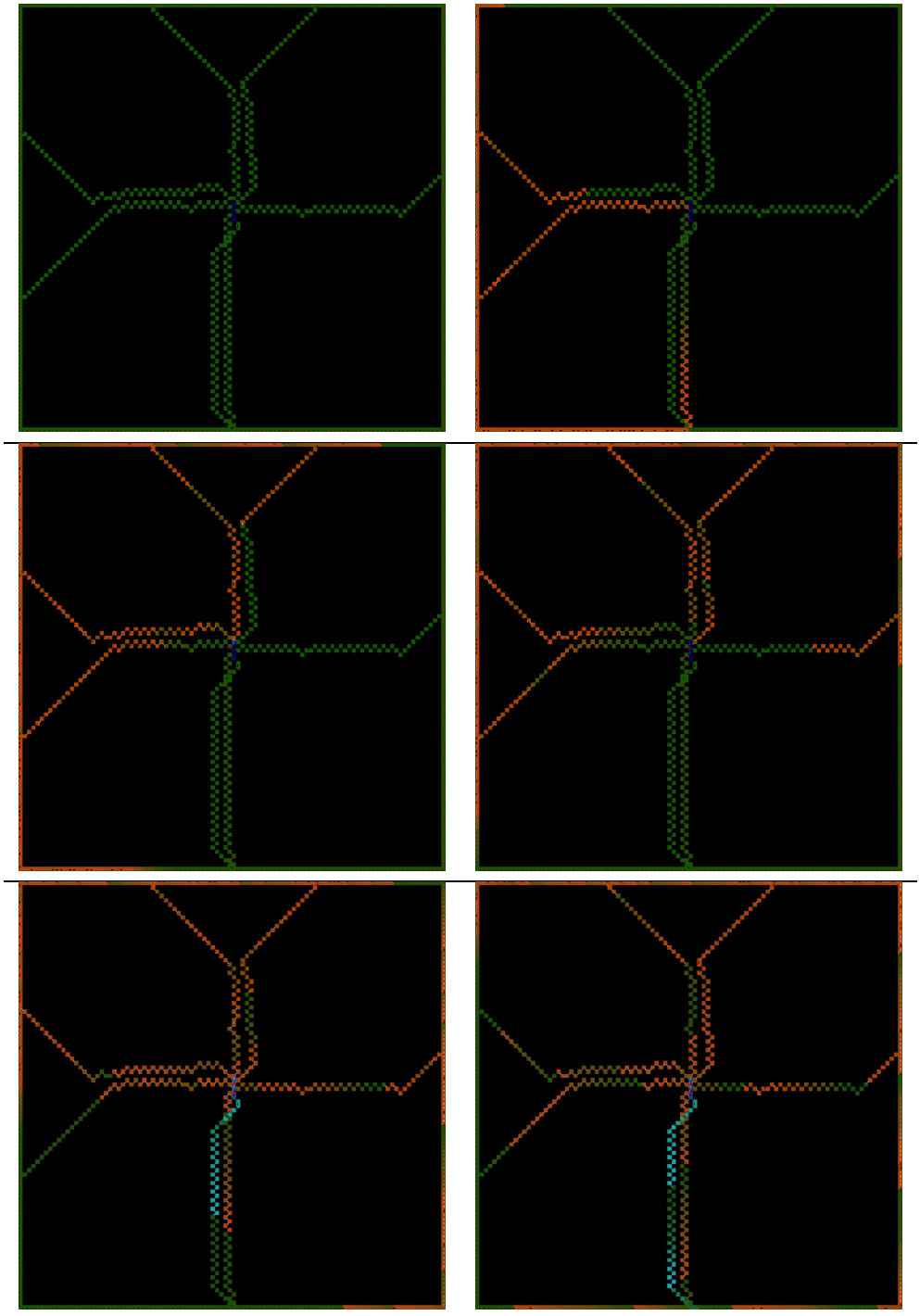


Figure 2-6: Six frames from the initial evolution of `lfsr.scm`

Chapter 3

Algorithms

3.1 SEA Implementation

3.1.1 Motivation

One application that is particularly well-suited to implementation in a model such as this is symmetric encryption as a Feistel cipher. Because these cipher structures can operate on streams, they can take advantage of an arbitrary degree of parallelism to deliver a corresponding degree of security, and because Feistel ciphers are typically expressed as dataflow diagrams, it is natural to express the algorithm in the form of a fixed picture that data flows through – the easiest way to program the logic CA.

In the space of Feistel ciphers, with a context in which resources are priced at fine granularity, it is natural to choose a minimalistic Feistel cipher which provides all the desired security properties such as diffusion and resistance to linear and differential cryptanalysis while requiring the least computation. This is the role filled by SEA: a Scalable Encryption Algorithm for Small Embedded Applications [39].

3.1.2 SEA Components

SEA assumes very little computing power from its host. The primitive operations used to construct the Feistel functions are:

1. Bitwise XOR
2. Substitution box composed from AND, OR, and XOR
3. Word rotate
4. Bit rotate
5. Addition

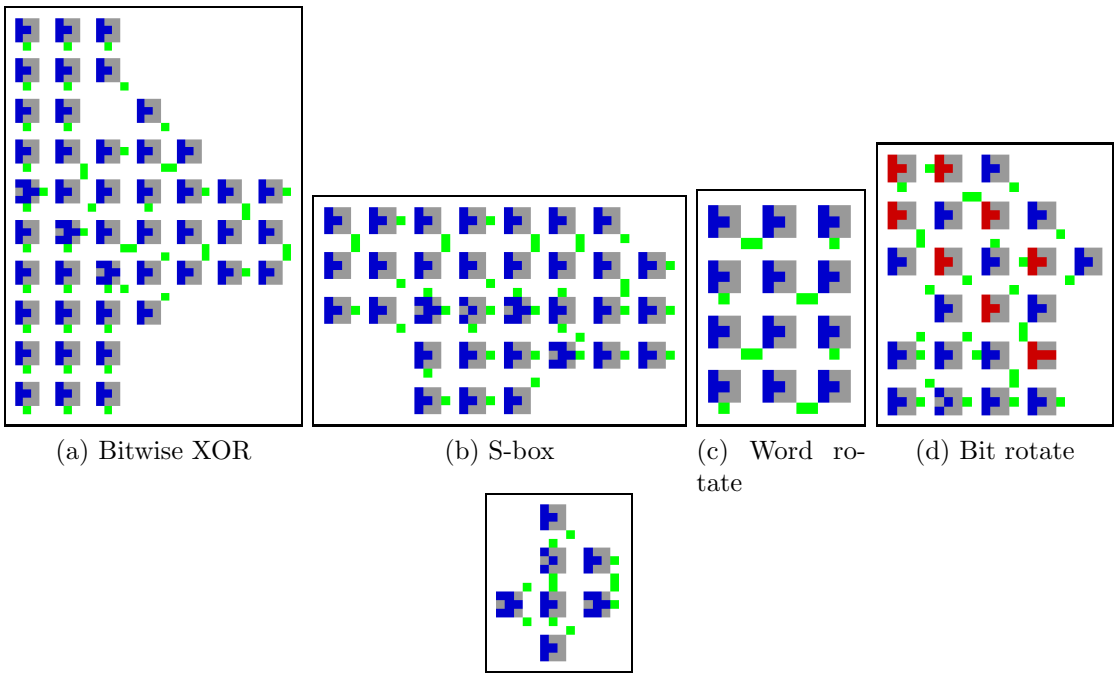
Each of these components has been implemented using the synchronous Logic CA model, as seen in Fig. 3-1. Note that computation proceeds from right to left and bottom to top in these figures.

Since XOR is a primitive function of the CA cell, bitwise XOR over 3 simultaneous streams is largely an exercise in routing and timing, with the computation taking place in the center (Fig. 3-1a). The substitution box (s-box, Figure 3-1b) is simply implemented as it is described in the SEA paper (with some extra cells to ensure timing uniformity in the Logic CA):

$$\begin{aligned} x_0 &= (x_2 \wedge x_1) \oplus x_0 \\ x_1 &= (x_2 \wedge x_0) \oplus x_1 \\ x_2 &= (x_0 \vee x_1) \oplus x_2 \end{aligned}$$

As shown, x_0 is on the bottom. Word rotate (Fig. 3-1c) is a series of four swaps which re-routes three parallel streams of bits to rotated positions. Bit rotate (Fig. 3-1d) is a

carefully timed delay circuit which extracts the last bit of each byte and delays it until the end of the byte. The addition block (Fig. 3-1e) is actually the simplest of these circuits – it need only keep track of the carry bit and execute a full add every time step.



(a) Bitwise XOR (b) S-box (c) Word rotate (d) Bit rotate

(e) Addition

Figure 3-1: Components of SEA implemented in the Logic CA

3.1.3 Complete Round

Given these primitives, an encryption round of SEA can be constructed as seen in Fig. 3-2. The two inputs are at the bottom edge on either side. The right-hand block is passed through an adder (which would be connected in the context of the full cryptosystem to a key generation round which looks much the same), then left to the S-box, then two of the three words are bit-rotated, and finally the result is XORed with the word-rotated left-hand block to produce what will be the right-hand block in the next round (in the context of the full cryptosystem, sequential rounds are followed by a crossover). Note that the right-hand block must arrive in advance of the

left-hand block in this model, since it must pass through the horizontal section before meeting the left-hand block at the XOR. By assembling this building block into the appropriately sized structure, we can obtain any desired degree of cryptographic confusion and diffusion, with no cost to continuous throughput (only latency).

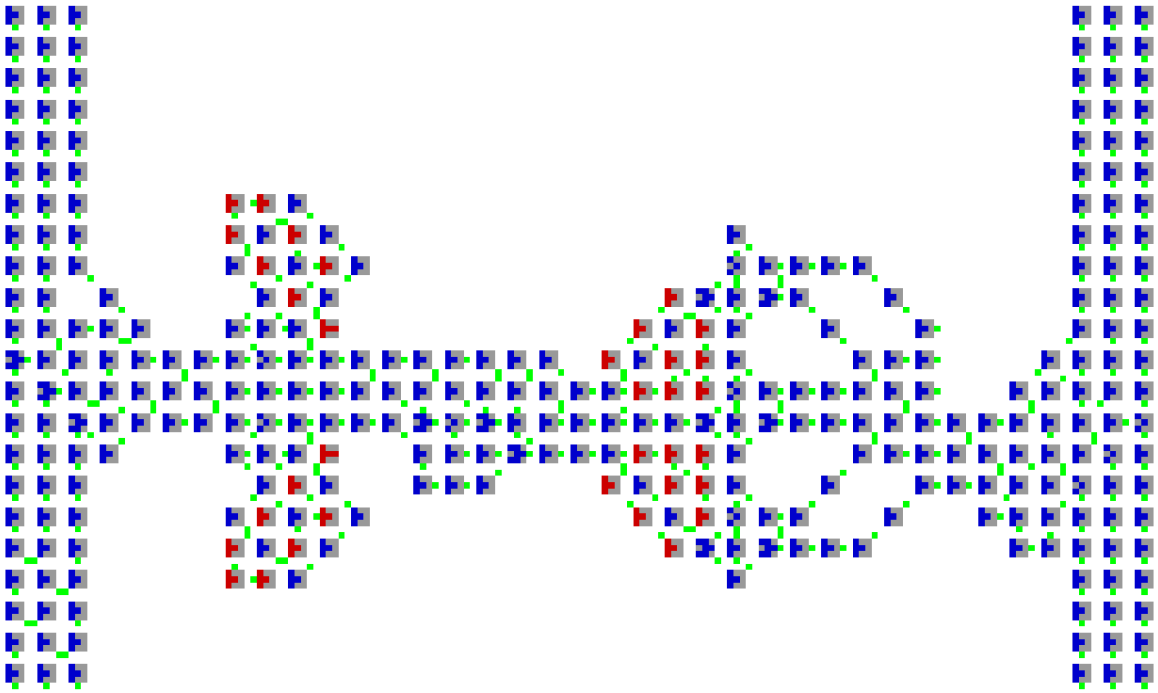


Figure 3-2: A single round of SEA built from primitives

3.1.4 Encrypt-Decrypt

Fig. 3-3 is a montage showing what happens when we feed the letters "CBA" through one encryption round and one decryption round of SEA. It should be read from left to right and then from top to bottom. In the first frame, you can see the vertically stacked letters on the left side, then you can follow the bits as they are delayed while the meaningless right-hand block is passing through the horizontal elements. In the fifth frame, the blocks meet at the XOR junction. The seventh frame shows the "encrypted" data in full between the encrypt round and the decrypt round. In the

eighth frame, this data is re-XORed with the same key, regenerated by the machinery of the decrypt round, and word-rotated in the inverse direction. In the final frame, we can see that the letters “CBA” have once again emerged.

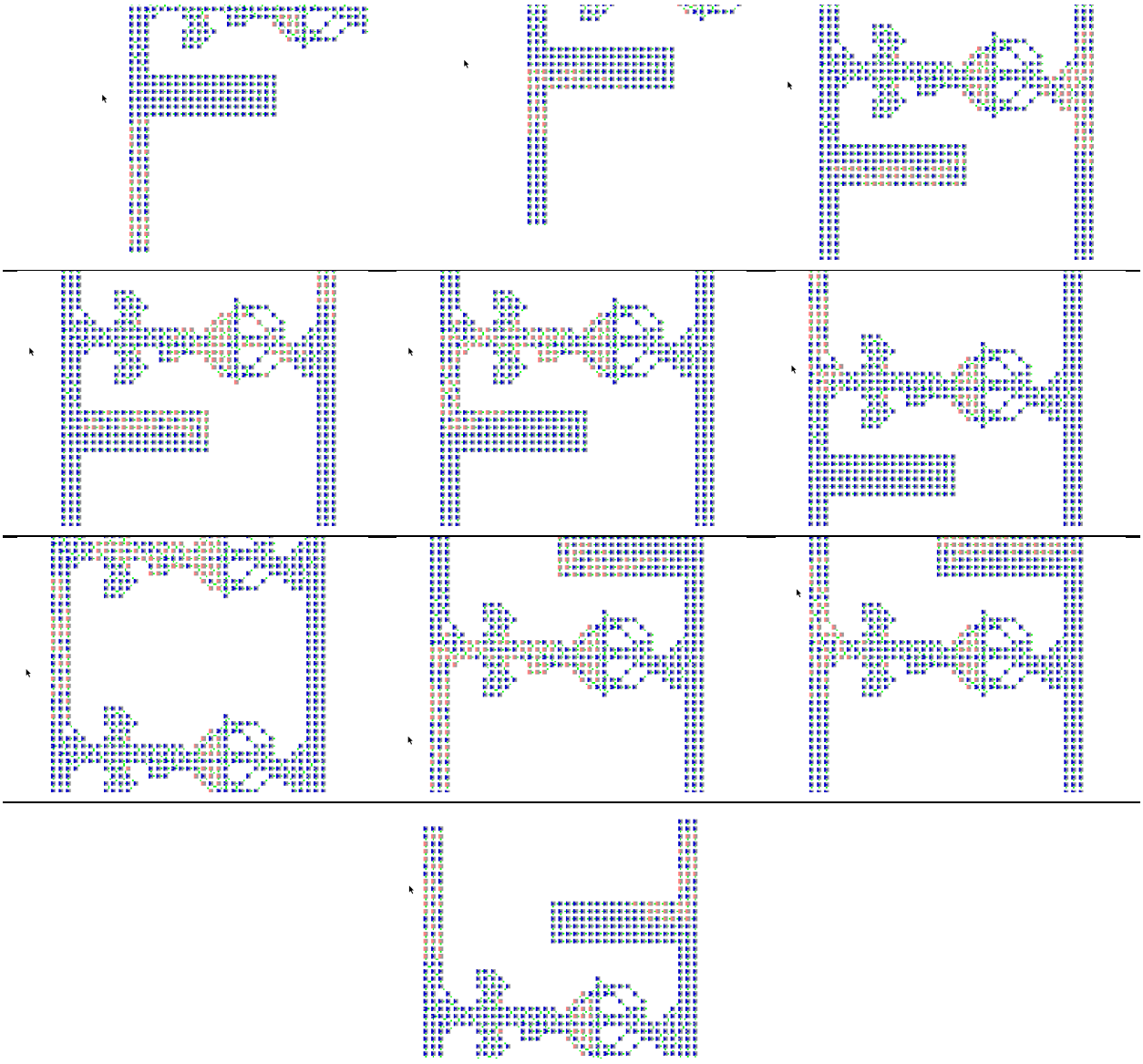


Figure 3-3: An encrypt round and a decrypt round of SEA passing the information “CBA”.

3.2 Bubble Sort Implementation

3.2.1 Overview

"Bubble sort" is a simple, classic sort algorithm which can be described as repeatedly checking each neighboring pair of elements and swapping them if they are out of order [21]. It is so called because elements which are out of place will gradually "bubble" up to their sorted location through a series of nearest-neighbor transpositions. On sequential computers, the checks must be performed one at a time, meaning that a sequence of $O(n^2)$ operations (one check/swap each) is needed to guarantee that the entire set has been sorted. Thus bubble sort is typically ignored in favor of algorithms such as quicksort, which can sort with only $O(n \lg n)$ operations. On a cellular computer, checks of non-overlapping pairs of elements can be performed simultaneously at no extra cost, so $O(n)$ operations (each comprising $O(n)$ checks) are sufficient – fewer than the best possible sequential algorithm. Note that with denser interconnection topologies, such as shuffle-exchange or hypercube, only $O(\lg^2 n)$ operations may be needed [7], but with only local connections, $O(n)$ operations is provably optimal [6].

The CA implementation of bubble sort is made from two main components, which we call the "switchyard" and the "comparator". Fig. 3-4 shows how these components interact. Note that this scheme can be viewed as a compromise between the sequential bubble sort algorithm and the "diamond" sorting network of Kautz [20], which is in turn equivalent to the odd-even transposition sorting network [22, 35]. Each comparator operates on two binary strings (elements to be sorted) and outputs them unmodified, along with a single-bit control line which indicates whether the elements are out of order. If so, the corresponding switchyard transposes them; otherwise, it also passes them back out unmodified for the next round of comparisons. Half the comparators or half the switchyards are active at any given time step (necessary since all the pairs being compared simultaneously are non-overlapping).

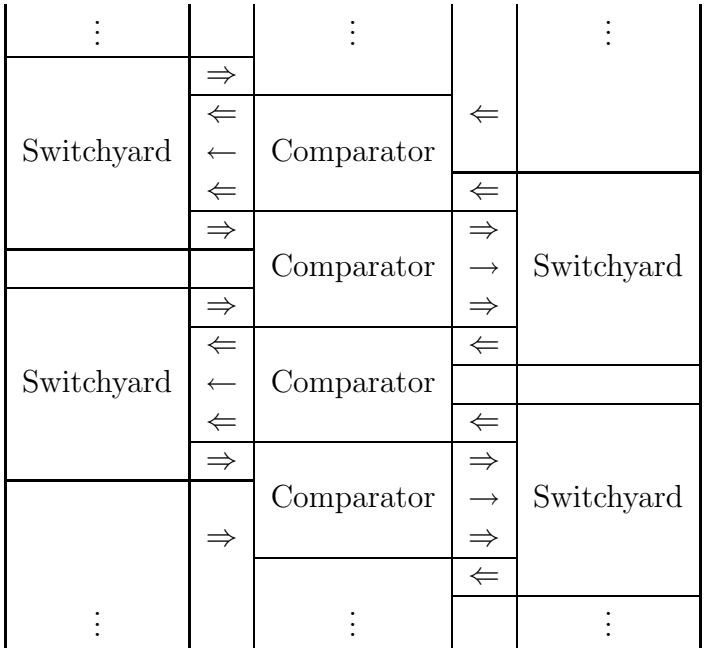


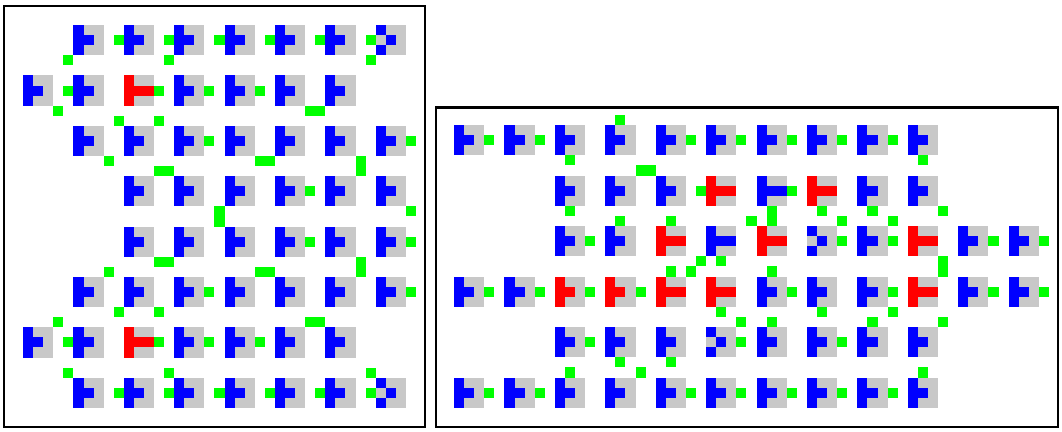
Figure 3-4: Block-level architecture of bubble sort in the Logic CA

3.2.2 Sort Components

Fig. 3-5 shows a left-side switchyard and comparator implemented in the synchronous Logic CA, oriented the same way as the topmost switchyard and comparator in Fig. 3-4. The inputs of the switchyard are on the right-hand side in the vertical center, with the control line being between the binary string inputs. The outputs of the switchyard are on the right-hand side at the vertical extremes. The inputs of the comparator are on the right, and the outputs on the left (with the control line in the vertical center).

The switchyard (Fig. 3-5a) includes very little logic; it is mostly wires (chains of AND gates whose inputs are both tied to the last). These paths extend from both inputs to both outputs (four in all), and the control paths extend from the control input to both corners (where the NAND gates are). The paths which cross in the middle are only enabled when the control line is on (they are ANDed with it), and the paths which are aligned with the top and bottom edges are only enabled when the control input is off (they are ANDed with the NAND of the control line). The paths are merged at the

outputs using OR gates.



(a) Switchyard (b) Comparator

Figure 3-5: Components of sort implemented in the Logic CA

The comparator (Fig. 3-5b) also includes long data paths, which can be traced along most of the top and bottom edges, but unlike the switchyard, contains significant logic and state in the center. At the right edge are detectors for bit mismatches in the input, which are inputs to two OR-based bit stretchers that stabilize NAND-NAND flip-flops which compute the output and latch it on the control line. A reset input, intended to connect to a timer depending on the length of strings being sorted, can be seen coming down from the top, toward the left-hand side.

3.2.3 Complete Sort

We can flip and assemble these elements, given the pattern in Fig. 3-4, into a CA like Fig. 3-6. In order to match up the vertical locations of inputs and outputs in the comparator and switchyard, vertical wires are added where needed.

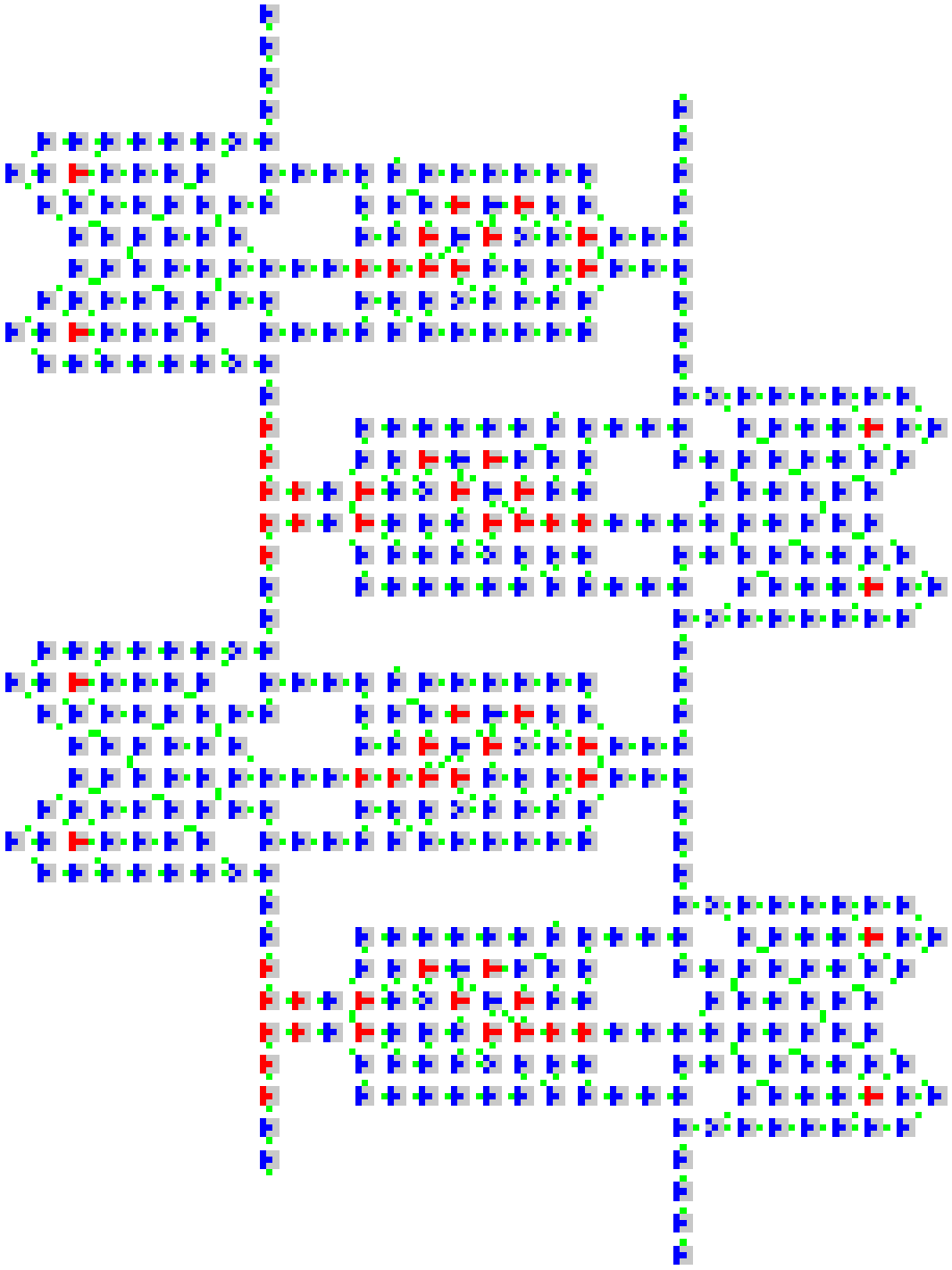


Figure 3-6: A four-element sorter built from primitives

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Ongoing and Future Work

4.1 Conformal Computing Team

This work was done in the context of the Conformal Computing project, a project to develop computers that:

- cover surfaces and fill volumes
- are incrementally extensible
- have embedded, unobtrusive form factors
- solve distributed problems with distributed solutions
- adapt to applications and workloads
- operate reliably from unreliable parts

This project is a collaboration between the Massachusetts Institute of Technology Center for Bits and Atoms and the North Dakota State University Center for Nanoscale

Science and Engineering. The Conformal Computing team includes Kailiang Chen, Kenny Cheung, David Dalrymple, Ahana Ghosh, Forrest Green, Mike Hennebry, Mariam Hoseini, Scott Kirkpatrick, Ara Knaian, Luis Lafeunte Molinero, Ivan Lima, Mark Pavicic, Danielle Thompson, and Chao You. Work being done in this project fills in the lower and higher level components necessary to turn the concepts in this thesis into a complete system:

- **Programming Models**
  - **Hierarchical Design Tool** and
  - **Mathematical Programming**
- Cellular Microcode
  - Logic CA or
  - ALA, possibly with
  - **Coded Folding**,
  - **Scale-Invariance**, and
  - **Fault Tolerance**
- **Hardware Realizations**
  - **CA Strips**,
  - **CA ASIC**, and
  - **Molecular Logic** [2, 9, 42]

4.2 Programming Models

The primary disadvantage to practical fabrication and use of ALA in their present form is the need to simultaneously initialize all cells with the configuration data

before useful computation can be performed, as well as the lack of special-purpose tools for generating this data.

4.2.1 Hierarchical Design Tool

Forrest Green is working on a design tool for the Logic CA and ALA that uses "libraries" of small hand-coded primitives for operations such as adding and comparing, and allows the composition of new, more complex primitives using a path-finding algorithm to connect input and output ports. This tool would have the appearance of a visual dataflow programming language like Max/MSP, Labview, or Simulink, but the picture would actually represent a directly executable plane of CA configuration data.

4.2.2 Mathematical Programming

Scott Kirkpatrick and Luis Lafuente Molinero are working on the theory of generating ALA patterns as the result of an optimization problem. For instance, the sort described in section 3.2 can be derived as the optimal searcher through the space of permutations described as products of primitive transpositions. In addition, they are developing an optimization solver which will run in software on the ALA, and fully solve certain classes of mathematical programs in general.

4.3 Model Improvements

4.3.1 Coded Folding

Erik Demaine and Kenny Cheung are helping to develop a protocol for loading configuration data in with a communication channel to exactly one cell, by computing a Hamiltonian path which contains all the cells to be programmed, and

informing each cell, after it is configured, in which direction it should forward the rest of the data stream, then finally propagating a signal to begin the computation. We are also considering similarities between ALA and von Neumann's original self-reproduction automaton [46] and are exploring ways to make the von Neumann automaton asynchronous and easier to design in.

4.3.2 Scale-Invariance

ALA are translation-invariant: moving a boundless, unprogrammed ALA any number of cells in any direction does not change it. This is particularly useful for problems with a naturally translational structure: sorting flat lists, processing streams, simulating locally interacting elements in space, etc. However, many types of problems are naturally scale-invariant (i.e. they contain structures similar to their own), such as parsing markup languages or complex recursive algorithms. These problems, when embedded in Euclidean space, require direct, high-speed interactions between relatively distant elements, since the number of elements reachable through n scaling steps of k scale factor grows as k^n, while the number of elements reachable through n translations in k dimensions grows as n^k, and $k^n \gg n^k$ as $n \to \infty$. Even though the ALA can propagate signals along fixed paths fairly quickly, we expect that with hardware available today, this speed would come at best within two orders of magnitude of the speed of light, and this cost will accumulate for scale-invariant sorts of problems. This suggests that a reasonable (though significant) modification to the ALA concept would be to add a hard-wired, scale-invariant overlay which helps speed up this type of problem.

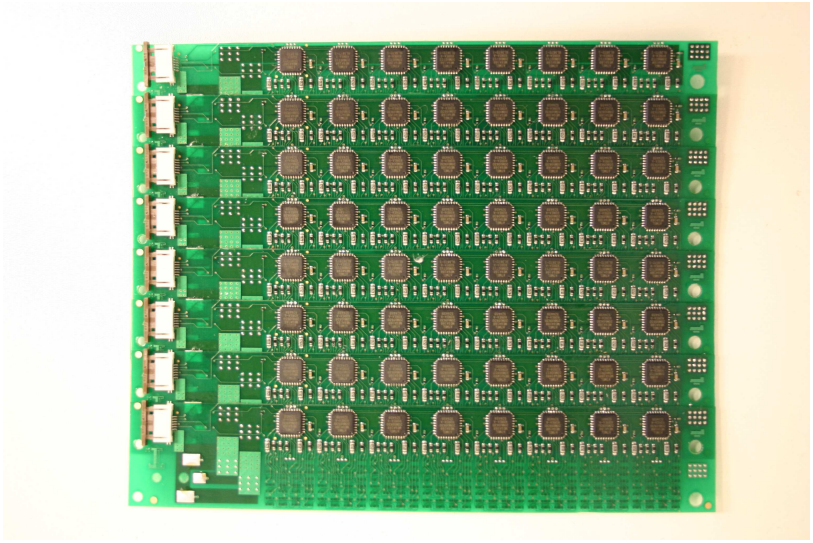
4.3.3 Fault Tolerance

Although error correction can implemented in ALA software using redundant wires and majority voting [45], we are exploring various ways to make the model itself tolerant of faults in the underlying hardware. This may take the form of an arm searching for defects and designing around them [24], or cells which implement a code internally [30]. Both methods may also be helpful, since some hardware defects are permanent as a result of fabrication errors, while others are transient as a result of thermal noise or other factors disturbing an element which is otherwise in specification.

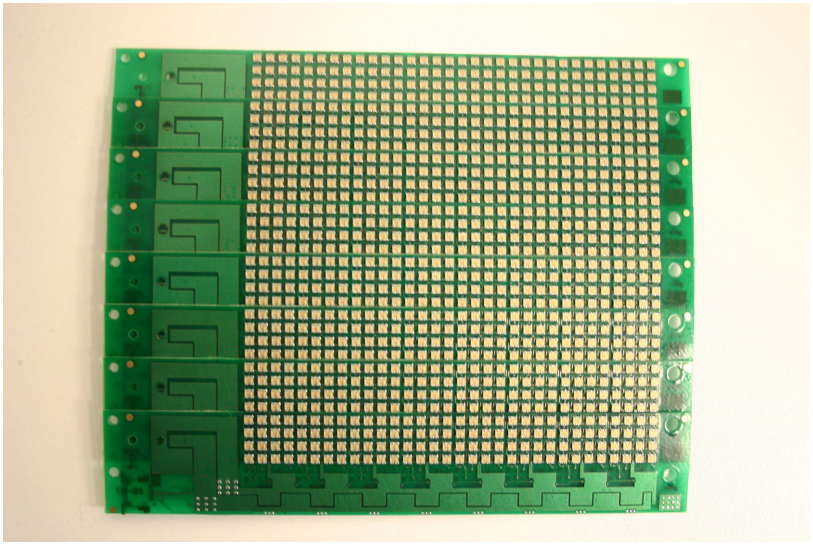
4.4 Hardware Realizations

4.4.1 CA Strips

Mike Pavicic, Michael Hennebry, and their team at the Center for Nanoscale Science and Engineering and North Dakota State University (NDSU) have created a hardware platform (Fig. 4-1) based on an 8x8 array of ATMega168 AVR processors, with a 32x32 array of color LEDs on the opposite face, which implements a Logic CA simulator. This substrate is actually composed of 8 "strips" each with 8 processors, and can be indefinitely extended in one direction. They are working towards processes which are indefinitely extensible in two directions, and eventually three.



(a) One side of the strips is an 8x8 array of AVR processors



(b) The other side is a 32x32 array of color LEDs

Figure 4-1: CA "strips" developed at North Dakota State University

4.4.2 CA ASIC

Chao You (also at NDSU) has designed a silicon layout for a Logic CA (Fig. 4-2) and Kailiang Chen (MIT) has fabricated an analog logic version of the Logic CA (Fig. 4-3) and is working on a silicon layout for an ALA.

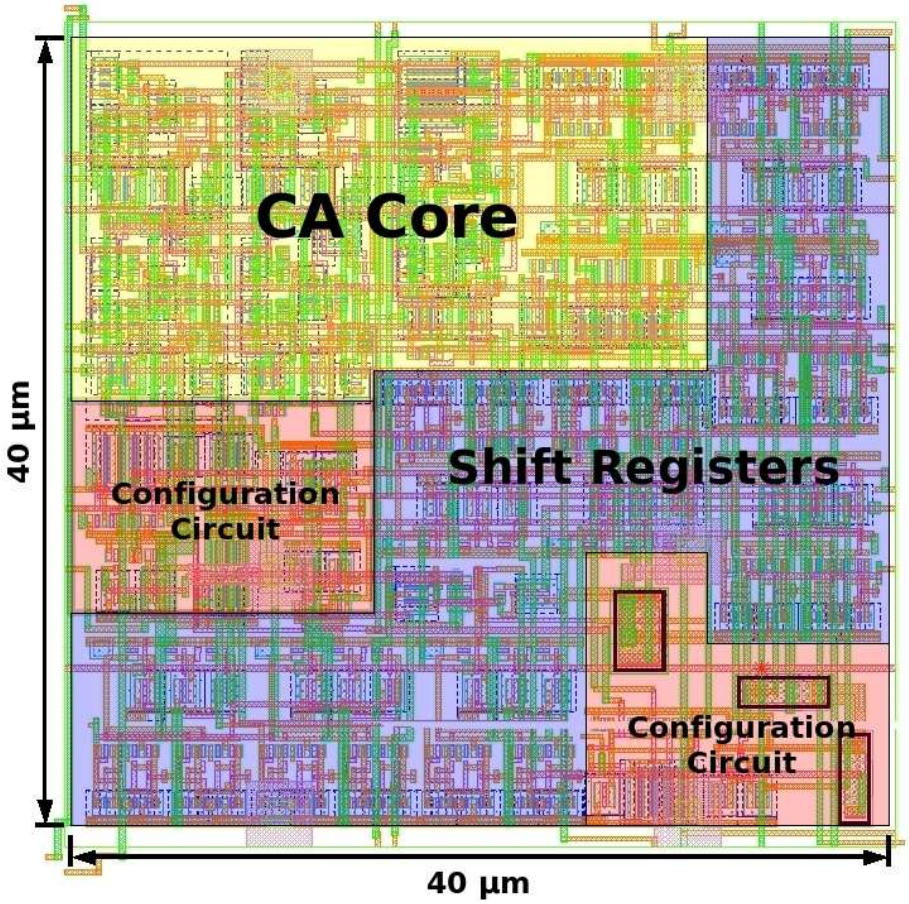


Figure 4-2: Logic CA silicon layout by Chao You

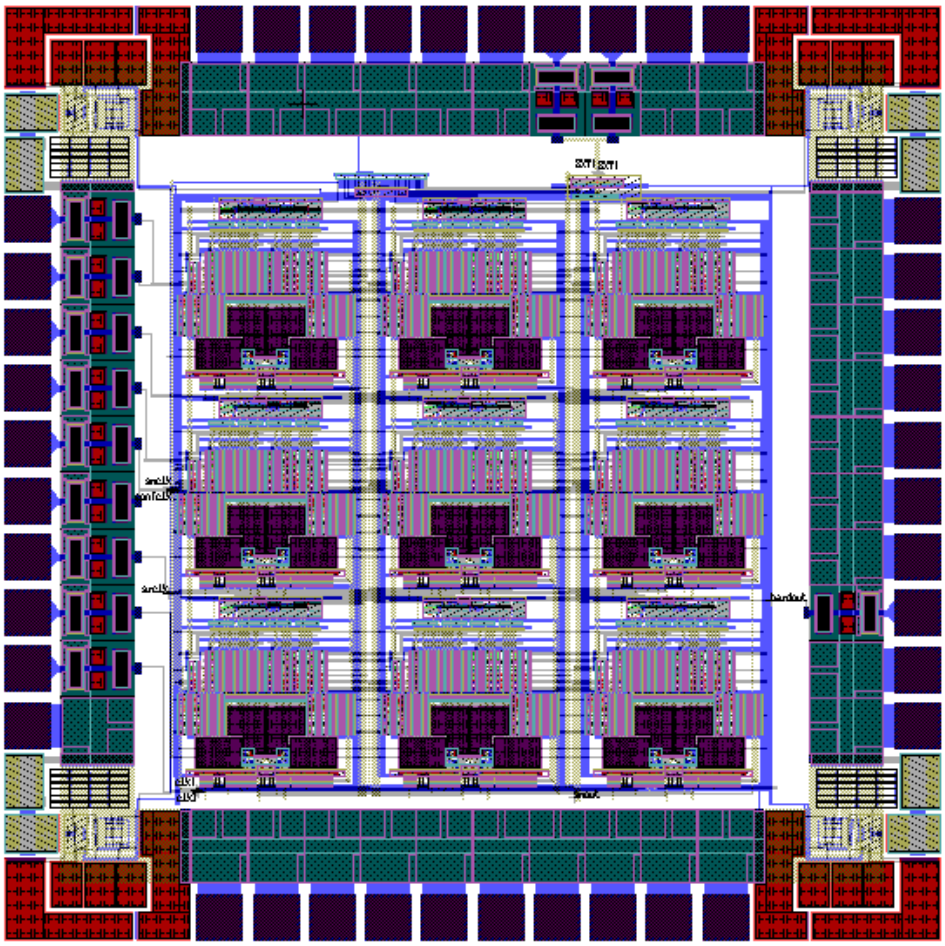


Figure 4-3: Analog Logic Automaton silicon layout by Kailiang Chen

4.4.3 Molecular Logic

We are also looking ahead toward direct-write nanostructures working with Joe Jacobson's group, which have demonstrated the fabrication of nanowire transistors [2], and toward other types of molecular logic such as that being developed at Hewlett-Packard Laboratories [9, 42].

4.5 Applications

Increasingly, computer graphics performance is improved mainly by adding pixel pipelines, but consider the possibility that every single pixel can be processed in parallel. We are examining 2D and 3D graphics applications in which sites at regular intervals on a planar array or one face of a spatial array corresponds to a light-emitting element, and the computation of pixel values takes place in a spatially distributed, concurrent way. For 3D graphics we may even imagine a raytracer operating by actually firing rays as propagating signals through the volume of the device. In addition, we are considering the case that these special sites on the surface correspond to photodetectors, or even that both types of sites are present, and using clever optical filtering to make an indefinitely high-resolution camera, or to make an indefinitely large multi-touch display.

Meanwhile, high-performance computing is limited by the power consumption of the largest computers, but also by the ability to program them. We have also had substantial interest from the high-performance computing community, not just about using physical ALA to solve scientific problems, but even to use the ALA programming model as a way to program existing supercomputers. By implementing an extremely well-optimized ALA simulator, the ALA software running on top of it need not take the standard considerations that make developing sequential software for supercomputers so difficult. In addition, the flexibility of ALA architectures allow for indefinite fixed-precision numbers, removing floating-point roundoff as a common source of scientific computing bugs. Implementing a high-performance computer using ALA as the physical substrate would provide all these benefits in addition to saving power.

Finally, we are looking forward to using the ALA to implement Marvin Minsky's Emotion Machine architecture [27], which requires numerous agents acting concurrently as well as a hierarchy of critics which determine which agents ought to be enabled at any given time. The massive, but non-uniform, parallelism of the Emotion

Machine fits neatly onto ALA (which are also non-uniform and massively parallel). Although this architecture would be a useful one for the development of ALA software in any case, it is supposed that this could lead to a machine which displays some characteristics of intelligence.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Concluding Remarks

Computer science as an industry, and to some extent as a field of study, was derailed by a series of unfortunate events that led to von Neumann's EDVAC [47], a machine made to do arithmetic in a flexible way, being the ancestor of all modern computer designs. The first microprocessor, Intel's 4004, was built using this architecture because it was designed for a high-end desk calculator. Unfortunately, the concept of the microprocessor has been tied to this architecture ever since, despite the best efforts of academics and venture capitalists alike, leading to much pain for programmers, who pass much of it on to the end user through unreliable software. However, this type of machine will soon finally reach the point in its evolution at which it is no longer even "good enough" for the constantly rising performance that industry expects and demands. This is a golden opportunity for a brand of "fundamentalist" computer science: we can revisit the final (rather than the earlier, misunderstood, and hopelessly popular) wisdom of von Neumann [46] and Backus [4], and give credence to all the great ideas from over 20 years ago (including but hardly limited to [19, 12, 24, 11, 20]) that have been ignored by the mainstream establishment. Fundamentalist computer science is also fundamentalist in that we are interested in the fundamentals of computing. This may sound like a vacuous statement, but there is

far too little attention given today to the theory of how real-world physics can best support reliable, controlled computations of arbitrary size and complexity.

In a world where computers are not so distanced from physics, they will be far more intuitive to work with and program, since humans are very comfortable with items that have a size, location, speed, and other well-defined properties. We call these things "concrete", although they may in fact be abstract concepts in our minds, because they can be thought of in the physical framework all people need to deal with the world we live in. In a modern computer program, variables are omnipresent, and materialize wherever they are summoned by name, while statements exist only one at a time, in an order that is not even possible (in general) to determine in advance. These concepts are plainly impossible physically, so we simulate them by using random-access memories and program counters. These concepts are all well and good, but should be chosen, rather than imposed. For instance, when so-called "object-oriented programming" is employed, the computer is required to do *extra* work to ensure that certain variables are only accessible within a restricted domain, while from a physical perspective, such a restriction ought to make the computer's work much lighter, since it would no longer need to deliver variables between distant objects while keeping their values consistent. The hiding of physics behind an over-generalized model of computation introduces an exaggerated tradeoff between "programmer time" and "computer time". Would you rather take full advantage of the computer's forced non-physicality, but be burdened in your programming by the non-intuitiveness of non-physicality; or would you rather program in an intuitive way (by encapsulating variables in objects and composing transforming functions with each other) but pay the price of having these more physically constrained ideas simulated by a non-physical model that is in turn being simulated by physics itself, causing program speed to suffer? Programming in an intuitive way ought to result in *faster* programs, because physical constraints are recognized to a greater extent.

In addition, the design, construction, and configuration of systems which are physically large or sparse should become far easier, because the size or sparsity (and the associated restrictions on state mixing) could be represented automatically in the programming model. No longer must such systems be subdivided into "nodes" – domains within which engineers were successfully able to circumvent physics without too much cost, and outside of which, there exist few principled ways to construct programs. Instead, by exposing the physical constraints within the programming models, these systems can be exploited fully without any distinguished programming techniques.

We can imagine a world [16, 23] which is filled with computing devices, which are not so much hidden as transparent. This computation can be put to use for everything from managing the energy use of buildings, to distributed videoconferencing in which network routing is derived as an optimization constrained by physics rather than as ad-hoc algorithms, to simulating human brains. All this computing can be made far easier to use, to the extent that the distinction between users and programmers will become blurred. Due to the less profound boundaries between the inside and outside of an individual "computer", it can be made available much more flexibly.

Of course, these ideas have a long lineage, but we believe that the need and the means to make it happen are present (or at worst, imminent). More than the final vision or the specific details, the overall message to take away is that we are likely on the verge of a paradigm shift since von Neumann's EDVAC architecture is no longer working for many industry applications, and that making the physics of computing transparent instead of hidden is a good way to proceed. On a lower level, the author has shown that the problem of synchronicity can be addressed by making the individual bits each their own processor which can enforce data dependencies, and that dividing space into a regular lattice of similar processing elements at the granularity of one bit is a simple yet effective way of representing the speed of light and density constraints of physics. This work is a small but important step toward the computers of the future.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

ALA Simulator Code

A.1 ca.c

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #define CELL_OUTPUTS 8
5
6 long int cell_fade = 1000000;
7 unsigned int shuffle_mix = 1000;
8
9 struct ca_cell {
10   char drawable;
11   struct ca_cell* input_a;
12   struct ca_cell* input_b;
13   struct ca_cell* outputs[CELL_OUTPUTS];
14   int output_count;
15   int input_a_output_idx;
16   int input_b_output_idx;
17   char function;
18   char input_a_state;
19   char input_b_state;
20   long int latest_update;
21   char latest_bit;
22   int order_index;
23   int draw_order_index;
```

```
24 };
25
26 struct ca_canvas {
27   unsigned int width;
28   unsigned int height;
29   struct ca_cell* cells;
30   int* update_order;
31   int ready_cells;
32   int* draw_order;
33   int drawable_cells;
34   unsigned long int ca_time;
35 };
36
37 char ca_and(char a, char b) {
38   return a & b & 1;
39 }
40 char ca_or(char a, char b) {
41   return (a | b) & 1;
42 }
43 char ca_xor(char a, char b) {
44   return (a ^ b) & 1;
45 }
46 char ca_nand(char a, char b) {
47   return ~(a & b)&1;
48 }
49
50 typedef char (*ca_operation)(char,char);
51
52 static ca_operation ca_functions[4] = {&ca_and, &ca_or, &
     ca_xor, &ca_nand};
53
54 int ca_cell_check_conditions(struct ca_cell* cell) {
55   if(!((cell->input_a_state & 2) && (cell->input_b_state &
        2))) return 0;
56   int i;
57   for(i=0; i<cell->output_count; i++) {
58     if(cell->outputs[i]->input_a == cell) {
59       if(cell->outputs[i]->input_a_state & 2) return 0;
60     } else if (cell->outputs[i]->input_b == cell) {
61       if(cell->outputs[i]->input_b_state & 2) return 0;
62     }
63   }
```

```
64   return 1;
65 }
66
67 int ca_cell_update(struct ca_cell* cell) {
68   if(ca_cell_check_conditions(cell)) {
69     char new_output_state;
70     int i;
71     new_output_state = 2 | ca_functions[(int)cell->
          function](cell->input_a_state,cell->input_b_state);
72     cell->latest_bit = new_output_state & 1;
73 #ifdef CA_DEBUG
74     printf("cell␣updated␣recently,␣output␣%d\n",
          new_output_state);
75 #endif
76     cell->input_a_state = 0;
77     cell->input_b_state = 0;
78     for(i=0; i<CELL_OUTPUTS; i++) {
79       if(cell->outputs[i]!=NULL) {
80         if(cell->outputs[i]->input_a == cell) {
81           cell->outputs[i]->input_a_state=new_output_state
                ;
82         }
83         if (cell->outputs[i]->input_b == cell) {
84           cell->outputs[i]->input_b_state=new_output_state
                ;
85         }
86       }
87     }
88     return 0;
89   } else {
90     return 1;
91   }
92 }
93
94 void ca_canvas_swap(struct ca_canvas* canvas, int n, int k
     ) {
95   int temp;
96   temp = canvas->update_order[n];
97   canvas->update_order[n] = canvas->update_order[k];
98   canvas->update_order[k] = temp;
99   canvas->cells[canvas->update_order[k]].order_index=k;
100  canvas->cells[canvas->update_order[n]].order_index=n;
```

```
101 }
102
103 void ca_canvas_shuffle(struct ca_canvas* canvas) {
104   int n;
105   if(shuffle_mix==0) {return;} else if (shuffle_mix==1000)
          {n=canvas->ready_cells;} else {
106   n = ((double)shuffle_mix)/1000*((double)canvas->
         ready_cells);}
107   if(n>canvas->width*canvas->height) n=canvas->width*
         canvas->height;
108   while (n > 1) {
109     int k = rand() % n;
110     n--;
111     ca_canvas_swap(canvas, n, k);
112   }
113 }
114
115 void ca_canvas_update(struct ca_canvas* canvas) {
116   int k;
117   struct ca_cell* updated_cell = NULL;
118   for (k=0;k<canvas->ready_cells;k++) {
119     int i=canvas->update_order[k];
120     if(canvas->cells[i].drawable) {
121       if(!(updated_cell || ca_cell_update(&canvas->cells[i
             ]))) {
122         updated_cell=&(canvas->cells[i]);
123         canvas->cells[i].latest_update=canvas->ca_time;
124       } else if (!ca_cell_check_conditions(&canvas->cells[
             i])) {
125         canvas->ready_cells--;
126         ca_canvas_swap(canvas, canvas->ready_cells, canvas
               ->cells[i].order_index);
127       }
128     } else {
129       canvas->ready_cells--;
130       ca_canvas_swap(canvas, canvas->ready_cells, canvas->
             cells[i].order_index);
131     }
132   }
133   if(updated_cell) {
134     int i;
135     if(updated_cell->order_index < canvas->ready_cells) {
```

```
136       canvas->ready_cells--;
137       ca_canvas_swap(canvas, canvas->ready_cells,
             updated_cell->order_index);
138     }
139     for(i=0; i<CELL_OUTPUTS; i++) {
140       if(updated_cell->outputs[i]!=NULL) {
141         if(updated_cell->outputs[i]->input_a ==
              updated_cell) {
142           if(updated_cell->outputs[i]->order_index>=canvas
                 ->ready_cells) {
143             if(ca_cell_check_conditions(updated_cell->
                    outputs[i])) {
144               canvas->ready_cells++;
145               ca_canvas_swap(canvas, canvas->ready_cells
                      -1, updated_cell->outputs[i]->order_index
                      );
146             }
147           }
148         }
149         if (updated_cell->outputs[i]->input_b ==
              updated_cell) {
150           if(updated_cell->outputs[i]->order_index>=canvas
                 ->ready_cells) {
151             if(ca_cell_check_conditions(updated_cell->
                    outputs[i])) {
152               canvas->ready_cells++;
153               ca_canvas_swap(canvas, canvas->ready_cells
                      -1, updated_cell->outputs[i]->order_index
                      );
154             }
155           }
156         }
157       }
158     }
159     if(updated_cell->input_a->order_index>=canvas->
          ready_cells) {
160       if(ca_cell_check_conditions(updated_cell->input_a))
            {
161         canvas->ready_cells++;
162         ca_canvas_swap(canvas, canvas->ready_cells-1,
              updated_cell->input_a->order_index);
163       }
```

```
164     }
165     if(updated_cell->input_b->order_index>=canvas->
           ready_cells) {
166       if(ca_cell_check_conditions(updated_cell->input_b))
             {
167         canvas->ready_cells++;
168         ca_canvas_swap(canvas, canvas->ready_cells-1,
              updated_cell->input_b->order_index);
169       }
170     }
171   }
172   ca_canvas_shuffle(canvas);
173   canvas->ca_time++;
174 #ifdef CA_DEBUG
175   if(canvas->ca_time % 100000 == 0) {
176     printf("ca_time:␣%ld\tunix␣time:␣%d\n", canvas->
         ca_time, (int)time(NULL));
177   }
178 #endif
179 }
180
181 void ca_clear_cell(struct ca_cell* cleared_cell) {
182   if(cleared_cell->input_a) {
183     cleared_cell->input_a->outputs[cleared_cell->
         input_a_output_idx]=cleared_cell->input_a->outputs
         [--cleared_cell->input_a->output_count];
184   }
185   if(cleared_cell->input_b) {
186     cleared_cell->input_b->outputs[cleared_cell->
         input_b_output_idx]=cleared_cell->input_b->outputs
         [--cleared_cell->input_b->output_count];
187   }
188
189   cleared_cell->drawable=0;
190   cleared_cell->function=0;
191   cleared_cell->input_a=NULL;
192   cleared_cell->input_b=NULL;
193   cleared_cell->input_a_state=0;
194   cleared_cell->input_b_state=0;
195   cleared_cell->input_a_output_idx=0;
196   cleared_cell->input_b_output_idx=0;
197   cleared_cell->latest_bit=0;
```

```
198   cleared_cell->latest_update=-cell_fade;
199 }
200
201 int ca_is_cell(struct ca_cell* cell) {
202   if(cell->drawable) {
203     return 1;
204   } else {
205     return 0;
206   }
207 }
208
209 void ca_canvas_clear(struct ca_canvas* canvas) {
210   int i;
211   memset(canvas->cells,0,sizeof(struct ca_cell)*canvas->
        width*canvas->height);
212   for (i=0;i<(canvas->width*canvas->height);i++) {
213     ca_clear_cell(&(canvas->cells[i]));
214     memset(&(canvas->cells[i].outputs),0,sizeof(struct
          ca_cell*[CELL_OUTPUTS]));
215     canvas->cells[canvas->update_order[i]].order_index=i;
216     canvas->cells[i].output_count=0;
217   }
218 }
219
220 struct ca_canvas ca_canvas_create(unsigned int width,
      unsigned int height) {
221   struct ca_canvas result;
222   int i;
223   result.width = width;
224   result.height = height;
225   result.ca_time = 0;
226   result.cells = (struct ca_cell*)malloc(width*height*
        sizeof(struct ca_cell));
227   result.update_order = (int*)malloc(width*height*sizeof(
        int));
228   result.draw_order = (int*)malloc(width*height*sizeof(int
        ));
229   result.ready_cells = width*height;
230   result.drawable_cells = 0;
231   for(i=width*height-1;i>=0;i--) {
232     result.update_order[i]=i;
233   }
```

```
234   ca_canvas_clear(&result);
235   return result;
236 }
237
238 int ca_index_translate(unsigned int width, unsigned int x,
      unsigned int y, int dir) {
239   switch(dir) {
240     case 0:
241       return (y*width)+(x+1);
242     case 1:
243       return (y+1)*width+(x+1);
244     case 2:
245       return (y+1)*width+x;
246     case 3:
247       return (y+1)*width+(x-1);
248     case 4:
249       return y*width+(x-1);
250     case 5:
251       return (y-1)*width+(x-1);
252     case 6:
253       return (y-1)*width+x;
254     case 7:
255       return (y-1)*width+(x+1);
256     default:
257       return y*width+x;
258   }
259 }
260
261 int ca_canvas_clear_cell(struct ca_canvas* canvas,
     unsigned int x, unsigned int y) {
262   if(x >= canvas->width || y >= canvas->height) {return
        1;}
263   if(!canvas->cells[(y*canvas->width)+x].drawable) {return
        0;}
264   canvas->draw_order[canvas->cells[(y*canvas->width)+x].
        draw_order_index]=canvas->draw_order[--canvas->
        drawable_cells];
265   ca_clear_cell(&(canvas->cells[(y*canvas->width)+x]));
266   return 0;
267 }
268
269 int ca_canvas_is_cell(struct ca_canvas* canvas, unsigned
```

```
     int x, unsigned int y) {
270   if(x >= canvas->width || y >= canvas->height) {return
        1;}
271   return ca_is_cell(&(canvas->cells[(y*canvas->width)+x]))
        ;
272 }
273
274 int ca_canvas_set_cell(struct ca_canvas* canvas, unsigned
      int x, unsigned int y, int func, int input_a_state, int
       input_b_state, int input_a, int input_b) {
275   struct ca_cell* new_cell = &(canvas->cells[y*(canvas->
        width)+x]);
276   struct ca_cell* neighbor_cell;
277   int translated_index, translated_index2;
278   if(x >= canvas->width || y >= canvas->height) {return
        1;}
279   new_cell->input_a_state = input_a_state;
280   new_cell->input_b_state = input_b_state;
281   new_cell->function = (char)func;
282   new_cell->drawable = 1;
283
284   canvas->draw_order[canvas->drawable_cells]=y*canvas->
        width+x;
285   new_cell->draw_order_index=canvas->drawable_cells;
286   canvas->drawable_cells++;
287
288   translated_index = ca_index_translate(canvas->width, x,
        y, input_a);
289   if(translated_index <= (canvas->width*canvas->height)) {
290     neighbor_cell=&(canvas->cells[translated_index]);
291     new_cell->input_a = neighbor_cell;
292     neighbor_cell->outputs[neighbor_cell->output_count]=
          new_cell;
293     new_cell->input_a_output_idx = neighbor_cell->
          output_count;
294     neighbor_cell->output_count++;
295   }
296
297   translated_index2 = ca_index_translate(canvas->width, x,
         y, input_b);
298   if(translated_index2 <= (canvas->width*canvas->height))
         {
```

```
299     neighbor_cell=&(canvas->cells[translated_index2]);
300     new_cell->input_b = neighbor_cell;
301     neighbor_cell->outputs[neighbor_cell->output_count]=
          new_cell;
302     new_cell->input_b_output_idx = neighbor_cell->
          output_count;
303     neighbor_cell->output_count++;
304   }
305
306   return 0;
307 }
308
309 void ca_canvas_print_states(struct ca_canvas* canvas) {
310   int i,j;
311   for(j=canvas->height-1; j>=0; j--) {
312     for(i=0; i<canvas->width; i++) {
313       struct ca_cell* ptr = &(canvas->cells[j*canvas->
            width+i]);
314       printf("%c␣", (ptr?((ptr->latest_bit)?'#':'.'):'␣'))
            ;
315     }
316     printf("\n");
317   }
318   printf("\n");
319 }
```

A.2 graphics.c

```
 1 #include <math.h>
 2 #include <GL/freeglut_std.h>
 3 #include <GL/freeglut_ext.h>
 4
 5 #define NAND1R 0.7411
 6 #define NAND1G 0.2705
 7 #define NAND1B 0.0156
 8 #define NAND0R 0.2000
 9 #define NAND0G 0.6941
10 #define NAND0B 0.7411
11 #define NANDXR 0.3372
12 #define NANDXG 0.2705
13 #define NANDXB 0.0156
14 #define XOR1R 0.7411
```

```
15 #define XOR1G 0.2039
16 #define XOR1B 0.3294
17 #define XOR0R 0.0431
18 #define XOR0G 0.5019
19 #define XOR0B 0.7411
20 #define XORXR 0.0431
21 #define XORXG 0.0156
22 #define XORXB 0.3372
23 #define OR1R 0.7450
24 #define OR1G 0.2549
25 #define OR1B 0.0156
26 #define OR0R 0.0666
27 #define OR0G 0.6941
28 #define OR0B 0.7372
29 #define ORXR 0.0823
30 #define ORXG 0.3372
31 #define ORXB 0.0156
32 #define AND1R 0.7411
33 #define AND1G 0.1764
34 #define AND1B 0.2000
35 #define AND0R 0.3215
36 #define AND0G 0.5019
37 #define AND0B 0.7411
38 #define ANDXR 0.3372
39 #define ANDXG 0.0156
40 #define ANDXB 0.0823
41 #define WIRE1R 1.0
42 #define WIRE1G 0.0
43 #define WIRE1B 0.0
44 #define WIRE0R 0.0
45 #define WIRE0G 0.5882
46 #define WIRE0B 1.0
47 #define WIREAR 1.0
48 #define WIREAG 1.0
49 #define WIREAB 1.0
50 #define WIREXR 0.0
51 #define WIREXG 0.0
52 #define WIREXB 0.0
53 #define BLACK 0,0,0
54 #define WHITE 1,1,1
55
56 unsigned int cell_size;
```

```
57 int and0,and1,or0,or1,xor0,xor1,nand0,nand1;
58 unsigned int draw_all = 1;
59
60 void ca_graphics_draw() {
61   int i;
62   static unsigned long int last_draw;
63   if(draw_all) {
64     glClear(GL_COLOR_BUFFER_BIT);
65   }
66   if(draw_all || last_draw < cell_fade) {
67     last_draw=cell_fade;
68   }
69   glBegin(GL_QUADS);
70   for(i=0; i<canvas.drawable_cells; i++) {
71     struct ca_cell* cell = &(canvas.cells[canvas.
         draw_order[i]]);
72     int x = canvas.draw_order[i] % canvas.width;
73     int y = canvas.draw_order[i] / canvas.width;
74     if(draw_all || (cell->latest_update >= last_draw -
         cell_fade)) {
75       double brightness = ((double)(cell_fade - (canvas.
           ca_time - cell->latest_update)))/((double)
           cell_fade);
76       if(!((canvas.ca_time - cell->latest_update) <=
           cell_fade)) brightness = 0.0;
77       switch(cell->function) {
78         case 0:
79           glColor3f(AND1R*brightness*cell->latest_bit+
               AND0R*brightness*(1-cell->latest_bit)+ANDXR
               *(1-brightness),AND1G*brightness*cell->
               latest_bit+AND0G*brightness*(1-cell->
               latest_bit)+ANDXG*(1-brightness),AND1B*
               brightness*cell->latest_bit+AND0B*brightness
               *(1-cell->latest_bit)+ANDXB*(1-brightness));
80           break;
81         case 1:
82           glColor3f(OR1R*brightness*cell->latest_bit+OR0R*
               brightness*(1-cell->latest_bit)+ORXR*(1-
               brightness),OR1G*brightness*cell->latest_bit+
               OR0G*brightness*(1-cell->latest_bit)+ORXG*(1-
               brightness),OR1B*brightness*cell->latest_bit+
               OR0B*brightness*(1-cell->latest_bit)+ORXB*(1-
```

```
            brightness));
83            break;
84          case 2:
85            glColor3f(XOR1R*brightness*cell->latest_bit+
                XOR0R*brightness*(1-cell->latest_bit)+XORXR
                *(1-brightness),XOR1G*brightness*cell->
                latest_bit+XOR0G*brightness*(1-cell->
                latest_bit)+XORXG*(1-brightness),XOR1B*
                brightness*cell->latest_bit+XOR0B*brightness
                *(1-cell->latest_bit)+XORXB*(1-brightness));
86            break;
87          case 3:
88            glColor3f(NAND1R*brightness*cell->latest_bit+
                NAND0R*brightness*(1-cell->latest_bit)+NANDXR
                *(1-brightness),NAND1G*brightness*cell->
                latest_bit+NAND0G*brightness*(1-cell->
                latest_bit)+NANDXG*(1-brightness),NAND1B*
                brightness*cell->latest_bit+NAND0B*brightness
                *(1-cell->latest_bit)+NANDXB*(1-brightness));
89            break;
90          default:
91            continue;
92        }
93        glVertex2i(x*cell_size, y*cell_size);
94        glVertex2i((x+1)*cell_size, y*cell_size);
95        glVertex2i((x+1)*cell_size, (y+1)*cell_size);
96        glVertex2i(x*cell_size, (y+1)*cell_size);
97      }
98    }
99    glEnd();
100   glBegin(GL_LINES);
101   for(i=0; i<canvas.drawable_cells; i++) {
102     struct ca_cell* cell = &(canvas.cells[canvas.
          draw_order[i]]);
103     int x = canvas.draw_order[i] % canvas.width;
104     int y = canvas.draw_order[i] / canvas.width;
105     if(draw_all || (cell->latest_update >= last_draw -
          cell_fade) || (cell->input_a && (cell->input_a->
          latest_update >= last_draw - cell_fade)) || (cell->
          input_b && (cell->input_b->latest_update >=
          last_draw - cell_fade))) {
106       struct ca_cell* ina = cell->input_a;
```

```
107       struct ca_cell* inb = cell->input_b;
108       int ina_d = ina - canvas.cells;
109       int inb_d = inb - canvas.cells;
110       int ina_x = (double)(((ina_d % canvas.width)*
             cell_size)+cell_size/2);
111       int ina_y = (double)(((ina_d / canvas.width)*
             cell_size)+cell_size/2);
112       int inb_x = (double)(((inb_d % canvas.width)*
             cell_size)+cell_size/2);
113       int inb_y = (double)(((inb_d / canvas.width)*
             cell_size)+cell_size/2);
114       int cur_x = (double)((x*cell_size)+cell_size/2);
115       int cur_y = (double)((y*cell_size)+cell_size/2);
116
117       int p1a_x = 0.25*ina_x+0.75*cur_x;
118       int p1a_y = 0.25*ina_y+0.75*cur_y;
119       int p2a_x = 0.75*ina_x+0.25*cur_x;
120       int p2a_y = 0.75*ina_y+0.25*cur_y;
121       int p1b_x = 0.25*inb_x+0.75*cur_x;
122       int p1b_y = 0.25*inb_y+0.75*cur_y;
123       int p2b_x = 0.75*inb_x+0.25*cur_x;
124       int p2b_y = 0.75*inb_y+0.25*cur_y;
125       int transa_x = (p1a_y-p2a_y);
126       int transa_y = (p2a_x-p1a_x);
127       int transb_x = (p1b_y-p2b_y);
128       int transb_y = (p2b_x-p1b_x);
129       double transa_length = (double)(sqrt(transa_x*
             transa_x+transa_y*transa_y))/((double)cell_size
             /6.0);
130       double transb_length = (double)(sqrt(transb_x*
             transb_x+transb_y*transb_y))/((double)cell_size
             /6.0);
131       transa_x = (double)(transa_x/transa_length);
132       transa_y = (double)(transa_y/transa_length);
133       transb_x = (double)(transb_x/transb_length);
134       transb_y = (double)(transb_y/transb_length);
135
136       if(cell->input_a_state==2) {
137         double brightness;
138         if(cell->input_a->latest_update >= (signed long
               int) canvas.ca_time - cell_fade) {
139           brightness = ((double)(cell_fade - (canvas.
```

```
            ca_time - cell->input_a->latest_update)))/((
            double)cell_fade);
140       } else {
141         brightness = 0;
142       }
143       glColor3f(WIREAR*brightness+WIRE0R*(1-brightness),
             WIREAG*brightness+WIRE0G*(1-brightness),WIREAB*
             brightness+WIRE0B*(1-brightness));
144       /*glVertex2d(p1a_x+1*transa_x, p1a_y+1*transa_y);
145       glVertex2d(p1a_x+3*transa_x, p1a_y+3*transa_y);*/
146     } else {
147       glColor3f(WIREXR,WIREXG,WIREXB);
148     }
149     glVertex2d(p1a_x+2*transa_x, p1a_y+2*transa_y);
150     glVertex2d(p2a_x+2*transa_x, p2a_y+2*transa_y);
151     if(cell->input_a_state==3) {
152       double brightness;
153       if(cell->input_a->latest_update >= (signed long
             int) canvas.ca_time - cell_fade) {
154         brightness = ((double)(cell_fade - (canvas.
             ca_time - cell->input_a->latest_update)))/((
             double)cell_fade);
155       } else {
156         brightness = 0;
157       }
158       glColor3f(WIREAR*brightness+WIRE1R*(1-brightness),
             WIREAG*brightness+WIRE1G*(1-brightness),WIREAB*
             brightness+WIRE1B*(1-brightness));
159       /*glVertex2d(p1a_x+1*transa_x, p1a_y+1*transa_y);
160       glVertex2d(p1a_x+3*transa_x, p1a_y+3*transa_y);*/
161     } else {
162       glColor3f(WIREXR,WIREXG,WIREXB);
163     }
164     glVertex2d(p1a_x+1*transa_x, p1a_y+1*transa_y);
165     glVertex2d(p2a_x+1*transa_x, p2a_y+1*transa_y);
166
167     if(cell->input_b_state==2) {
168       double brightness;
169       if(cell->input_b->latest_update >= (signed long
             int) canvas.ca_time - cell_fade) {
170         brightness = ((double)(cell_fade - (canvas.
             ca_time - cell->input_b->latest_update)))/((
```

```
                double)cell_fade);
171           } else {
172             brightness = 0;
173           }
174           glColor3f(WIREAR*brightness+WIRE0R*(1-brightness),
                 WIREAG*brightness+WIRE0G*(1-brightness),WIREAB*
                 brightness+WIRE0B*(1-brightness));
175           /*glVertex2d(p1a_x+1*transa_x, p1a_y+1*transa_y);
176           glVertex2d(p1a_x+3*transa_x, p1a_y+3*transa_y);*/
177         } else {
178           glColor3f(WIREXR,WIREXG,WIREXB);
179         }
180         glVertex2d(p1b_x+2*transb_x, p1b_y+2*transa_y);
181         glVertex2d(p2b_x+2*transb_x, p2b_y+2*transb_y);
182         if(cell->input_b_state==3) {
183           double brightness;
184           if(cell->input_b->latest_update >= (signed long
                 int) canvas.ca_time - cell_fade) {
185             brightness = ((double)(cell_fade - (canvas.
                   ca_time - cell->input_b->latest_update)))/((
                   double)cell_fade);
186           } else {
187             brightness = 0;
188           }
189           glColor3f(WIREAR*brightness+WIRE1R*(1-brightness),
                 WIREAG*brightness+WIRE1G*(1-brightness),WIREAB*
                 brightness+WIRE1B*(1-brightness));
190           /*glVertex2d(p1a_x+1*transa_x, p1a_y+1*transa_y);
191           glVertex2d(p1a_x+3*transa_x, p1a_y+3*transa_y);*/
192         } else {
193           glColor3f(WIREXR,WIREXG,WIREXB);
194         }
195         glVertex2d(p1b_x+1*transb_x, p1b_y+1*transb_y);
196         glVertex2d(p2b_x+1*transb_x, p2b_y+1*transb_y);
197       }
198     }
199     glEnd();
200     glFlush();
201     glutSwapBuffers();
202     last_draw = canvas.ca_time;
203     if(draw_all) draw_all--;
204 }
```

```
205
206 void ca_graphics_reshape(int w, int h) {
207   draw_all=5;
208   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
209   glLoadIdentity();
210   if (w <= h) {
211     gluOrtho2D(0,canvas.width*cell_size,canvas.height*
          cell_size*(1-(GLfloat)h/(GLfloat)w)/2,canvas.height
          *cell_size+canvas.height*cell_size*((GLfloat)h/(
          GLfloat)w-1)/2);
212   } else {
213     gluOrtho2D(canvas.width*cell_size*(1-(GLfloat)w/(
          GLfloat)h)/2,canvas.width*cell_size+canvas.width*
          cell_size*((GLfloat)w/(GLfloat)h-1)/2,0,canvas.
          height*cell_size);
214   }
215 }
216
217 void ca_graphics_init(unsigned int cell_size_) {
218   cell_size = cell_size_;
219   int glutArgc = 0;
220   glutInit(&glutArgc, NULL);
221   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
222   glutInitWindowSize(canvas.width*cell_size, canvas.height
        *cell_size);
223   glutCreateWindow("CA");
224   glClearColor(0.0,0.0,0.0,0.0);
225   gluOrtho2D(0,canvas.width*cell_size,0,canvas.height*
        cell_size);
226   glutDisplayFunc(ca_graphics_draw);
227   glutReshapeFunc(ca_graphics_reshape);
228   glutTimerFunc(msecs, timercb, 0);
229   glutSetOption(GLUT_ACTION_ON_WINDOW_CLOSE,
        GLUT_ACTION_GLUTMAINLOOP_RETURNS);
230   glutMainLoop();
231   glClear(GL_COLOR_BUFFER_BIT);
232 }
```

A.3 main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <libguile.h>
4 void timercb(int);
5 unsigned int msecs = 100;
6 #include "ca.c"
7 struct ca_canvas canvas;
8 #include "graphics.c"
9
10 SCM scm_canvas_init(SCM width, SCM height) {
11   canvas = ca_canvas_create(scm_to_uint(width),scm_to_uint
       (height));
12   ca_canvas_clear(&canvas);
13   return SCM_BOOL_T;
14 }
15
16 SCM scm_canvas_update(void) {
17   ca_canvas_update(&canvas);
18   return SCM_BOOL_T;
19 }
20
21 SCM scm_canvas_clear(void) {
22   ca_canvas_clear(&canvas);
23   return SCM_BOOL_T;
24 }
25
26 SCM scm_canvas_clear_cell(SCM x, SCM y) {
27   if(ca_canvas_clear_cell(&canvas, scm_to_uint(x),
       scm_to_uint(y))) {
28     return SCM_BOOL_F;
29   } else {
30     return SCM_BOOL_T;
31   }
32 }
33
34 SCM scm_canvas_set_cell(SCM x, SCM y, SCM func, SCM
     input_a_state, SCM input_b_state, SCM input_a, SCM
     input_b) {
35   if(ca_canvas_set_cell(&canvas, scm_to_uint(x),
       scm_to_uint(y), scm_to_int(func), scm_to_int(
```

```
      input_a_state), scm_to_int(input_b_state), scm_to_int
      (input_a), scm_to_int(input_b))) {
36      return SCM_BOOL_F;
37    } else {
38      return SCM_BOOL_T;
39    }
40  }
41
42  SCM scm_canvas_is_cell(SCM x, SCM y) {
43    if(ca_canvas_is_cell(&canvas, scm_to_uint(x),
        scm_to_uint(y))) {
44      return SCM_BOOL_T;
45    } else {
46      return SCM_BOOL_F;
47    }
48  }
49
50  SCM scm_canvas_print_states(void) {
51    ca_canvas_print_states(&canvas);
52    return SCM_BOOL_T;
53  }
54
55  SCM scm_set_shuffle_mix(SCM shuffle_mix_v) {
56    shuffle_mix = scm_to_uint(shuffle_mix_v);
57    return SCM_BOOL_T;
58  }
59
60  SCM scm_graphics_init(SCM cell_size, SCM cell_fade_v) {
61    cell_fade = scm_to_uint(cell_fade_v) + 2;
62    ca_graphics_init(scm_to_uint(cell_size));
63    return SCM_BOOL_T;
64  }
65
66  SCM scm_glut_main_loop_event(void) {
67    glutMainLoopEvent();
68    return SCM_BOOL_T;
69  }
70
71  SCM scm_graphics_draw() {
72    ca_graphics_draw();
73    return SCM_BOOL_T;
74  }
```

```
75
76 SCM scm_set_msecs(SCM msecs_val) {
77   msecs=scm_to_uint(msecs_val);
78   return msecs_val;
79 }
80
81 void timercb(int value) {
82   SCM idle_symbol = scm_c_lookup("timer");
83   SCM idle_func = scm_variable_ref(idle_symbol);
84   scm_call_0(idle_func);
85   glutTimerFunc(msecs,timercb,0);
86 }
87
88 int main (int argc, char *argv[])
89 {
90   if(argc != 2) {
91     printf("Usage: %s <ca file>\n", argv[0]);
92     return 0;
93   }
94   scm_init_guile();
95   scm_c_define_gsubr("canvas-init",2,0,0,scm_canvas_init);
96   scm_c_define_gsubr("canvas-update",0,0,0,
       scm_canvas_update);
97   scm_c_define_gsubr("canvas-clear",0,0,0,scm_canvas_clear
       );
98   scm_c_define_gsubr("canvas-clear-cell",2,0,0,
       scm_canvas_clear_cell);
99   scm_c_define_gsubr("canvas-is-cell",2,0,0,
       scm_canvas_is_cell);
100  scm_c_define_gsubr("canvas-set-cell",7,0,0,
       scm_canvas_set_cell);
101  scm_c_define_gsubr("canvas-print-states",0,0,0,
       scm_canvas_print_states);
102  scm_c_define_gsubr("set-msecs",1,0,0,scm_set_msecs);
103  scm_c_define_gsubr("set-shuffle-mix",1,0,0,
       scm_set_shuffle_mix);
104  scm_c_define_gsubr("graphics-init",2,0,0,
       scm_graphics_init);
105  scm_c_define_gsubr("graphics-draw",0,0,0,
       scm_graphics_draw);
106  scm_c_define_gsubr("glut-main-loop-event",0,0,0,
       scm_glut_main_loop_event);
```

```
107
108   srand(time(NULL));
109
110   scm_c_primitive_load(argv[1]);
111
112   return 0;
113 }
```

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

Example Input Code

B.1 smart-wire.scm

```
1 (load "pairing-heap.scm")
2 (load "sets.scm")
3
4 (define favor-diagonals-heuristic
5   (lambda (cur dest p)
6     (let* ((propx (caaar p))
7            (propy (cadaar p))
8            (curx (car cur))
9            (cury (cadr cur))
10           (destx (car dest))
11           (desty (cadr dest))
12           (square (lambda (n) (* n n)))
13           (pyth (lambda (x y) (sqrt (+ (square x) (square
                 y)))))
14           (dirx (- propx curx))
15           (diry (- propy cury))
16           (vecnorm (/ (pyth dirx diry) (sqrt 2)))
17           (ndirx (/ dirx vecnorm))
18           (ndiry (/ diry vecnorm))
19           (testx (+ curx ndirx))
20           (testy (+ cury ndiry)))
21       (+ (length p) (pyth (- testx destx) (- testy desty))
            ))))
22
23 (define direct-path-heuristic
```

```
24   (lambda (cur dest p)
25     (let* ((propx (caaar p))
26            (propy (cadaar p))
27            (destx (car dest))
28            (desty (cadr dest))
29            (square (lambda (n) (* n n)))
30            (pyth (lambda (x y) (sqrt (+ (square x) (square
                  y))))))
31       (+ (length p) (pyth (- propx destx) (- propy desty))
          ))))
32
33 (define cell-crowding
34   (lambda (x y)
35     (count (lambda (p) (and (>= (car p) 0) (>= (cadr p) 0)
          (< (car p) canvas-width) (< (cadr p) canvas-height
        ) (canvas-is-cell (car p) (cadr p))))
36       `((,(1- x)     ,y )
37         ( ,(- x 2)   ,y )
38         (,(1- x) ,(1- y))
39         (    ,x  ,(1- y))
40         (    ,x   ,(- y 2))
41         (,(1+ x) ,(1- y))
42         (,(1+ x)     ,y )
43         (,(+ x 2)    ,y )
44         (,(1+ x) ,(1+ y))
45         (    ,x  ,(1+ y))
46         (    ,x   ,(+ y 2))
47         (,(1- x) ,(1+ y))))))
48
49 (define uncrowded-cell-heuristic
50   (lambda (weight)
51     (lambda (cur dest p)
52       (let* ((propx (caaar p))
53              (propy (cadaar p))
54              (destx (car dest))
55              (desty (cadr dest))
56              (square (lambda (n) (* n n)))
57              (pyth (lambda (x y) (sqrt (+ (square x) (
                    square y))))))
58         (+ (* (cell-crowding propx propy) weight) (length
              p) (pyth (- propx destx) (- propy desty)))))))
59
```

```
60 (define favor-uncrowded-diagonals-heuristic
61   (lambda (weight1 weight2)
62     (lambda (cur dest p)
63       (let* ((propx (caaar p))
64              (propy (cadaar p))
65              (curx (car cur))
66              (cury (cadr cur))
67              (destx (car dest))
68              (desty (cadr dest))
69              (square (lambda (n) (* n n)))
70              (pyth (lambda (x y) (sqrt (+ (square x) (
                   square y)))))
71              (dirx (- propx curx))
72              (diry (- propy cury))
73              (vecnorm (pyth dirx diry)))
74         (+ (* (cell-crowding propx propy) weight1) (/
              weight2 vecnorm) (length p) (pyth (- propx
              destx) (- propy desty)))))))
75
76 (define smart-wire
77   (lambda (src dest heuristic)
78     (let* ((closed set-empty)
79            (final-path '())
80            (neighbors (lambda (p)
81                         (let ((x (caaar p))
82                               (y (cadaar p)))
83                           (list
84                             (cons `((,(1- x)     ,y ) 0) p
                                 )
85                             (cons `((,(1- x) ,(1- y)) 1) p
                                 )
86                             (cons `((    ,x  ,(1- y)) 2) p
                                 )
87                             (cons `((,(1+ x) ,(1- y)) 3) p
                                 )
88                             (cons `((,(1+ x)     ,y ) 4) p
                                 )
89                             (cons `((,(1+ x) ,(1+ y)) 5) p
                                 )
90                             (cons `((    ,x  ,(1+ y)) 6) p
                                 )
91                             (cons `((,(1- x) ,(1+ y)) 7) p
```

```
                                  )))))
92            (pq-comp (lambda (a b) (<= (car a) (car b))))
93            (init-path `((,src 0)))
94            (queue ((unit-pq pq-comp) (list (heuristic src
                 dest init-path) init-path))))
95       (while (and (not (pq-empty? queue)) (null?
            final-path))
96              (let* ((p (cadr ((pq-min pq-comp) queue)))
97                     (x (caar p)))
98                (set! queue ((pq-remove-min pq-comp) queue)
                     )
99                (if (not (member x closed))
100                 (if (and (>= (car x) 0) (>= (cadr x) 0)
                       (< (car x) canvas-width) (< (cadr x)
                       canvas-height) (or (not (
                       canvas-is-cell (car x) (cadr x)) ) (
                       equal? x src)))
101                    (if (equal? x dest)
102                      (set! final-path p)
103                      (begin
104                        (set! closed ((set-adjoin equal?) x
                             closed))
105                        (for-each (lambda (succ) (set!
                             queue ((pq-insert pq-comp) queue
                              (list (heuristic x dest succ)
                             succ))))
106                                  (neighbors p))))))))
107       (if (null? final-path)
108         (error (list "couldn't␣find␣path␣from" src "to"
              dest)) (dumb-wire final-path)))))
109
110 (define dumb-wire
111   (lambda (path)
112     (cond ((null? path) '())
113           ((null? (cdr path)) '())
114           (#t (let ((cell (car path)))
115                 (begin (canvas-set-cell (caar cell) (cadar
                      cell) 1 0 0 (cadr cell) (cadr cell))
116                    (dumb-wire (cdr path)))))))))
```

B.2 lfsr.scm

```
1 (use-modules (srfi srfi-1))
2
3 (define canvas-width 100)
4 (define canvas-height 100)
5
6 (defmacro ring (x1 y1 x2 y2)
7   `(begin
8      (do ((i 1 (1+ i)))
9          ((> i ,(- x2 x1)))
10         (begin
11           (canvas-set-cell (+ ,x1 i) ,y1 1 3 3 4 4)
12           (canvas-set-cell (- ,x2 i) ,y2 1 3 3 0 0)))
13     (do ((j 1 (1+ j)))
14         ((> j ,(- y2 y1)))
15         (begin
16           (canvas-set-cell ,x2 (+ ,y1 j) 1 3 3 6 6)
17           (canvas-set-cell ,x1 (- ,y2 j) 1 3 3 2 2)))))
18
19 (load "smart-wire.scm")
20
21 (define timer (lambda ()
22   (let lp ((count 100))
23     (if (= count 0)
24       (graphics-draw)
25       (begin
26         (canvas-update)
27         (lp (1- count)))))))
28
29 (set-msecs 16)
30
31 (canvas-init canvas-width canvas-height)
32
33 (ring 0 0 99 99)
34 (define clusterx 50)
35 (define clustery 53)
36 (canvas-set-cell clusterx clustery 2 0 0 2 3)
37 (canvas-set-cell clusterx (- clustery 1) 2 0 0 2 1)
38 (canvas-set-cell clusterx (- clustery 2) 2 0 0 2 3)
39 (canvas-set-cell clusterx (- clustery 3) 2 0 0 2 1)
40 (canvas-set-cell clusterx (- clustery 4) 2 0 0 2 3)
```

```
41
42 (define my-heuristic (favor-uncrowded-diagonals-heuristic
     3 2))
43 ;(define my-heuristic direct-path-heuristic)
44 (smart-wire '(99 60) `(,(+ clusterx 1) ,(- clustery 2))
     my-heuristic)
45 (smart-wire '(30 99) `(,clusterx ,(+ clustery 1))
     my-heuristic)
46 (smart-wire '(70 99) `(,(+ clusterx 1) ,clustery)
     my-heuristic)
47 (smart-wire '(0 30) `(,(- clusterx 1) ,(- clustery 1))
     my-heuristic)
48 (smart-wire '(0 70) `(,(- clusterx 1) ,(+ clustery 1))
     my-heuristic)
49 (smart-wire '(49 0) `(,(- clusterx 1) ,(- clustery 3))
     my-heuristic)
50 (canvas-clear-cell 50 0)
51 (smart-wire `(,clusterx ,(- clustery 4)) '(50 0)
     my-heuristic)
52
53 (set-shuffle-mix 1000)
54 (graphics-init 12 1500)
```

B.3 ring.scm

```
1 (defmacro repeat (times action)
2   `(do ((i 0 (1+ i)))
3        ((>= i ,times))
4        ,action))
5
6 (defmacro ignore (body) #t)
7
8 (defmacro ring (x1 y1 x2 y2)
9   `(begin
10      (do ((i 0 (1+ i)))
11          ((>= i ,(- x2 x1)))
12          (begin
13            (canvas-set-cell (+ ,x1 i) ,y1 1 (if (= i 0) 3
                 0) (if (= i 0) 3 0) 0 0)
14            (canvas-set-cell (- ,x2 i) ,y2 1 0 0 4 4)))
15      (do ((j 0 (1+ j)))
16          ((>= j ,(- y2 y1)))
```

```
17          (begin
18            (canvas-set-cell ,x2 (+ ,y1 j) 1 0 0 2 2)
19            (canvas-set-cell ,x1 (- ,y2 j) 1 0 0 6 6)))))
20
21 (define frameloop-count 0)
22
23 (define timer (lambda ()
24   (canvas-update)
25   (set! frameloop-count (1+ frameloop-count))
26   (if (> frameloop-count 100)
27     (begin
28       (graphics-draw)
29       (set! frameloop-count 0)))))
30
31 (set-msecs 0)
32 (set-shuffle-mix 0)
33
34 (canvas-init 2000 2000)
35 (ring 0 0 1998 1999)
36 (graphics-init 1 2500)
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald J. Sussman, and Ron Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, 2000.

[2] Vikrant Agnihotri. Towards in-situ device fabrication: electrostatic lithography and nanowire field effect devices. Master's thesis, Massachusetts Institute of Technology, 2005.

[3] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[4] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[5] Edwin Roger Banks. Cellular Automata. Technical Report AIM-198, MIT, June 1970.

[6] G. Baudet and D. Stevenson. Optimal Sorting Algorithms for Parallel Computers. *IEEE Transactions on Computers*, C-27(1):84–87, 1978.

[7] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *SPAA '91: Proceedings of the third annual ACM symposium on Parallel Algorithms and Architectures*, pages 3–16, New York, NY, 1991. ACM.

[8] William Joseph Butera. *Programming a paintable computer*. PhD thesis, 2002. Supervisor-V. Michael Bove, Jr.

[9] Yong Chen, Gun-Young Jung, Douglas A. Ohlberg, Xuema Li, Duncan R. Stewart, Jan O. Jeppesen, Kent A. Nielsen, Fraser J. Stoddart, and Stanley R. Williams. Nanoscale molecular-switch crossbar circuits. *Nanotechnology*, 14(4):462–468, 2003.

[10] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1), 2004.

[11] Jack B. Dennis. Modular, asynchronous control structures for a high performance processor. pages 55–80, 1970.

[12] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, April 1982.

[13] Uriel Frisch, Dominique d'Humières, Brasl Hasslacher, Pierre Lallemand, Yves Pomeau, and Jean-Pierre Rivet. Lattice gas hydrodynamics in two and three dimensions. In *Lattice-Gas Methods for Partial Differential Equations*, pages 77–135. Addison-Wesley, 1990.

[14] D. Geer. Is it time for clockless chips? *Computer*, 38(3):18–21, March 2005.

[15] Neil A. Gershenfeld and Isaac L. Chuang. Bulk spin-resonance quantum computation. *Science*, 275(5298):350–356, January 1997.

[16] Neil A. Gershenfield. *When Things Start to Think*. Henry Holt and Co., January 1999.

[17] V. Gutnik and A.P. Chandrakasan. Active ghz clock network using distributed plls. *Solid-State Circuits, IEEE Journal of*, 35(11):1553–1560, Nov 2000.

[18] M. Hénon. On the relation between lattice gases and cellular automata. In *Discrete Kinetic Theory, Lattice Gas Dynamics, and Foundations of Hydrodynamics*, pages 160–161. World Scientific, 1989.

[19] W. Daniel Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.

[20] W. H. Kautz, K. N. Levitt, and A. Waksman. Cellular interconnection arrays. *IEEE Trans. Comput.*, 17(5):443–451, 1968.

[21] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching*, volume 3. Addison-Wesley Professional, 2 edition, April 1998.

[22] Jacob Kornerup. Odd-even sort in powerlists. *Information Processing Letters*, 61(1):15–24, 1997.

[23] Ray Kurzweil. *The Singularity Is Near : When Humans Transcend Biology*. Viking Adult, September 2005.

[24] Frank Blase Manning. Automatic test, configuration, and repair of cellular arrays. Master's thesis, Massachusetts Institute of Technology, 1975.

[25] Norman Margolus. An embedded dram architecture for large-scale spatial-lattice computations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 149–160, New York, NY, USA, 2000. ACM.

[26] R. C. Minnick. Cutpoint cellular logic. *IEEE Transactions on Electronic Computers*, EC-13(6):685–698, December 1964.

[27] Marvin Minsky. *The Emotion Machine*. Simon and Schuster, 2006.

[28] David Misunas. Petri nets and speed independent design. *Commun. ACM*, 16(8):474–481, 1973.

[29] T. Murata. State equation, controllability, and maximal matchings of petri nets. *IEEE Transactions on Automatic Control*, 22(3):412–416, 1977.

[30] F. Peper, Jia Lee, F. Abo, T. Isokawa, S. Adachi, N. Matsui, and S. Mashiko. Fault-tolerance in nanocomputers: a cellular array approach. *Nanotechnology, IEEE Transactions on*, 3(1):187–201, March 2004.

[31] C. A. Petri. Nets, time and space. *Theoretical Computer Science*, 153(1-2):3–48, January 1996.

[32] Manu Prakash and Neil Gershenfeld. Microfluidic bubble logic. *Science*, 315(5813):832–835, February 2007.

[33] Ronny Ronen, Avi Mendelson, Konrad Lai, Shih L. Lu, Fred Pollack, and John P. Shen. Coming challenges in microarchitecture and architecture. *Proceedings of the IEEE*, 89(3):325–340, 2001.

[34] A. W. Roscoe and C. A. R. Hoare. The laws of Occam programming. *Theoretical Computer Science*, 60(2):177–229, September 1988.

[35] S. N. Salloum and A. L. Perrie. Fault tolerance analysis of odd-even transposition sorting networks. In *proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 155–157, 1999.

[36] F.L.J. Sangster and K. Teer. Bucket-brigade electronics: new possibilities for delay, time-axis conversion, and scanning. *Solid-State Circuits, IEEE Journal of*, 4(3):131–136, Jun 1969.

[37] R. G. Shoup. *Programmable Cellular Logic Arrays*. PhD thesis, Carnegie Mellon University, 1970.

[38] Alvay Ray Smith. *Cellular Automata Theory*. PhD thesis, Stanford University, 1970.

[39] Francois-Xavier Standaert, Gilles Piret, Neil Gershenfeld, and Jean-Jacques Quisquater. SEA: a scalable encryption algorithm for small embedded applications. In *proceedings of the ECRYPT Workshop on RFID and Lightweight Crypto*, Graz, Austria, July 2005.

[40] G. L. Steele and W. D. Hillis. *Connection Machine Lisp: fine-grained parallel symbolic processing*. ACM Press, 1986.

[41] Guy L. Steele, Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, March 2007.

[42] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and Stanley R. Williams. The missing memristor found. *Nature*, 453(7191):80–83.

[43] Tommaso Toffoli, Silvio Capobianco, and Patrizia Mentrasti. When – and how – can a cellular automaton be rewritten as a lattice gas?, September 2007.

[44] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling (Scientific Computation)*. The MIT Press, April 1987.

[45] John von Neumann. *Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components*, pages 43–98. Princeton University Press.

[46] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[47] John von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[48] Tony Werner and Venkatesh Akella. Asynchronous processor survey. *Computer*, 30(11):67–76, 1997.