ALA ASIC: A Standard Cell Library for Asynchronous Logic Automata

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
AUG 24 2010
LIBRARIES

by

Forrest Oliver Reece Green

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

ARCHIVES

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

© Massachusetts Institute of Technology 2010. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
March 31, 2010

Certified by..
Neil Gershenfeld
Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by ....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

ALA ASIC: A Standard Cell Library for Asynchronous Logic Automata

by

Forrest Oliver Reece Green

Submitted to the Department of Electrical Engineering and Computer Science
on March 31, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis demonstrates a hardware library with related tools and designs for Asynchronous Logic Automata (ALA) gates in a generic 90nm process development kit that allows a direct one-to-one mapping from software to hardware. Included are basic design tools to enable writing ALA software, the necessary hardware designs for implementation, and simulation techniques for quickly verifying correctness and performance. This thesis also documents many of the hazards and opportunities for improving them including helpful variations to the ALA model, design tool needs, better simulation models, and hardware improvements.

To embody software you could compile a hardware description language to an FPGA or synthesize it all the way to transistors. Alternatively, you could use your favorite high level language and run it on a standard processor. However, the widening gap between traditional models of computation and the reality of the underlying hardware has led to massive costs for design and fabrication as well as numerous issues for scalability and portability. Unlike any of these other approaches, ALA aligns computational and physical descriptions making it possible to use a direct one-to-one mapping to convert an ALA program to a circuit or other physical artifact that executes that program. No unpredictable fitters or compilers are needed and no extra expertise is needed for specific technologies. Similar to Mead-Conway design rules ALA designs trade flexibility for portability and ease of design. Unlike Mead-Conway design rules, ALA designs do not require any further verification—the design rule primitives are logical operations suitable for use in analysis at the algorithmic level. ALA separates many of the scaling issues that plague integrated circuit design by cleanly separating algorithm design from hardware engineering—improving design verification, tape-out costs (by reusing masks), yield, portability, and the ability to break designs across multiple chips. ALA designs are not limited to integrated circuits and could just as easily be implemented in microfluidics, magnetic logic, or a lattice of molecular logic gates. Although each of these technologies would require implementing a basic set of gates and tiling rules, hardware (or equivalently software) can be developed using the same deterministic noiseless digital abstraction using the same design in many different technologies.

Thesis Supervisor: Neil Gershenfeld
Title: Professor of Media Arts and Sciences

Acknowledgments

This work was supported by the MIT Center for Bits and Atoms and the U.S. Army Research Office under grant number W911NF-09-1-0542.

I would like to express my appreciation for the support of MIT's Center for Bits and Atoms and its sponsors and the administrative and academic staff at MIT that have made it possible to be here.

I thank Neil Gershenfeld, my thesis advisor, for his vision and advice. In directing the Center for Bits and Atoms, he has created a challenging and rewarding environment which has nurtured the ideas in this thesis and many others. His guidance has been invaluable in bringing this work to fruition.

I thank Ara Knaian for taking me on as a UROP and introducing me to the Center for Bits and Atoms which ultimately enabled me to become a part of this amazing team.

I thank the other members of the RALA team including Jonathan Bachrach, Kailiang Chen, David Dalrymple, Erik Demaine, Scott Greenwald, Bernhard Haeupler, Ara Knaian, and, Peter Schmidt-Nielsen as well as the members of the larger PHM group for their collaboration, contributions and camaraderie. Without your conversations, assistance, and late night food orders I doubt I would have made it through this process.

I thank my friends here at MIT and back in Texas for making sure that my research wasn't the only interesting part of my life.

I thank the many professors and TAs here at MIT that have equipped me with many of the skills and experiences that have been crucial to my academic career. In particular I would like to thank 6.004, 6.374, 6.375, and 6.376 for key insights into circuit design and computer architectures as well as 6.115, 6.141, MAS.863, and MASLAB for teaching me how to build stuff.

I would like to thank my parents better than I know how to in words: Thanks guys! I love you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many computational tools, with applications ranging from computer graphics to cryptography, travel the road from clunky prototype to miniaturized commodity hardware. In this process they can be implemented in an easy to distribute fashion as a program for a desktop computers, altered to run in specialized but easy to acquire customizable hardware, such as field programmable gate arrays (FPGAs), and reimplemented and fabricated as application-specific integrated circuits (ASICs). Though the development and deployment costs increase with specialization of the hardware, the performance benefits can be staggering. Unfortunately efficient implementations for each class of hardware typically require substantial redesign to cater to the quirks of the technology. Asynchronous Logic Automata (ALA) is a new model of computation that attempts to approximate the physical constraints common to all possible hardware into single easy to manipulate abstraction. The basic idea (detailed in chapter 2) is to allow the designer to place a network of simple processing elements in a physical space and connect them together. Forcing the designer to maintain physically realistic spatial relationships ensures that it will be possible to generate a hardware implementation regardless of the underlying technology—working with everything from reconfigurable hardware, to custom integrated circuits.

Even if hardware expertise is available, portability can still be an issue. In theory, a design done for one integrated circuit (IC) process using Mead-Conway design rules, can still be used after feature scaling and other improvements. In practice, changes in gate carrier mobility, interconnect resistance, relative capacitances, etc. can easily upset the delicate balances in a complex IC design. Simply scaling designs will not take full advantage of things like more metal layers and variable threshold transistors or handle the issues that arise in deep sub-micron technologies. This thesis demonstrates a standard cell library implemented in a generic 90nm process development kit of ALA cells, a design abstraction that allows a direct one-to-one mapping from software to hardware that is portable to different processes scales as well as radically different hardware such as microfluidics, magnetic logic, or even technologies not yet predicted. This mapping from software to hardware partitions algorithmic design issues from technology specific engineering details so that the design of application-specific integrated circuits (ASICs) can be done portably without regard to the underling process technology. Although there are an assortment of possible high level tools and languages to improve performance and ensure correctness of the algorithms, hardware synthesis of ALA designs is simply an assignment of tiles to locations and requires no compilers, fitters, or post-design verification.

ALA is based on the assumption that storing information requires a minimum energy per bit and that the universe allows only a finite energy density (and thus information density) and only local interactions. It excludes the potential for quantum-coherent computations as such mechanisms are currently impractical and poorly understood. It also assumes that computation and communication consume power. While charge conserving techniques based on reversible computing have been used to reduce power consumption (for example [18, 22]) they are not considered in the ALA model. When creating fully reversible programs with predicted or current technology (gigahertz frequency CMOS for example) it is unclear that it is possible to achieve high enough Q factors that resetting a computation by reversing it would be more efficient than simply dissipating a cycle's worth of power. It may be possible to use charge conserving techniques to improve power consumption in the cells (a topic of current research), but this would be a hardware optimization rather than a change to the programming model. Finally, it assumes that it is possible to use external entropy sources or pseudo random number generators if probabilistic behavior is needed.

The ALA model has been engineered rather than derived. Although finite information density and local interactions are fundamental properties of physics (perhaps excluding certain quantum phenomena), ALA is far from the only possible model that respects those. It would be possible to add or remove some cell types or make individual cells more complex or simpler while still respecting the fundamental design goals of ALA, however through trial and error we have found what we believe is a good balance between simplicity of the cells (to allow portability to other technologies), ease of design (by providing logical operations as the primitives), and fidelity to the underlying fabrication capabilities (to improve efficiency). Many of the similarities between ALA and other models of computation come from using the same approach for a given trade off. Although the choice of rules is not absolute, there are important properties that must be maintained to ensure correct operation and to respect the physics.

When we use a specific technology to design a computer we limit our choice of building blocks in exchange for simpler abstractions and easier fabrication. You can build a computer out of transistors in silicon, microfluidic channels [15] in glass, or tinker toys and strings [9]. For each technology, a reasonable approach is to start with a library of basic components (transistors and wires, different shapes of channels, or memory spindles and output ducks) and use those to build up desired functions. However, by the time we write algorithms for these computers, we are many layers of abstraction from the underlying physics. While these abstractions are necessary to mantain the sanity of the programmers, they can make it difficult to write efficient programs. For the von Neumann programming model, hardware designers have worked hard to preserve the fiction of fast sequential operation with constant time access to all memory, but for a modern desktop this is far from true. Caching, branch prediction, parallel dispatch, speculative execution, register renaming, pipeline stages, and the myriad of other techniques in the processor architect's tool bag make it nearly impossible to predict when, in a given program, it will proceed immediately to the next instruction or spend tens of millions of cycles swapping in a new page of memory from disk. ALA attempts to resolve this issue by exposing the same building blocks to the programmers that are available to the hardware designers. While there are necessarily approximations to ensure usability, care has been taken to ensure that these approximations don't limit the programmers ability to predict the final performance of their algorithm. Many of the same techniques that we see in processor architectures would likely be components in higher level ALA descriptions.

Presented in this thesis are a set of designs and tools for creating ASICs from ALA designs including:

1. Several simple ALA modules as well as a minimal programming library for expressing hierarchical ALA designs
2. Circuit designs for a library of ALA cells that can be tiled to generate an ASIC layout
3. A graphical simulator for ALA that does performance modeling orders of magnitude faster than general techniques
4. Suggestions for several alternative designs and possible improvements

1.1 Historical Background

The foundations of most modern computer architectures can be traced back to work started by John von Neumann in the 1940s [20] and to this day the majority of computers use the same basic principle of a single active processing element connected to a large bank of passive memory. His later work on Cellular Automata (CA) was inspired by his interest in self reproducing machines, but in the process he demonstrated the possibility of computation in CAs. Later work by Roger Banks [2], published in 1971, demonstrated universal computation with simpler symmetric rules. Of the many other CAs that have been shown to be universal, one of my personal favorites is John Conway's Game of Life [10], which was a major inspiration for my initial interest in computer science. While CAs were not attractive for early computers, the development and scaling of integrated circuits changed many of the engineering constraints. Instead of expensive, power-hungry vacuum tubes connected by fast, cheap and nearly arbitrary wiring, designs were using cheap and power-efficient transistors with increasingly difficult routing and wire delay issues. Unfortunately, the von Neumann processor architecture, which is dominant even today, assumes that interconnect is free and has no way of expressing spatial constraints. The need for new computing abstractions was recognized early in the 1980s [8] and inspired dataflow machines, the CAM-8, and the Connection Machine among others. In many cases, their lack of success arguably stemmed from non-technical issues [17]. The same scaling trends that began to make these alternate architectures favorable have continued to progress. Even mainstream hardware has had a distinct trend towards more parallel computing, but progress has been slow on better parallel programming abstractions and nearly nonexistent on spatial computing models. At today's gigahertz clock speeds, speed of light limits can be significant even over centimeter distances. The continued scaling of process technologies and clock speeds will make spatial constraints significant well below the level of a chip. There are predictions [1] that in deep submicron designs less than 1% of a chips area will be reachable in a single cycle. These trends have lead to my research group's interest in physically based models of computation. Notable prior work in which ALA was developed includes [3, 6, 7, 11].

1.2 Relationship of ALA to Other Models

ALA has many similarities with other models of computation. This is partly from convergent engineering under similar constraints and partly from using good ideas where they are compatible. In this section we discuss similarities and differences with selected models. More details on many of these relationships and others can be found in [7]

1.2.1 RISC Processor Architectures

ALA can be considered as an extrapolation of the current trend for multi-core RISC architectures to its limit as an infinite array of single bit processors with a fixed program. Unlike a single processor, which becomes progressively more difficult to scale up in speed, ALA performance scales by spreading computation over more cells. ALA designs work at a fine enough granularity that many of the tricks that could be used to speed up a traditional processor, like pipelining or speculative execution, can be implemented above the ALA abstraction. This makes it possible to integrate these techniques where they will help without being forced to pay their overhead in all cases.

1.2.2 Asynchronous Logic

Asynchronous logic is not a new concept, but making use of it often involves specific expertise and significant caution to ensure correct operation. The careful choice of ALA gates and design rules encapsulates all of the non-deterministic behavior to ensure that designers do not need to worry about rare edge cases but can still have most of the benefits to power from only consuming dynamic power where useful computation is taking place and robustness to power supply or process variations.

1.2.3 Petri Nets

Although similar to Petri nets, in that cell updates are activated by the presence of tokens on their inputs and are allowed to happen in any order, the cell update rules have been carefully restricted to enforce deterministic results regardless of the firing order. It is worth noting that there are some situations where this is not beneficial, but a possible fix for this is described in section 6.3.

1.2.4 CA Logic

Arbitrary computation has been demonstrated in many different cellular automatons (notably [2], [19], [5]); however these typically take dozens to hundreds of cells running for many generations to perform even a single logical operation or only work for very specific kinds of computation (i.e. [14]). In ALA the primitive rules are chosen to try and minimize the cost of implementing practical systems by allowing a single cell in one step to perform a logical operation. This is a significant benefit both for simplicity of creating designs from the ALA gates as well as flexibility in implementing the underlying hardware. Finally, the global clock needed in synchronous CAs isn't physically realistic and is avoided by using ALA without the complications that arise in other asynchronous CAs such as [13].

1.2.5 Wavefront and Systolic Arrays

ALA also has a strong resemblance to wavefront and systolic array processors, but there are no restrictions on how information flows through the system allowing arbitrary classes of computation with irregular data dependencies.

1.2.6 FPGA and Sea of Gates Designs

ALA design bears a strong resemblance to digital circuit design and in particular to sea of gate design styles. Similarly, the reconfigurable version of ALA (RALA) resembles FPGAs. However, ALA designs have a direct one-to-one mapping to the hardware that does not depend on fitters and their synthesis capabilities. Even for experienced users it can be difficult to account for the impact of fitter behavior on a design. Unlike in an FPGA, where the capabilities are often chosen based on what is efficient to implement in a specific technology, the rules for ALA are chosen to ensure that they can be implemented in many different technologies. This makes ALA designs substantially more likely to be portable. Section 4.4.1 compares a very simple design done in ALA and with standard synchronous gates. Although the custom design has better performance, after accounting for improvements from chapter 6's designs and clock generation and distribution costs, the ALA design would likely be competitive with traditional FPGA and sea of gate implementations.

1.2.7 Dataflow Architectures

ALA gates are dataflow architectures in that computation is triggered by the flow of information except the scaling and scheduling issues that plagued dataflow architectures are solved by limiting cells to local interconnect and allowing the designer to choose how to handle that constraint.

Chapter 2

The ALA Software Model

ALA is a programming model based on cellular automata, which consists of a regular grid of cells. Each cell has a permanent configuration or 'gate' that controls how it will pass and accept chunks of information or 'tokens' from adjacent neighbors. Different gates can pass tokens unchanged, fanout tokens, compute logic functions, or create and destroy tokens. Gates are analogous to the primitive instructions on a processor. Unlike a petri net, tokens contain a small amount of internal information on which the logic functions are computed or behaviors are controlled. Gates do not operate in lock step and can operate at different speeds, however their update rules wait until enough tokens have arrived to produce a deterministic result.

This chapter focuses on a specific ALA programming model describing the particular gates that were developed and how they work. This chapter also discusses some of the motivations for choosing this specific set of gates.

2.1 An ALA Gate Set

The gates used here are very similar to those used in [3,6,11,12] but have been revised slightly to make the hardware simpler and take advantage of things that were easy to implement in CMOS. The variations used here likely bring the rules closer to a lowest common denominator for many representative technologies. Slightly different optimizations may be possible in other technologies (microfluidics, magnetic logic, etc.) however it is difficult to predict exactly how they would differ before doing those designs. Comparing possible optimizations across a broad spectrum of technologies remains a topic for future work.

These ALA designs use a rectangular 2D grid of cells that can be set as one of ten different gates with outputs going to adjacent neighbors (not including diagonals). Unlike the reconfigurable variant described in [11], the cell configuration is hardwired and cannot be changed short of making a new chip. Cells communicate by passing boolean tokens to each other that can be either true (T) or false (F). The output of a cell doesn't need to have a token at all times and can be empty (X). Inputs and outputs to an ALA program are done by special cells that exchange tokens with the outside world. One could imagine having pixels in a display or some kind of sensor at regular intervals throughout the ALA grid or a network connection attached at a specific point. Figure 2.1 shows the symbols for tokens and the set of gates we implemented. The basic operation of all of the cells is relatively uniform throughout. Each cell waits until all of the needed input tokens have arrived from neighbors, consumes them, and emits new output tokens. In the CMOS implementation, inputs are consumed by signaling over an acknowledge wire connected to the cell that emitted that token which then clears its output. For a microfluidic implementation, input tokens might literally be physically consumed. The implementation used here differs from those described in [6,11,12] in that cells may produce outputs before all the inputs arrive if it is possible to deduce what the output token will necessarily be. For example if one of the inputs to an AND gate is an F (false) token, then the cell may produce an output of F before the other input arrives. This is called short circuiting. This is more powerful than previous versions of ALA because any design that would have worked with older versions will still produce the same outputs but not necessarily as quickly. This can allow some designs to function which would have stalled indefinitely without short circuiting. This is unlikely to be a problem unless the design was connected to an external peripheral which was time sensitive. Arguably, this is a generalization where cells are allowed to take on faith that the other input would eventually arrive. This idea can be further extended by allowing acknowledge signals to propagate up

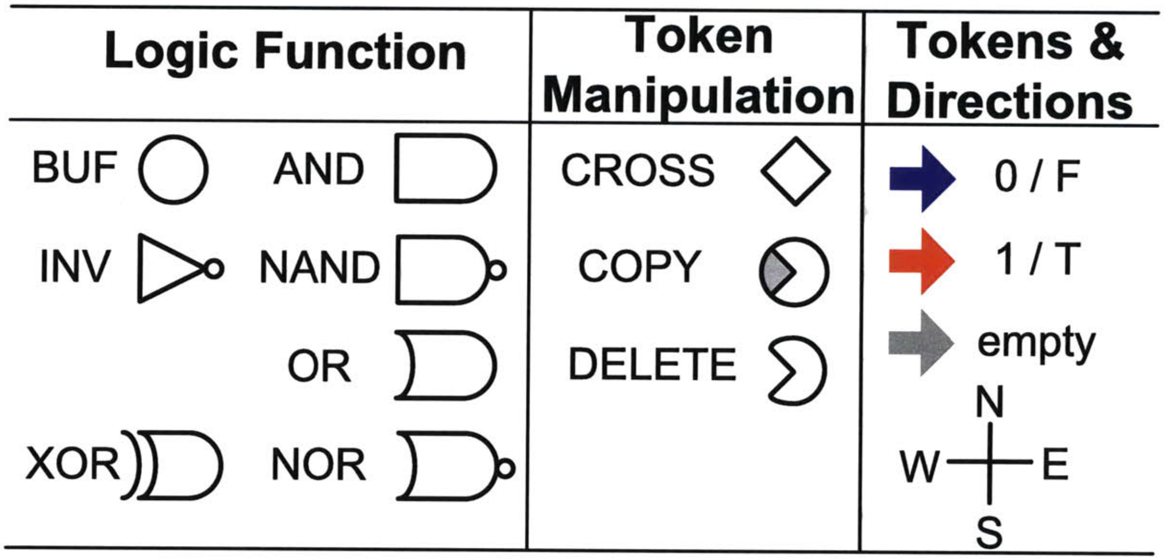

Figure 2-1: Ten Implemented ALA Gates and Token Legend

empty inputs as antibits.

The ten types of gates are divided into 6 sub-types. Because of the dual rail encoding of tokens in the CMOS implementation, inverting a token can be done by swapping how the inputs are connected. By inverting inputs or outputs, the buffer and inverter have equivalent implementations. Likewise, AND/NAND/OR/NOR are equivalent by DeMorgan's law. Except for the CROSSOVER, all gates have symmetric outputs that can be fanned out to up to two different neighbors.

2.1.1 BUFFER/INVERTER Gate Behavior

The BUFFER/INVERTER gate is a one input gate that waits until the input has a token and the output is empty and then emits it (or its complement) to the output and acknowledges the input. A chain of BUFFER gates connected to each other can hold a series of tokens. A simple way to generate a periodic sequence is to have a loop of BUFFER gates that have been initialized to your desired sequence. As the sequence travels around the loop, a copy can be fanned out. It is important to note that in addition to allowing temporary storage of tokens, the BUFFER/INVERTER gate is necessary because interconnect is only local in ALA, meaning cells can only communicate if they are adjacent to one another. There is no wire primitive—cells must be connected by a chain of BUFFER gates.

2.1.2 AND/NAND/NOR/OR Gate Behavior

The AND gate is a two input gate that waits until at least one F input or two T inputs have arrived and the output is empty. It then emits a F output if there was an input of F, and a T output if both inputs were T. After the output has been generated and both inputs have arrived, they are acknowledged. NAND, NOR, and OR can be constructed from AND by using DeMorgan's law.

2.1.3 XOR Gate Behavior

The XOR gate is a two input gate that wait waits until both inputs have arrived and emits their exclusive or, i.e. it emits an F if both inputs match and a T if they do not match. Unlike the AND gate, it does not short circuit.

2.1.4 CROSSOVER Gate Behavior

The CROSSOVER gate is essentially two BUFFER gates in the same position. It waits for an input and emits the input in the same direction it was going. So if an input arrives from the east of the CROSSOVER gate, the gate emits it to the west.

2.1.5 COPY Gate Behavior

The COPY gate is a two input gate with asymmetric inputs called control and data. Control selects whether or not to copy the data input. So when a T token arrives at the control input, the token on the data input is copied to the output, but not acknowledged and therefore remains on the data input. If more T tokens arrives on the control, the gate will continue to emit copies of the token waiting on the data input. An F token at the control input causes the data input to be passed on as it would be in a buffer and be acknowledged. If you consider the BUFFER gate to pass tokens across space, the COPY gate passes them across time. The COPY gate always emits the same number of tokens as it consumes via the control input, but consumes only as many tokens via the data input as it receives F tokens on the data input.

2.1.6 DELETE Gate Behavior

The DELETE gate is the dual of the COPY gate. Like the COPY gate, the two asymmetrical inputs are called control and data. Unlike the COPY gate, control selects whether or not to delete the data input. When a T token arrives at the control input, a token is acknowledged from the data input and no new tokens are emitted. When an F token arrives at the control input, the data is passed on with the same behavior as a COPY gate when it receives an F control token. The DELETE gate always consumes the same number of tokens on the control and data inputs, but emits only as many tokens as it receives F control inputs.

2.2 Choice of Rules

There is a minimum complexity/variety necessary to enable universal behaviors, but this is fairly low. For example, Banks demonstrated universal logic in a 3 state 4 neighbor synchronous CA [2] (which can in turn be implemented by 2 state 4 neighbor CA). However in a system like that, many cells must undergo many steps to compute even a single binary operation. One of the goals for ALA was to ensure that implementations of practical sorts of systems could be done with reasonably small overheads. To allow optimization below the granularity of the cells, we wanted cells that were as complex as would be useful. On the other hand, very complicated cells would be difficult to analyze and implement in other technologies and might have capabilities that would be wasted. The eventual choice was to use two input boolean operations as the core granularity.

Although in general, NAND is sufficient to construct universal Boolean logic functions, within ALA we also need explicit support for wiring and crossovers. To fully take advantage of the asynchronous nature of the circuits, we need to be able to pass information between domains that are operating at different rates. The COPY and DELETE gates provide this functionality and the CROSSOVER gate ensures that we can route signals in different domains across each other without worry about their relative rates as would be necessary using XORs. Thus the true minimum set of gates would contain NAND, CROSSOVER (which could be used as a buffer), COPY, and DELETE. However, for ease of design we choose to include the other logic functions as well. Since the hardware uses a dual rail encoding for the logic in which inversion can be done by swapping where inputs or outputs are connected, AND, NAND, OR, and NOR all use the same circuit. It would have been possible to include a much larger set of gates, but finite design effort and a desire for compatibility with future reconfigurable designs motivated us to keep the number of gates close to the minimum.

At least in CMOS, it is likely that actual wires could be used to connect over dozens or possibly hundreds of cells without significant degradation of performance. This would reduce power consumption substantially.

2.2.1 Ensuring Deterministic Behavior and Allowed Variations

The choice of gates for ALA can not include anything that can be affected by *when* tokens arrive. This is to avoid introducing technology dependent design constraint based on how fast each gate processes inputs. The current design ensures this by using what is close to a single-assignment single-reader model one might see used for parallel programming. New values can be written, but only after the old one has been garbage collected by acknowledge signals.

2.2.2 ALA Buffer Chain Example

This section will provide an example of the basic operation of an ALA design. Here is a diagram of four ALA cells (labeled A,B, C, and D). The arrows between cells represent state storage elements for tokens. Red arrows indicate where a True token is stored, blue arrows indicate where a False token is stored, and gray arrows indicated that no token is stored at that location. The three cells labeled A, B, and C are buffer cells that wait until their input has a token and their output is empty. The fourth cell, labeled D, is an AND cell that atomically consumes a pair of input tokens and produces a single output token that is a logical AND of the two consumed tokens.

Figure 2.2.2 shows the initial state of the cells. Figure B shows the same cells again a short time later. We see that the True token waiting at the input of cell B has been propagated to cell C.

After another small amount of time, figure 2-3 shows both cells A and C have fired propagating their tokens onwards. Note that because the cells are operating asynchronously there is no defined order for these events. (In the hardware described later in this document the token would probably propagate all the way to cell D by the time the acknowledge logic cleared the token from the input.) However, regardless of the order of these updates, the next token to arrive at cell D will be a True token. Eventually the two tokens will end in the configuration shown in figure 2-4. The True token waiting at the input to cell D will not propagate through the AND cell labeled D because it must wait until both inputs are present. Similarly, cell C cannot propagate the False token from its input because it is blocked by the token waiting in front of cell D.

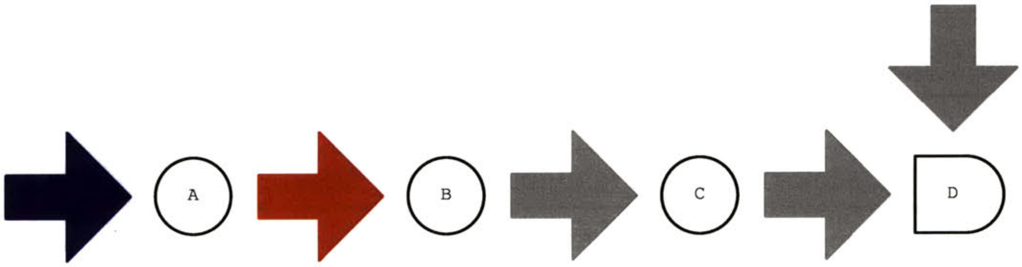

Figure 2-2: Initial Condition for Buffer Demo

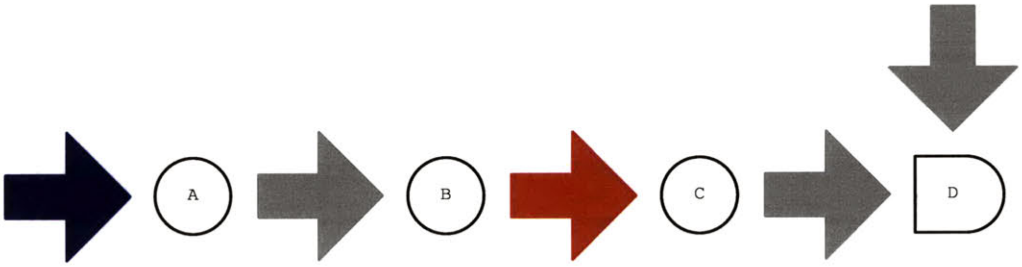

Figure 2-3: Buffer Demo After 1 “Step”

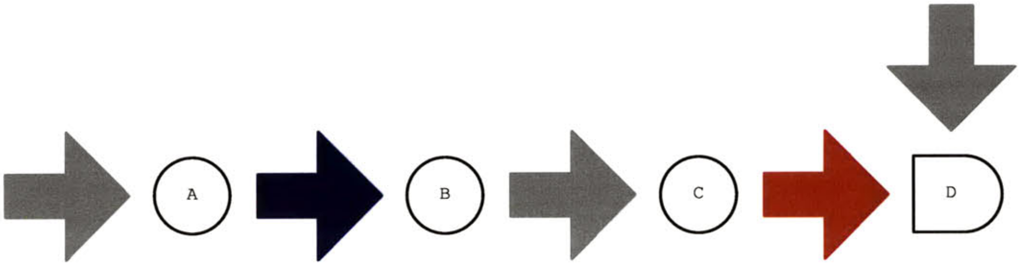

Figure 2-4: Buffer Demo After 2 “Steps”

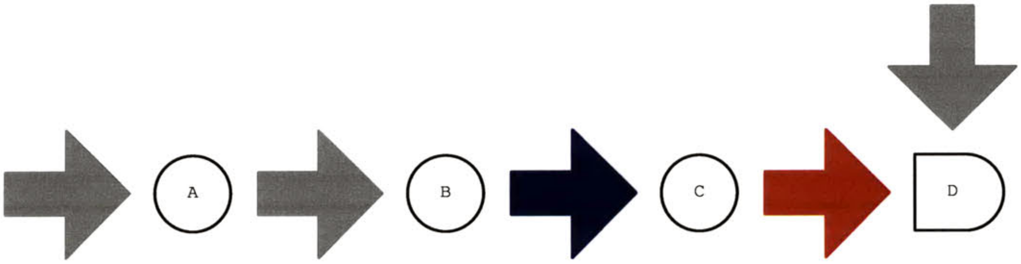

Figure 2-5: Buffer Demo After 3 “Steps”

Chapter 3

Example ALA Designs and Tools

This chapter presents a few simple examples of ALA designs and tools. The development of better design tools and more efficient designs are active areas of research and have progressed beyond what is described in this chapter, however these designs offer a good introduction to how gates can be combined and used and some minimal requirements for design tools.

3.1 Oscillators and Control Signals

Transporting data and performing computations have roughly equivalent costs in the ALA universe so for many applications it is useful to work with serial streams of data. This creates a trade off between throughput of blocks and their area and complexity. While doing serial processing, it is often necessary to have a periodic control signal (for example to mask out carry bits in a stream of fixed width additions so that the overflow doesn't end up in subsequent calculations). A simple solution for this is to create a loop of BUFFER gates initialized to the desired sequence.

This structure can be initialized to hold an arbitrary pattern which will be emitted at the output and loop back around so that it is repeated. However, many situations do not call for an arbitrary sequence but instead need a single False token in the loop or similar. For these situations, filling the entire bit loop with the pattern costs n^2 power for each copy of the output (size of ring grows with n, and each bit has to travel

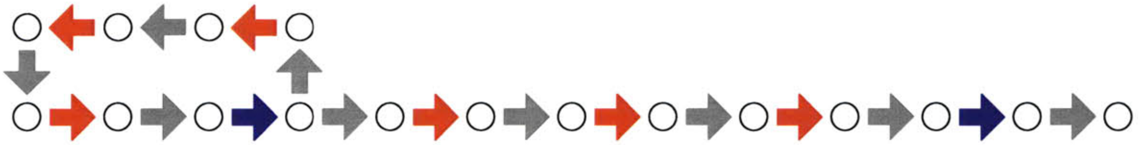

Figure 3-1: Bit Loop Emitting (0,1,1,1) Attached to Line of Buffers

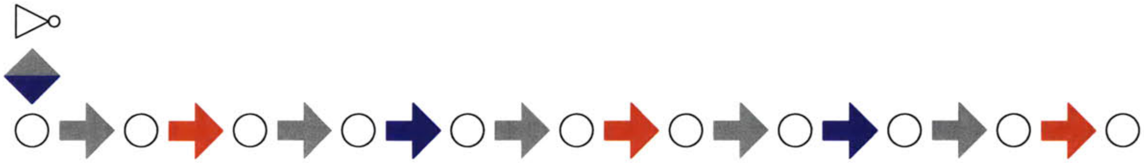

Figure 3-2: Binary Oscillator Attached to Line of Buffers

around the entire ring every cycle) and linear space. However, we can generate a signal with an arbitrary period with a constant power cost per bit and logarithmic space cost by building a hierarchical oscillator. We start with a simple 2 state oscillator as shown in figure 3.1. A short set of buffer cells has been attached to the output to illustrate the generated output.

Now there are two units that we will add to this. The first is a bit doubler that emits two copies of each input it receives. The two cells on top form another oscillator that produces a stream of alternating True/False tokens. These feed into the control of the copy at the bottom which will duplicate the data input on the True tokens and pass it through on the False input which has a net result of creating two copies of each input token. One of these with a binary oscillator connected to the input will produce two True tokens followed by two False token, as show in figure 3.1, for a period of 4 tokens.

By chaining together these bit doublers we can generate any period that is a power of two. Because each doubler only fires half as often as the one before it, the total power is a geometric sequence summing to a constant. Figure 3.1 shows a heat map of an oscillator indicating the relative amounts of power consumed in each cell (red is most, blue is least).

While power of two length sequences are sufficient for many situations, we may

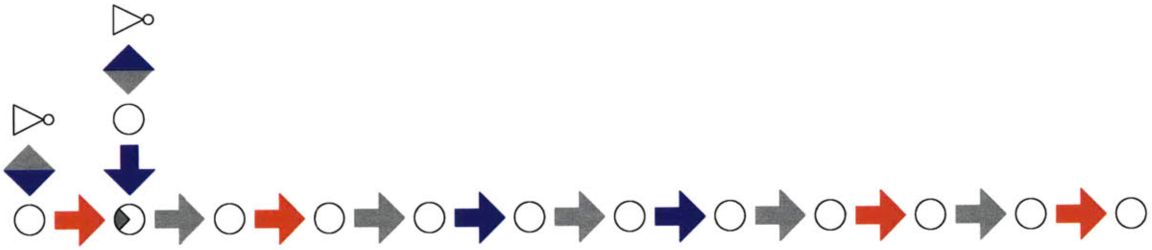

Figure 3-3: 4 State Oscillator and Line of Buffers

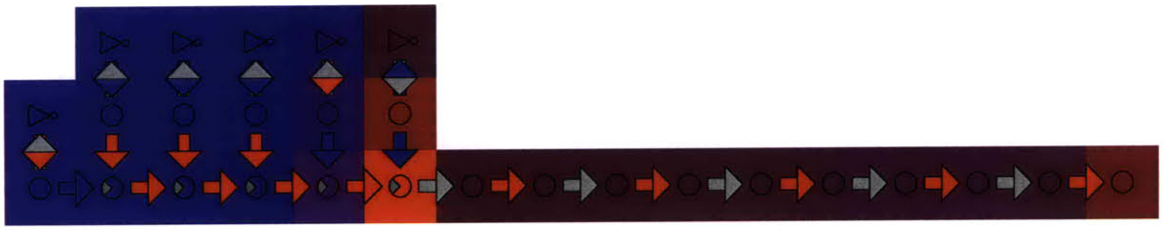

Figure 3-4: Heat Map of 64 State Oscillator

also want to be able to generate sequences with arbitrary periods. A circuit that copies the next bit after a False-True sequence will effectively add one to the period. Combining this with the doubler makes it possible to generate sequences of arbitrary periods. Appendix B shows a parametric design that generates oscillators with arbitrary periods.

3.2 Adder Design

An important aspect of the ALA architecture is that it allows trade-offs between energy, throughput, and area even for fairly basic primitives. The following are

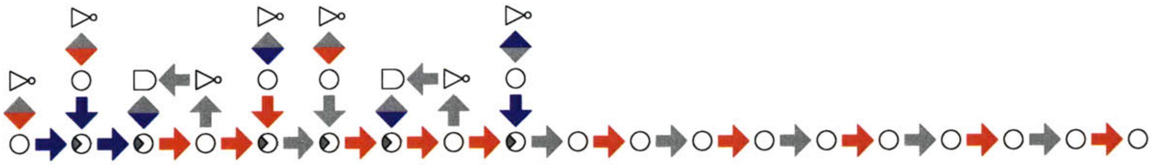

Figure 3-5: 42 State Oscillator

Cells used	Throughput (Steps)	Throughput	Energy/Bit (pJ)
8	1/2	773 MHz	1.51
11	2/3	781 MHz	2.68
13	1	1.24 GHz	3.00

Table 3.1: Relative Adder Performances

three different adder designs which range from smallest size and energy per token through two larger and higher power designs with better throughput. The smallest design, shown in figure 3.2 was created by hard-earned insight about how to order the intermediate steps of the computation to preserve intermediate information, as well as lots of trial and error focused only on size. It has the lowest throughput. The intermediate design in figure 3-6 lays out the expected operations along a diagonal to try and ensure uniform buffering depth. The largest and fastest design from [12] shown in figure 3-7 used graph analysis techniques to add the exact buffering needed to compensate for the feedback of the carry bit and achieve perfectly equal buffering with maximum performance.

The following chart shows their relative performance in Specter simulations. Here step throughput refers to the steady state ratio of output tokens to updates if all cells are updated simultaneously in a synchronous fashion as discussed in [12]. A graphical representation of the step throughput can be seen in the spacing of tokens on the output buffers in figures 3.2, 3-6 and 3-7, where the maximum throughput would be 1 token every other cell.

The difference between the step throughput and the predicted throughput from circuit simulations likely stems from optimizations in the circuit design that allow tokens to propagate through ready cells faster than the full handshake time. By optimizing our design to allow this, we reduced the cost of buffering mismatch and decreased latency substantially. However in comparison to each other, the increased area and power usage still provide a significant speed improvement. Although counterexamples could likely be constructed, this indicates that even very simple performance metrics could be useful to compare designs across technologies.

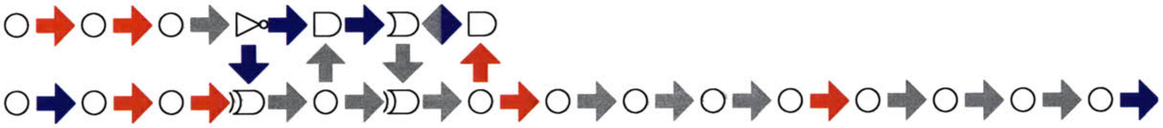

Figure 3-6: 8 Cell Adder

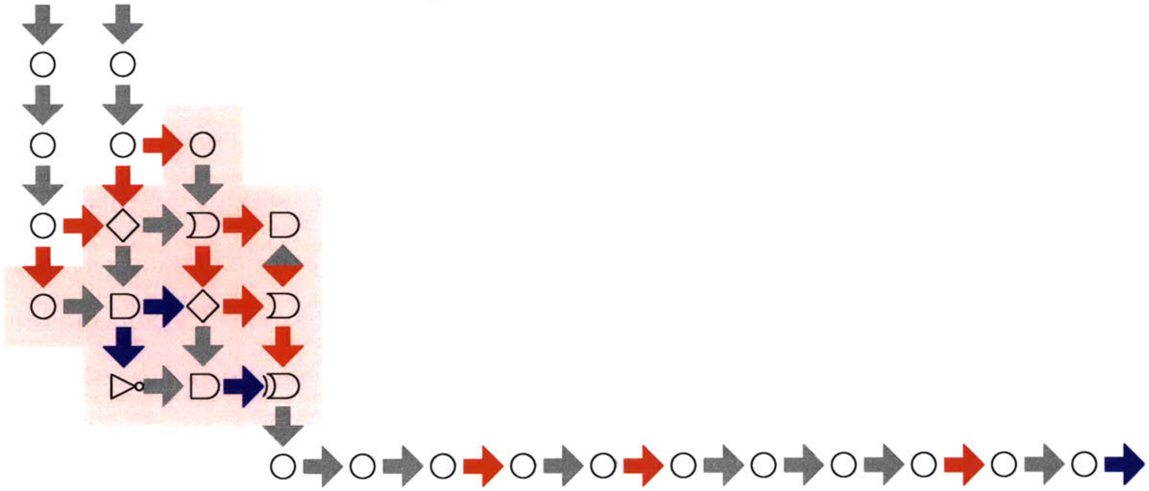

Figure 3-7: 11 cell adder

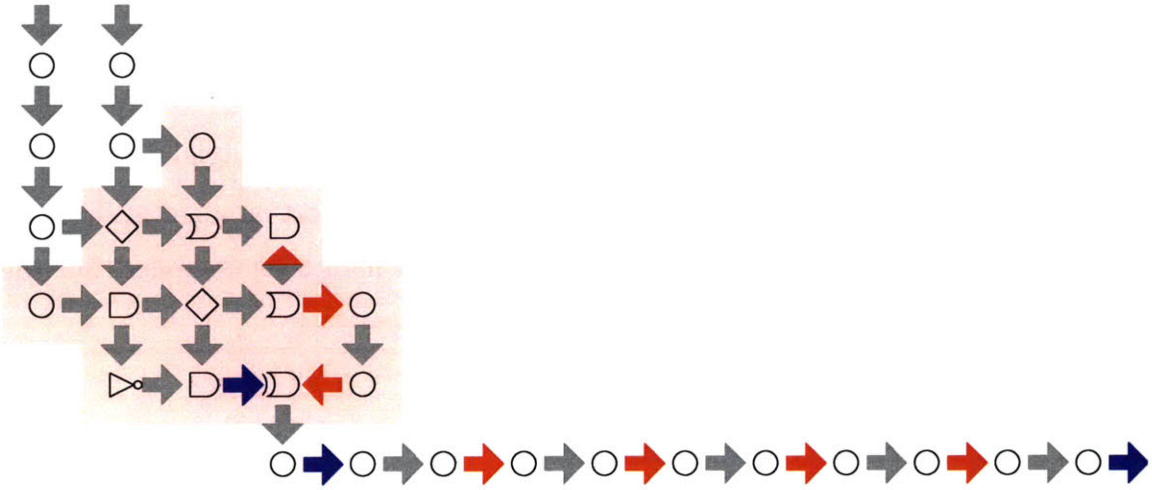

Figure 3-8: 13 Cell Adder

3.3 Design Tool

While pencil and paper or your favorite vector graphics program are reasonably useful for prototyping designs, a more structured approach is necessary to get designs into a format suitable for simulation or automatic processing. Many designs can also be expressed with repeating structures or can be parametrized in terms of things like word length or important constants. While more specific tools are being developed, the designs in this document were generated by Python code that makes calls to the simulator to define the initial cell positions. The choice of language was nearly arbitrary, but the use of a programming language allows substantial flexibility for ways to describe hierarchical and parametric designs. Although working in this fashion requires substantial visualization skills (and often a good bit of debugging), it makes it very easy to add new design capabilities simply by writing more Python modules. For example, I implemented a simple search that builds a chain of buffers along the shortest unoccupied path between two cells. In their current state using these tools is analogous to using an assembler. It is possible to specify a design if you know exactly what you want, but there are no safety nets. One could easily imagine adding a simple type system to help ensure that modules were being connected correctly or automatically doing buffering checks that identify sections where mismatched data path lengths could harm performance to try and come closer to a high level language. Tackling this issue fully would be a substantial research project on its own. The tools used here are more analogous to an assembler than a high level compiler, but they still allow moderately complex designs with the ALA gate set.

3.3.1 Hierarchical Designs

One very important feature of high level languages is the ability to abstract away complex functions and later refer to them as primitive operations. One of the first extensions I did was to create a part class that allows cells or other parts to be added using a local coordinate system. A part could than be instantiated at any position, rotation, or reflection. This serves the same role as function calls in a high level language by encapsulating complex behavior in a way that is easy to reference multiple times. It also turned out to improve readability of the generated designs substantially. Simply coloring the backgrounds for different parts made it much easier to see how the modules were connected to each other. Interestingly enough, part shapes tend to be sufficiently distinct that it was possible to recognize them without any additional markings. Admittedly, knowing what the parts do requires either careful examination or prior knowledge.

3.3.2 Parametric Designs

A feature that will be useful to preserve in future design tools is the ability to do parametric designs. The spatial structure of ALA naturally lends itself to graphical design tools, but there is a class of designs that are not readily possible with a direct graphical description. The appendix shows an example of this in a parametric design that creates oscillators with arbitrary periods. The simple addition of repeating tiles covers a broad class of designs, although there are examples where more complex behaviors can be critical. The oscillator is a simple example, but more complex ones could include the Fast Fourier Transform and many examples from cryptography. The ability to optimize away unneeded data paths can represent substantial savings as each "function call" requires space to be allocated for the module that computes that function. On the other hand, design with graphical primitives has the advantage of being much more intuitive. The balance between graphical intuition and descriptive flexibility remains an interesting question for future research.

Chapter 4

ALA Hardware Design

The asynchronous aspect of ALA could be implemented in traditionally clocked logic with token state simply remaining unchanged when cells are not ready to update. Unfortunately this consumes dynamic power even when cells are idle and requires a globally distributed clock. In addition, having cells operate in lock step greatly increases the sensitivity of performance to careful design, as shown with the adders. A clocked design would likely be worth exploring to see to what extent the simpler logic enables faster operation or lower dynamic power. The current designs uses a Quasi Delay Insensitive (QDI) style. In many ways QDI is the "purest" kind of computationally universal asynchronous logic in that it makes the fewest assumptions about the underlying technology. While this made design easier and increases portability and reliability, subsequent experiments predict that using a Self-Timed logic style could result in substantial area savings and an additional 30% improvement in power and throughput [3]. Other designs in progress, include one with carefully selected representations for the internal signals to reduce power and area that avoids ratioed logic to allow lower operating voltages for improved efficiency at lower speeds, as well as a design based on SRAM cells that could substantially reduce transistor count. At this time, these designs are incomplete and remain as subjects for future work.

4.1 Asynchronous Handshaking

Synchronous circuits are carefully designed to ensure that all signals will be ready before the next clock edge. In asynchronous circuits, a local ready signal is computed that replaces the clock. This can be done with a carefully engineered delay element or a data representation, like the one used here, where intermediate and final values can be recognized. In our design we used a dual rail data encoding where two wires, one for 0/F and one for 1/T, represented the state of each token. When both of these lines were low the port is said to be empty or X. If the T or F line was high that was the state of the port. Both lines high was an invalid state. We used a four phase handshake where a third acknowledge line would go high to signal ready for a new token and go low to indicate after the new token had been latched.

4.1.1 State Storage with C-Elements

Each ALA cell needs to hold a small amount of state which would normally be held by a register or latch in regular sequential logic. However, in asynchronous logic there is no global clock to coordinate operation among registers as normal. To compensate for this a common component in asynchronous logic is a C-Element. Instead of using a single input to trigger when data begins being passed to the output it waits for a combination of inputs to be set or cleared and then transitions or otherwise holds the current state. Careful design of the logic ensures that the computation remains within the stable inputs until it is complete, which then triggers the output to change and cascades into the next phase of the computation. There are numerous ways to implement C-Element (for some examples see [16]). Here is an inefficient but easy to understand C-Element constructed from an AND gate, NOR gate (AND with inverted inputs), and a Set Reset Flipflop. When both inputs or set or cleared the output will likewise be set or cleared.

The C-Element is a powerful component for the design of asynchronous circuits. For example, the following circuit is an asynchronous 4 state oscillator. It is left as an exercise for the reader to understand the exact operation. It is very similar to the

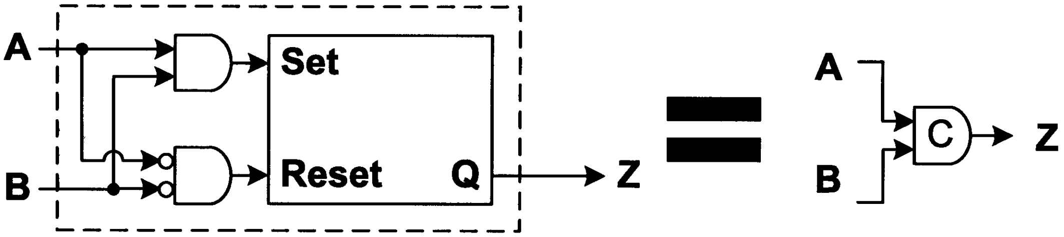

Figure 4-1: Basic C-Element built from AND/NOR/SR-Flipflop

handshake mechanism in a precharge full buffer described in the next section if you want a hint.

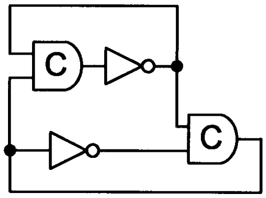

Figure 4-2: C-Element Oscillator

4.1.2 Precharge Full Buffer

The core of each cell centers around a precharge full buffer similar to [21]. This circuit implements a 4 phase handshake. In the idle state C1 and C2 are reset and C3 and C4 are set. In this situation ackA (C3) will be high to signal that the buffer is waiting for a new input. The handshake works as follows:

1. New input arrives at either A0 or A1 triggering C1/2 to emit a new Z value if ackZ is high
2. The new Z value causes ackA to go low by way of Zempty resetting C4 in the process
3. Low ackA from this buffer causes the preceding buffer to clear this buffers input

4. Aempty going high sets ackA (signaling ready for new token) and returns the cell to idle

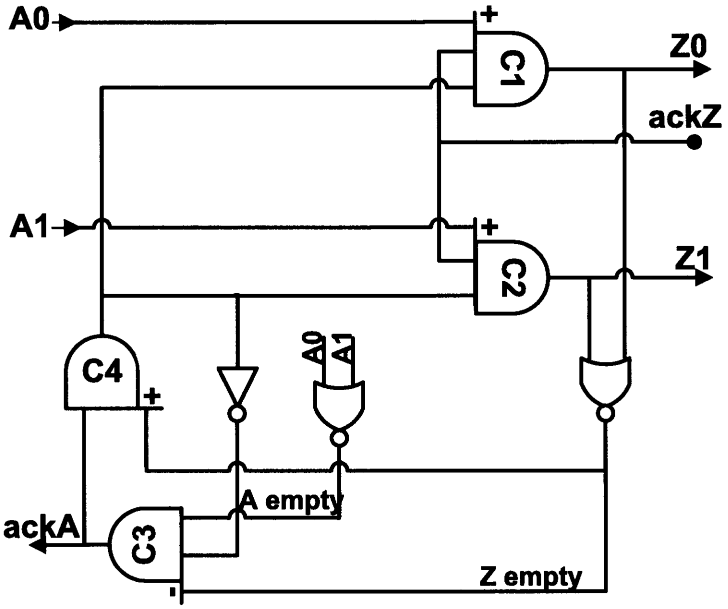

Figure 4-3: PCFB Schematic

The full state diagram for the PCFB is shown in figure 5.1. For more details of operation, information on incorporating logic, and the modifications to implement COPY and DELETE cells see [4].

4.2 Gate Layouts

Figures 4.2 to 4-8 show the full layouts for the 6 gates classes. The BUF Gate can be converted to a INV gate by swapping the T/F token wires of either the input or the output. Similarly the NAND gate can be converted to AND, OR, or NOR. Fanout requires the addition of a C-Element which is not shown.

4.3 Performance

After finishing the layouts we used the Spectre simulation tool to estimate their performance. Table 4.3 shows our results. Throughput is the maximum rate at which a gate can process available tokens. Latency is the time between when the inputs arrive and when the output is emitted. Because outputs can be emitted immediately before the full handshake is complete, tokens can propagate through empty cells much faster than the cycle time that limits throughput. This significantly decreases the performance impact of buffering mismatches. Energy/Op is the amount of energy consumed when a cell fires. Transistors measures the total number of transistors needed for each kind of cell. Transistor Area is the total area of the transistor gates (the control terminal of the MOSFET, not the entire cell). Cells are placed on a lattice that is 28.2 μm by 4.295 μm to accommodate the largest cell (the crossover) and thus occupy 122 μm^2 each.

Adding fannout to a gate is done with a single C-element and increases the Energy/Op by about 0.039 pJ and decreases the Throughput by 10 to 15%.

Gate	Throughput	Latency	Energy/Op	Transistors	Transistor Area
BUF/INV	1.71 GHz	0.0893 ns	0.144 pJ	52	2.724 μm^2
CROSS[1]	1.71 GHz	0.0893 ns	0.144 pJ	104	5.448 μm^2
AND[2]	1.41 GHz	0.105 ns	0.163 pJ	60	3.084 μm^2
XOR	1.27 GHz	0.115 ns	0.168 pJ	64	3.520 μm^2
COPY	1.37 GHz	0.0909 ns	0.231 pJ	75	3.984 μm^2
DELETE	1.50 GHz	0.105 ns	0.170 pJ	63	3.408 μm^2

Table 4.1: Cell Performance Numbers

4.4 Design Flow: ALA to Silicon

We have developed a library of ALA cells in a generic 90nm process that can be arbitrarily tiled. The first layer of tiles are function tiles that select an operation (i.e.

[1] Energy/Op for the CROSS cell is for 1 token in either direction.

[2] AND, NAND, NOR, and OR as well as BUF and INV have equivalent hardware; logical inversion on inputs and outputs are trivial due to dual rail encoding

NAND, COPY, CROSS) as well as choosing between one or two outputs. Directly on top of this are tiles that connect inputs and outputs with their counterparts in adjacent tiles. Figure 4.4 shows the tiles needed to create a NAND cell with inputs from the east and north. While standardizing the locations of the input and output ports and adding a separate layer of wiring was largely motivated by wanting to avoid doing new layouts for every combination of inputs, it illustrates the ease with which ALA could be used in a sea of gates style fabrication process. All ten of the implemented function tiles could be encapsulated with just 4 tiles and a wiring layer that can rearrange inputs. AND, NAND, OR, and NOR are equivalent and would work as a buffer/inverter if the inputs were connected in the same direction, XOR has different logic and could serve as the second buffer in a crossover by connecting one of the 0 wires in the input to the ack pin. COPY, and DELETE would be the remaining two tiles. With fairly minor optimizations it would likely be possible to get this area overhead within a factor of two of the current tiles sizes. Using these superfunction tiles, it would be possible to fabricate custom ALA ASICS with only one or two custom masks which would drastically reduce cost (in both money and time) of fabrication.

Currently, the process of selecting and placing tiles is manual, but because of the direct one to one mapping to the hardware, the only obstacle to automating this process is being able to manipulate the appropriate file formats. As it stands, I was able to design, layout, verify, and simulate a small LFSR in less than two and a half hours. The majority of that time was spent on tracking down a bug where I had used two input tiles that connected to the same side of the function tile and on setting up simulations. Figure 4.4 shows the ALA design for a three bit LFSR and its hardware layout.

4.4.1 Comparison with a traditional design: 3 Bit LFSR

For comparison with existing techniques, I implemented a 3 bit LFSR in the same process technology using synchronous logic. Table 4.4.1 shows the relative performance of the ALA and custom versions before layout. This comparison ignores the cost of external clocking (which could as much as double total power consumption) and potential benefits of the asynchronous circuits. To actually deploy this design the clock speed would likely need to be significantly lower to account for manufacturing imperfections. After accounting for the clock overhead and some of the possible improvements from chapter 6, the overhead would be favorably comparable to the 10x overhead one might expect when using an FPGA instead of custom hardware.

Design	Spectre Throughput	Spectre Energy	Transistor Count	Gate Area
ALA	1.23 GHz	0.717 pJ	340	1482
Custom	5.68 GHz	0.0196 pJ	78	101
Overhead	4.61	36.6	4.36	14.67

Table 4.2: Comparison of ALA Design and Custom Design Before Layout

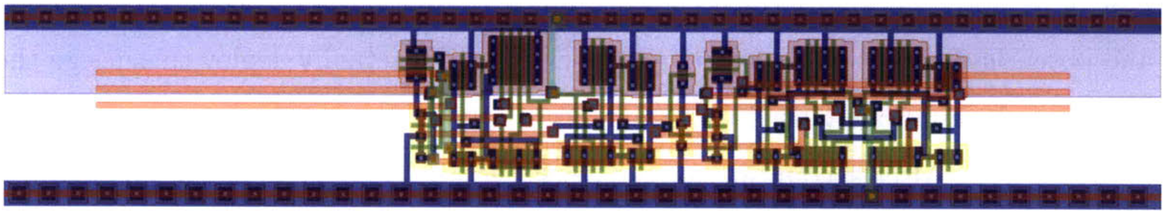

Figure 4-4: BUF Gate

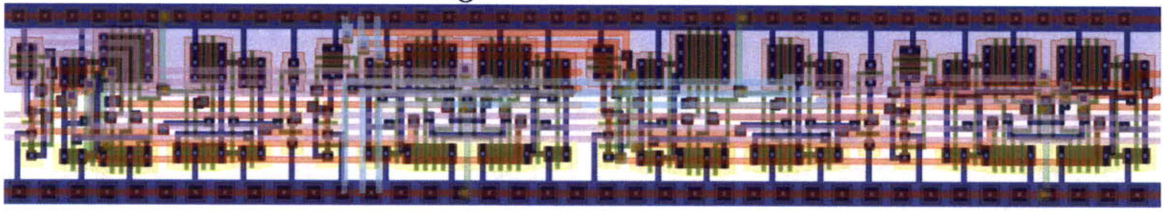

Figure 4-5: CROSS Gate

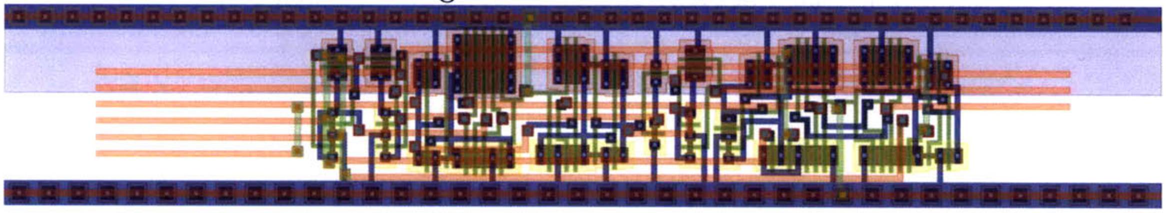

Figure 4-6: NAND Gate

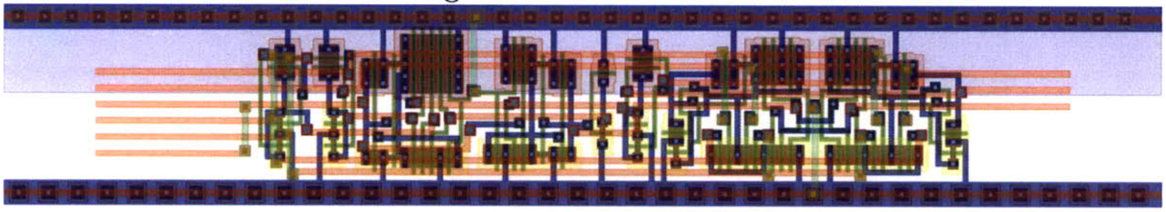

Figure 4-7: XOR Gate

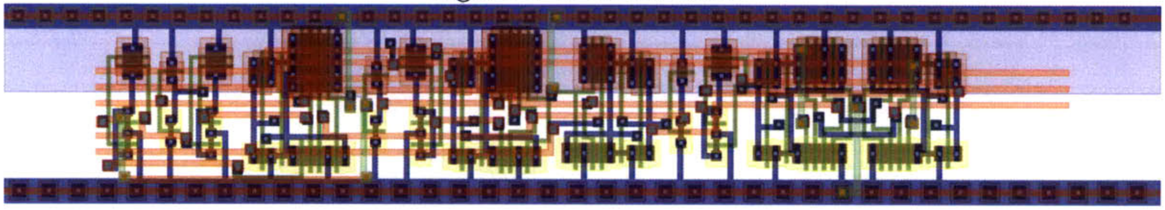

Figure 4-8: COPY Gate

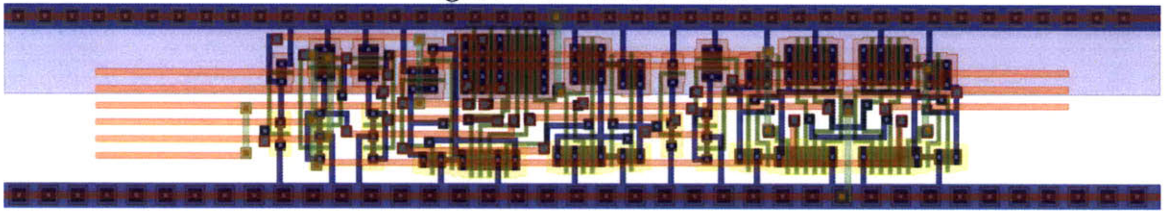

Figure 4-9: DELETE Gate

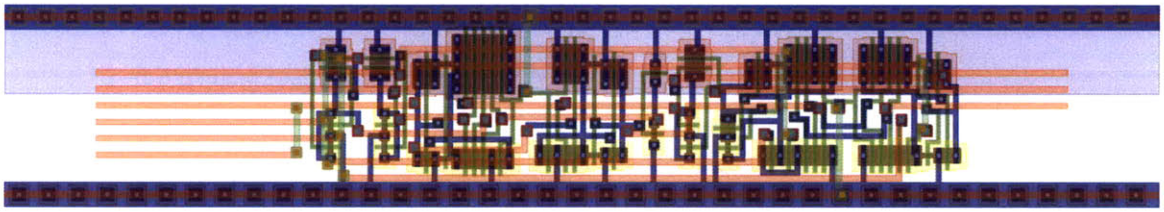

Figure 4-10: Main Tile For NAND Cell

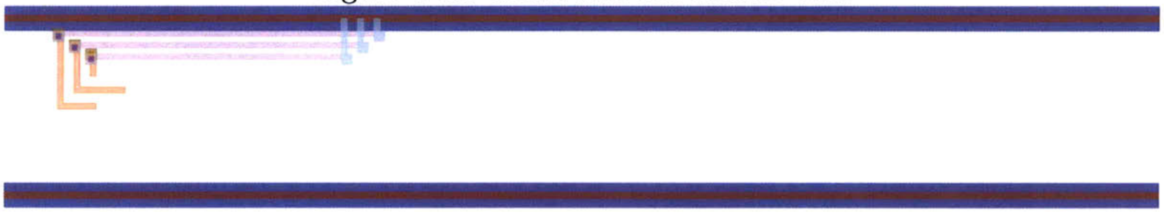

Figure 4-11: Overlay Tile With Wires For North Input

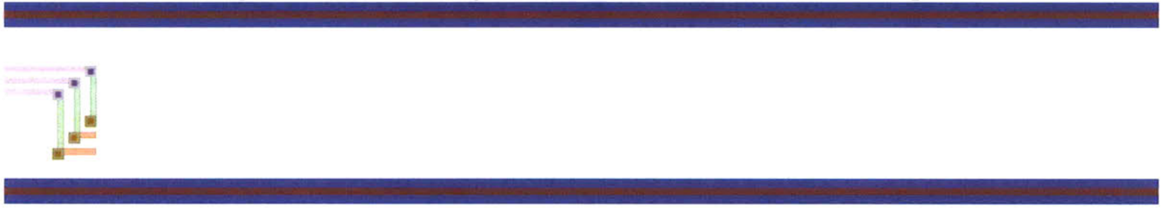

Figure 4-12: Overlay Tile With Wires For West Input 1

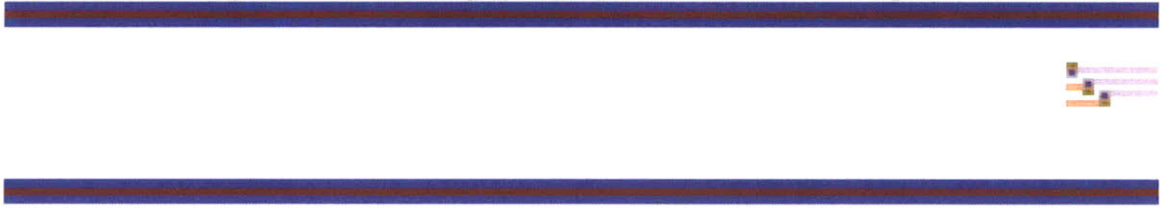

Figure 4-13: Overlay Tile With Wires For East Output

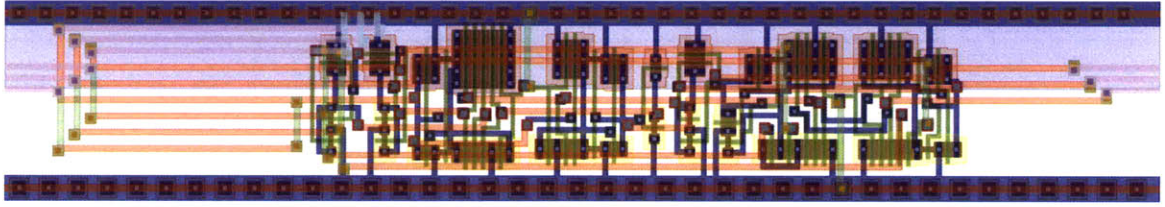

Figure 4-14: Combination of All Tiles

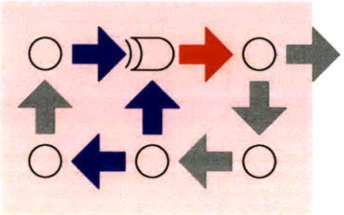

Figure 4-15: LFSR as ALA

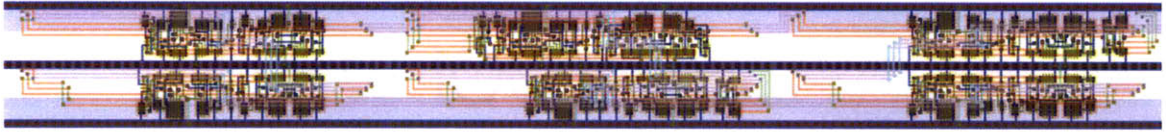

Figure 4-16: LFSR as Layout

Chapter 5

ALA Simulation

Although we are planning to do so in the near future, none of the ALA hardware designs have been fabricated. Instead, we have used performance estimates based on Spectre simulations of our design done in a generic 90nm process. From these simulations I extracted timing and power consumption information which I integrated with a custom simulator for ALA. This chapter describes a discrete event model of the ALA hardware and the simplified model used in the custom ALA simulator TkALA.

5.1 Discrete Event Model of PCFB

One of the powerful aspects of ALA is that there is only a small set of unique cells with very controlled discrete interactions. Figure 5.1 shows the full set of events and interactions for a buffer cell and its two neighbors. Each of the ovals represents a transition on one of the C-elements inside the cell. Boxes represent transitions in neighboring cells. Solid black arrows represent internal dependencies. Colored arrows represent external dependencies of internal transitions. Dashed arrows represent dependencies of external transitions. Each of these arrows has some delay value associated with it.

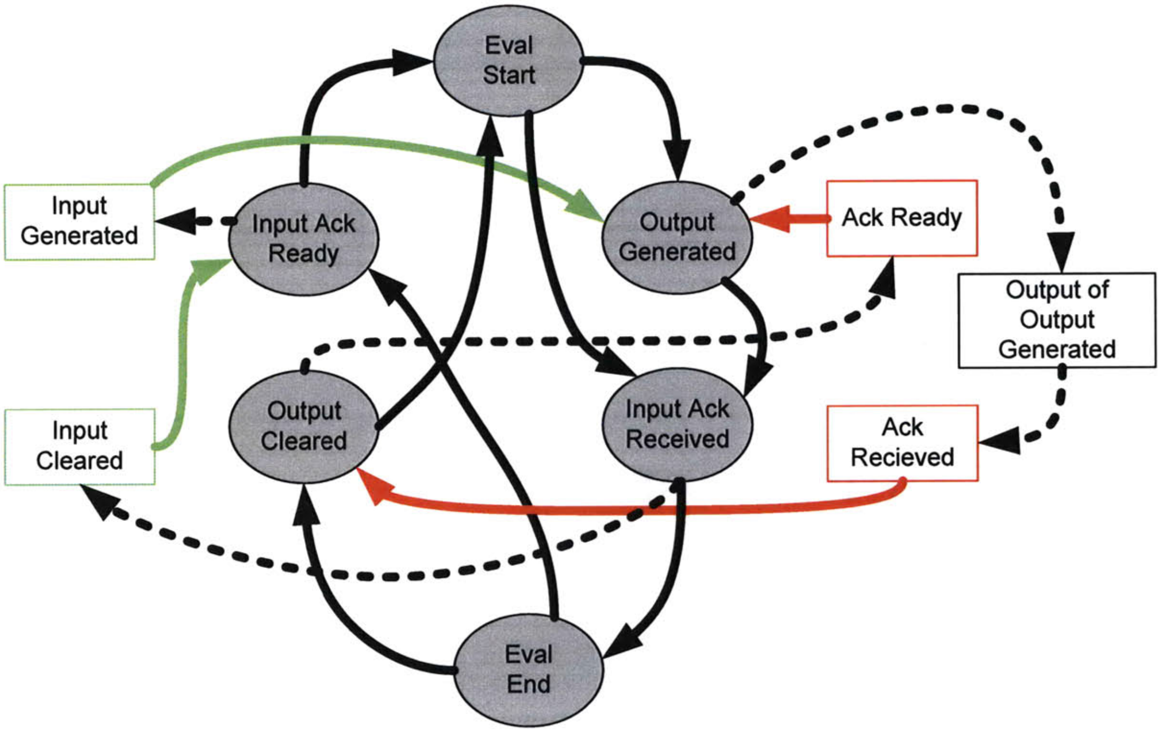

Figure 5-1: Full PCFB State Diagram

5.2 Simplified Model of PCFB

The full state diagram contains many events that are not externally visible and some delays which can be redundant when the only information that we are likely to care about is when new output tokens are generated. I used a simpler model that attempts to capture the critical delays without needing all of the intermediate events. The simplified state diagram for a buffer cell is shown in figure 5.2. This model was created primarily through inspection and for compatibility with the simulator architecture. It could likely be improved with a more rigorous analysis.

5.3 TkALA Simulator

Well before the hardware was designed, we needed a simulator to see how hard different gate sets were to work with. To fill this need I wrote TkALA. After the hardware was designed I added timing and power information extracted from Spectre simula-

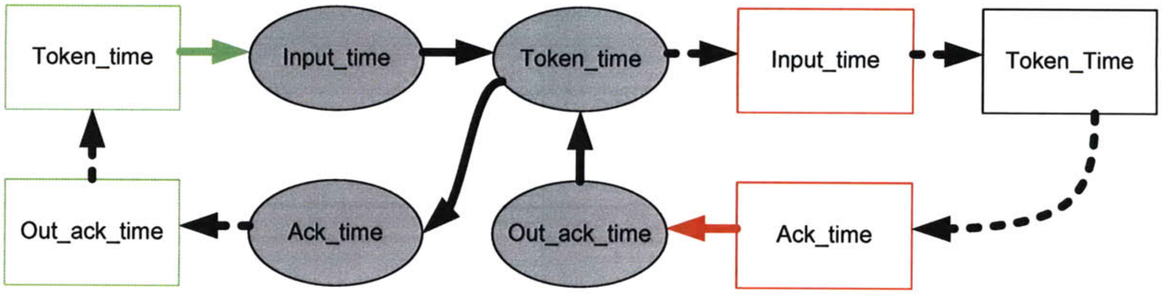

Figure 5-2: Simplified PCFB State Diagram

tions of our cell designs and the simplified state model to enable performance analysis directly from the TkALA simulator. Despite being an interactive tool written entirely in Python, TkALA is substantially faster than Spectre thanks to the simplicity of the event model. Although the exact timing was difficult to measure, a design in Spectre that would take 10-15 minutes to simulate took only few seconds in TkALA (which for designs of this size was limited because cell updates were triggered by keyboard events). Unfortunately, the simulator was primarily intended for interactive use and experimenting with different gate rules. As a result, the timing information was an incremental hack on top of the existing framework and doesn't fully match the states in the hardware. Work is currently in progress to write a new simulator in C++ that uses the complete state model and doesn't have a graphical interface. I expect the new simulator will be many orders of magnitude faster and will further exceed Spectre in speed while improving on TkALA in accuracy. Currently, there are some flaws in how short-circuiting is handled. Because of these, it is likely possible to construct circuits with arbitrarily large errors, but this is an issue with the current implementation rather than a fundamental flaw in the approach. Future simulators will not have these issues. The designs shown here were chosen because they didn't operate in a way that exposed this problem. Table 5.1 shows the results of simulating several different designs in both tools.

That simply modeling cell updates as discrete events produces fairly accurate results indicates substantial possibilities for rapidly analyzing and improving large designs using the techniques described in [12] combined with more accurate event

ALA Design	Throughput (MHz)			Energy per Token (pJ)		
	Spectre	TkALA	Error	Spectre	TkALA	Error
8 Cell Adder	772	758	1.8%	1.68	1.40	-16%
Multiplier	752	730	2.6%	12.9	14.7	14%

Table 5.1: Comparison of Simulation results between Spectre and TkALA

models. While modeling circuits as discrete events is not a new idea, the strict correspondence between the ALA design and the hardware means that issues that normally arise as netlists are converted to layouts can be avoided.

Chapter 6

Alternative Hardware Implementations

Just as the rules used are a subset of a broader class of possible rules, the design presented in chapter 4 is a single possible implementation. As the first implementation of ALA it doesn't incorporate many of the lessons learned while doing the design. This chapter attempts to capture some of the lessons that were learned too late to be incorporated. While an exciting prospect for future work would be to leverage the design flexibility of ALA to enable rapid development of designs in maturing technologies, there is much room for improvement in the CMOS implementations. The example design makes minimal assumptions about timing of the gates and is very robust to process variation. While this made it easier to design, it ignores the potential to heavily optimize the small number of cells and reap the benefits as they are duplicated many times. The self-timed implementation presented in [3] is predicted to have over 30% better speed and energy consumption in exchange for more rigorous design constraints. This chapter describes two other possible designs that might offer similar improvements as well as some additional considerations on CMOS scaling and possible new gate types.

6.1 Quad-Rail Asynchronous SRAM Half-Buffer

In order to be able to buffer a full token per cell, the current design needs to store additional state to indicate if the current token has been latched or not; this differentiates between a token that is in the process of being copied from one cell to the next and two tokens that are in adjacent cells. An alternative is to require at least one empty cell between tokens. Tokens can then remain separated without using any additional state storage (or arguably by reusing the token storage). This changes the high level semantics and could make many designs no longer function. For example a two cell loop with one token would jam because the token couldn't propagate forward without filling in the empty cell—essentially it would be at risk of eating its own tail. A good cell type to consider in that case would be a full-buffer cell that was two half-buffer cells in series.

Although this is strictly less powerful than the suggested version, it has a very simple implementation. By storing state in SRAM cells, using ratioed logic, passing signals and their complements to avoid inversions, using half-buffer cells, and being careful about signal representation to use stronger NMOS transistors where possible, we can significantly decrease hardware size. Figure 6.1 shows a design for a half-buffer that uses only 20 transistors. This design is incomplete because it doesn't include transistor sizes and may have timing glitches that prevent correct operation.

Assuming that proper sizing or using signals from the other side of the SRAM cell (and adding an inverter) would be sufficient to avoid any issues, the operation is fairly straightforward. An empty cell has BF and BT low. A cell has a token when either BF or BT are high. When a token is present in the preceding cell (as determined from the outputs AT and AF) and no token is present in the next cell (as determined from $\overline{CF}$ and $\overline{CT}$) then one of the two SRAM cells will be pulled low on the left side setting either BF or BT high. Once the preceding cell has cleared its value (as seen from $\overline{AF}$ and $\overline{AT}$) and the next cell has latched the token (as seen from CF and CT) the branches on the right will pull down BF and BT resetting the token. For use in a full circuit two additional NMOS transistor will be needed to

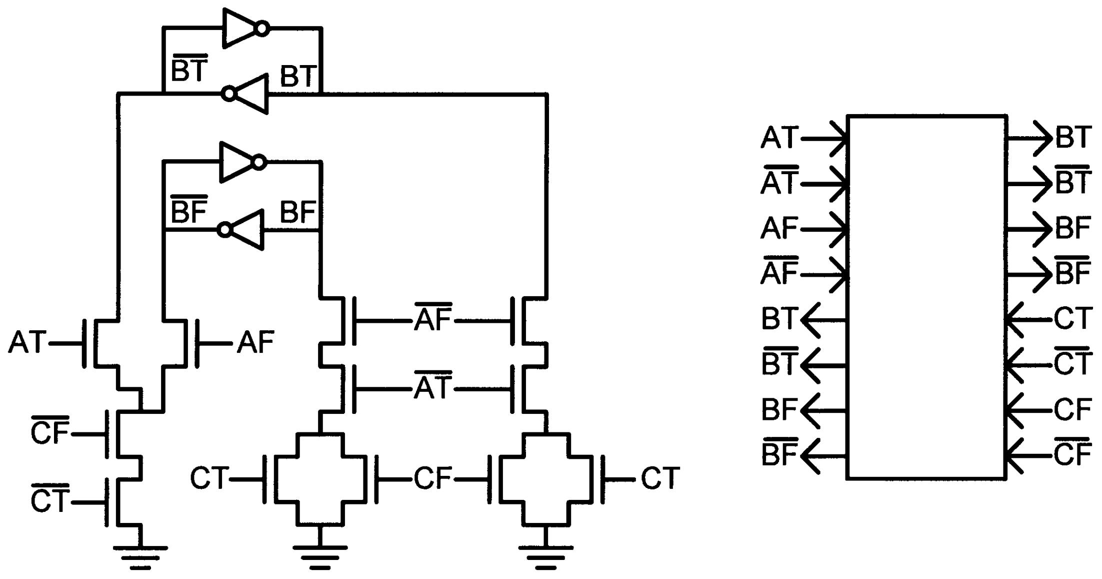

Figure 6-1: Quad-Rail Asynchronous SRAM Half-Buffer and Connections

reset the circuit. The design relies on the fact that AT and AF will never be high at the same time so the set branch on the left can be shared.

To create an AND gate in this style, simply replace the transistor driven by AT with two in series controlled by XT and YT and the transistor driven by AF with ones driven by XF and YF in parallel. Similarly for fanout, modify the stack to insure that set and reset are only done when all of the output cells are full or empty.

There is no reason why this design style couldn't be used to implement a full-buffer, but it will require additional design work.

6.2 Using Non-Ratioed Logic for Low Voltage Low Energy Operation

Much of the energy dissipated in CMOS logic comes from charging and discharging capacitive loads formed by the transistor gates. Because the energy stored in a capacitor is $\frac{1}{2}CV^2$, reducing the operating voltage of a circuit is an easy way to reduce power consumption. In synchronous circuits reducing the voltage must be done carefully to ensure that the timing constraints are still satisfied and may require reducing the clock speed. Asynchronous circuits can be designed so that they automatically slow down with voltage to maintain correct operation (as the designs in chapter 4 do). We collected data shown in figure 6.2 to demonstrate this for an early design. For situations where speed is not critical or where energy is limited, operating at a lower supply voltage can be very helpful.

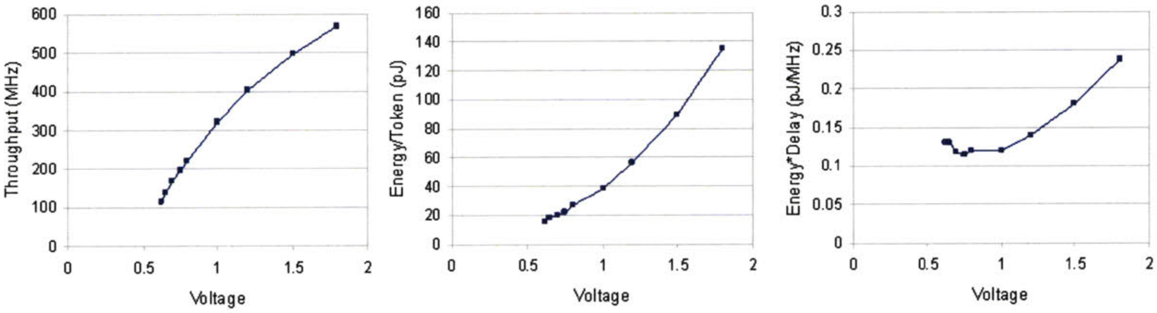

Figure 6-2: Voltage Scaling for 4-bit 3-tap FIR Filter

Sadly, the ratioed logic used in the C-elements can not switch below about $0.6V$. To improve on this, I experimented with an alternate design that used C-elements constructed from SR latches controlled by And and Nor gates. The design shown here ran in simulation at as low as $0.14V$ with an order of magnitude decrease in energy per token (though also a three order of magnitude decrease in speed). After layout and with real world noise and process variation it might not be feasible to reduce the operating voltage to such an extreme, but there remains substantial room for energy improvement. If the processing demand was variable and multiple supply voltages were available, it would be possible to trade efficiency for speed boosts on the fly while the circuit was running. Although this particular design has more transistors than the design in chapter 4 (86 instead of 52), the total transistor width is less (116.5 times minimum instead of 227) resulting in lower power consumption even at the same voltage.

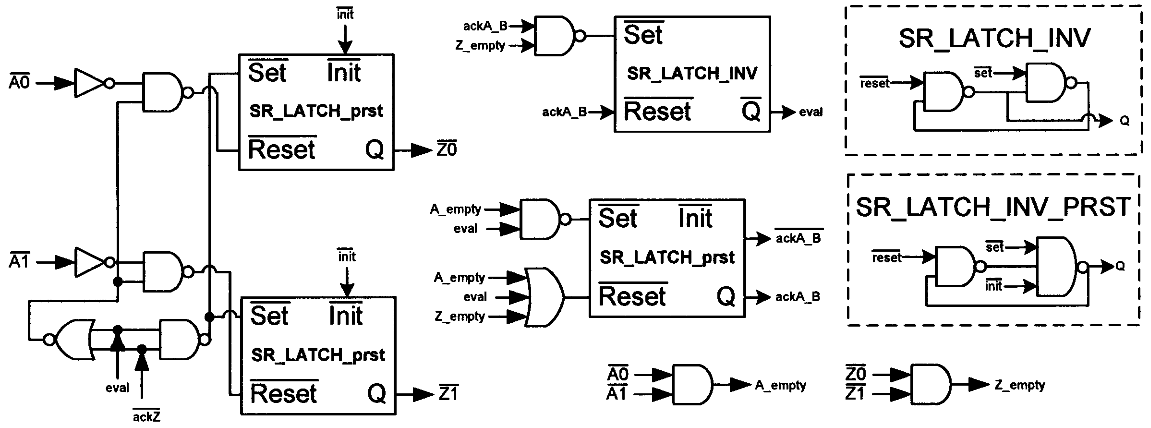

Figure 6-3: Non-Ratiod PCFB Design

6.3 Lack of Asynchronous Merge

For all of the cells, the order in which tokens arrive doesn't affect how cells are consumed or the value of the next cell that will be produced. This ensures that regardless of variations in cell speeds across fabricated chips or technologies, the resulting values will be consistent. This is an important property for ensuring technology independence and it is enforced at the individual cell level making it impossible to design a system that doesn't work correctly. However, one of the important goals of using asynchronous cells is to ensure that power is only consumed where operations are taking place and this goal is not fully met with the basic gate set. The issue stems from what we call timing domains. Any connected group of logic cells (AND, NAND, XOR, BUF, etc.) will fire exactly once (in the steady state) for each token that passes through the group. Essentially, there is a conservation of tokens similar to conservation of current in an electrical circuit. The total number of tokens may vary (because of fanout and fanin), but the number of firings will be constant. Using the copy and delete gates breaks cells into different timing domains (copy gates have control and output in the same domain and delete gates have control and input in the same domain). Breaking cells into different domains allows some sections to be run slower which can consume less power. This is perhaps most notable in something like the oscillator design from chapter 3 which asymptotically consumes a constant amount of dynamic power (though most implementations are likely to have some leakage).

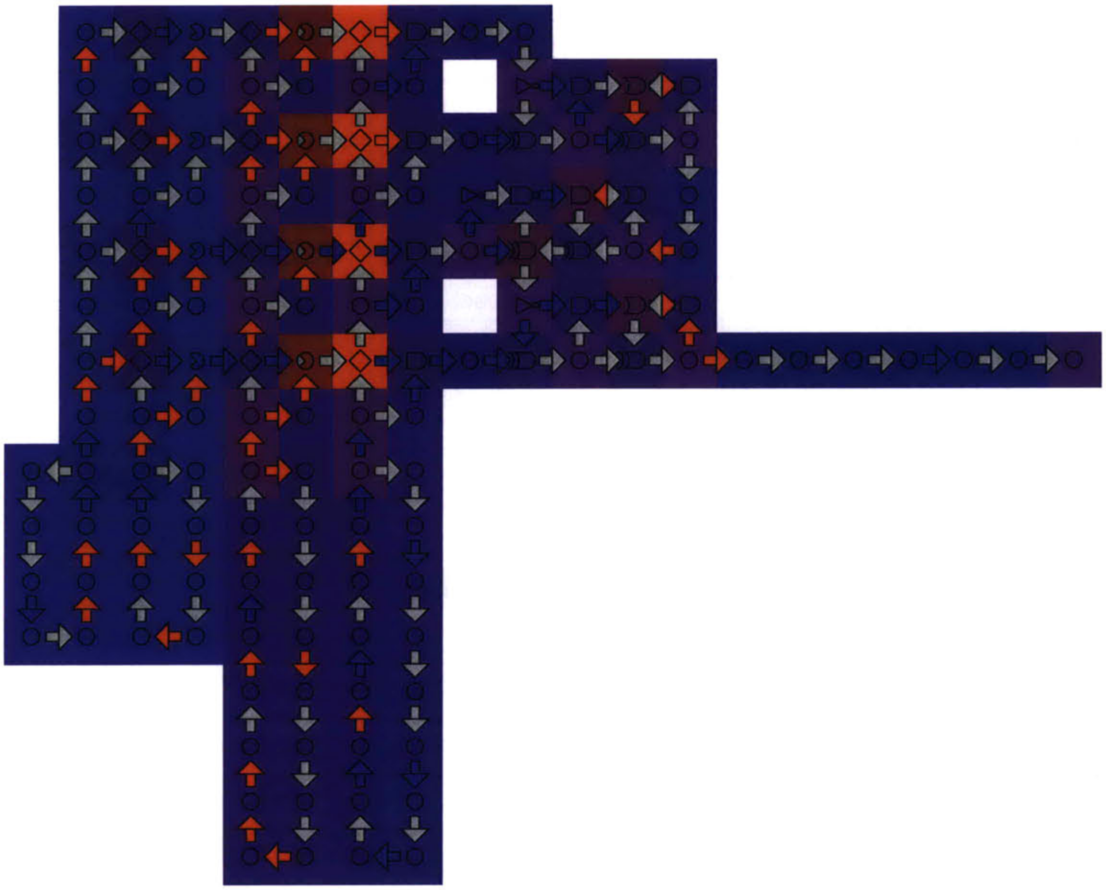

Figure 6-4: Heat Map of a Simple Multiplier

Figure 6.3 shows a simple multiplier where gates have been colored based on energy consumption. Although some gates consume more power per cycle (the crossover consuming power for both directions in particular) we can see the boundaries between the domains.

While being able to programmatically cross these domains does work in the majority of cases one might have, it remains difficult to implement asynchronous interactions between modules. For example, imagine implementing a multi-agent system where agent interactions involved unpredictably sending messages to each other. Ideally it shouldn't be necessary to send any data unless a new message is ready. However, with only copy and delete gates it is impossible to construct a circuit that will, for example, accept the first available message. This is largely by design to ensure that there are no possible race conditions. It would be plausible to create a distributed "clock" signal and have agents send null messages, but this could require substantial design work for the agents and might involve a significant power overhead for the null messages. It would also work poorly if one group of cells was running substantially slower than the rest (due to manufacturing or power supply variations for example). However, many of the obvious gates one might use are difficult to use in any way without making assumptions about the timing of the hardware.

6.3.1 POLL Gate

We have considered many potential new gate types that would enable an asynchronous merge, however often the designs need to make assumptions about the timing of the hardware to guarantee correctness. Even designs that initially seem correct can have critical issues that are only exposed with specific update patterns. Of these gates the one that seems the most promising, though not very efficient, I call a POLL gate. Like COPY and DELETE it is a two input gate with a control and data input. When the control input is T it emits a T token if there is a token at the input or a F token if there is no token. When the control input is F it waits until a token arrives on the data input and then consumes and emits that token. This gate appears to function correctly in many cases, but more thorough examination is needed.

6.4 Lack of Wires: Transport Cell

Although one of the key goals of ALA is to expose the cost of wiring, wires are still very good. The constraint that tokens must be buffered at every cell is likely far too conservative. An obvious fix would be to add a gate that is literally just a wire, but only so many of these could be chained together before delays became impractical or crosstalk had substantial risk of causing errors. For a given technology with a known RC delay of the wiring and buffer drive strength it is possible to compute an optimal spacing between buffers. A good compromise would be to allow the designer to specify arbitrarily long wires with a single post-processing step to insert buffers at the optimal spacing. While this somewhat violates the spirit of the ALA design it would likely offer substantial performance gains. Determining exactly what kinds of spacings are optimal in typical technologies is a topic for future work.

Chapter 7

Conclusion

If you read and understand chapter 2 you will be an ASIC designer. While you may not be capable of designing general ASICs, this thesis presents all of the ingredients necessary to go from an ALA design to taping out an ASIC. Chapter 3 and the appendixes describe some basic building blocks to help build advanced ALA designs. With some minor work to migrate to a specific process technology, the designs in chapter 4 can be used to fabricate an ASICs that implements any ALA program. The simulation methods from chapter 5 can estimate the performance of ALA designs almost as accurately and much faster than current state-of-the-art general purpose tools. Though performance of most ALA designs are currently inferior to full custom designs, chapter 6 presents several ideas that will minimize that gap.

There are weaknesses in the current ALA model that need to be fixed. The addition of a wire cell would take minimal effort but likely help even simple designs. As more complicated designs are developed the addition of an asynchronous merge operation (perhaps using the POLL gate described in section 6.3.1) will likely be worth the potential design complications. Although it is larger than what is documented here, the current library of ALA designs will need to be expanded.

In considering ALA is a substitute for traditional models of hardware there is significant potential to explore new high-level languages. The von Neumann processor has been the dominant model of computation for the last 50 years of technological progress. With current computers well into the realm of diminishing returns on optimizations there is a significant opportunity in trying to expose the full capabilities of modern fabrication processes. While it has weaknesses as noted above, this thesis is a step closer in that direction. Using ALA, as shown here, to align computational and physical models presents an oppertunity to make the design of ASICs substantially easier by making design verification easier, reducing tape-out costs by reusing masks, increasing yield, and enabling portability between processes and beyond silicon.

Appendix A

Hierarchical Design Example: Scalable Encryption Algorithm Round

```
from part import part

class line(part):
  def __init__(self, ala, start, dx, dy, len, dir, parent=None, values=[]):
    self.start = start
    self.dx = dx
    self.dy = dy
    self.len = len
    self.dir = dir

    part.__init__(self, ala, (0,0), parent=parent, mirror_x=False, mirror_y=False)
    self.initial_values = values

```

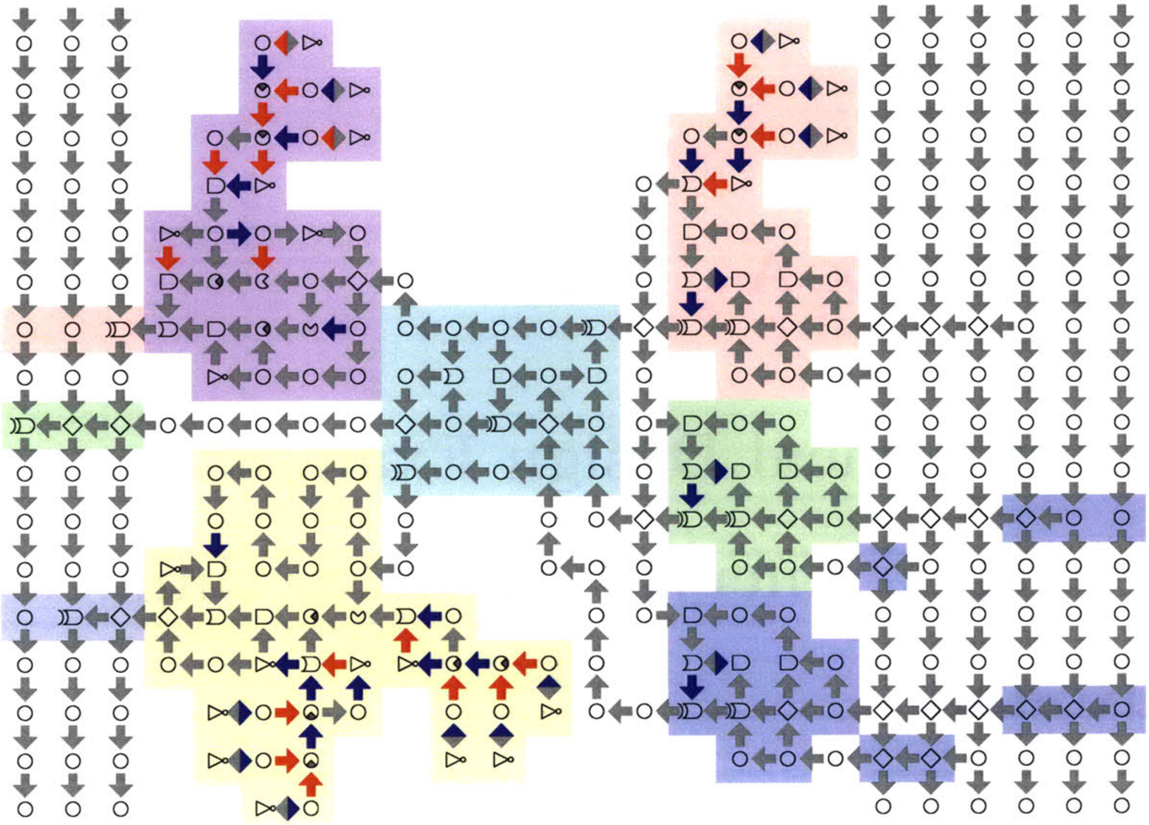

Figure A-1: Scalable Encryption Algorithm Round

```

    def build(self):
      self.values = list(self.initial_values)
      x = self.start[0]
      y = self.start[1]
      dx = self.dx
      dy = self.dy

      for i in range(0, self.len):
        self.make_wire((x + dx*i, y+dy*i), self.dir, init=self.
            next_val())

    def next_val(self):
```

```
    if len(self.values) > 0:
      v = self.values.pop()
    else:
      v = 'x'
    return v


class bit_rotate_r(part):
  def __init__(self, ala, location, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)

  def build(self):
    r, c = 0, 0

    self.add(c,r,'wire', 'e');c+=1
    self.add(c,r,'nand', 'ee');c+=1
    self.add(c,r,'wire', 'e0');c+=1
    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'nand', 'ww');c+=1

    r-=1; c=0

    self.add(c,r,'cross', 'wn');c+=1
    self.add(c,r,'wire', 'w');c+=1
    self.add(c,r,'delete', 'wn1');c+=1
    self.add(c,r,'copy', 'wn');c+=1
    self.add(c,r,'and', 'wn1');c+=1

```

```
        r-=1; c=0

        self.add(c,r,'wire', 'n');c+=1
        self.add(c,r,'delete', 'nw0');c+=1
        self.add(c,r,'copy', 'ws');c+=1
        self.add(c,r,'and', 'ws␣');c+=1
        self.add(c,r,'or', 'wn');c+=1

        r-=1; c=0

        self.add(c,r,'wire', 'n');c+=1
        self.add(c,r,'wire', 'w');c+=1
        self.add(c,r,'wire', 'w');c+=1
        self.add(c,r,'nand', 'ww');c+=1

        r+=7; c=1
        self.add(c,r,'nand', 'ee');c+=1
        self.add(c,r,'wire', 'w1');c+=1
        r-=1; c=0
        self.add(c,r,'nand', 'ee');c+=1
        self.add(c,r,'wire', 'w0');c+=1
        self.add(c,r,'copy', 'n0w1');c+=1
        r-=1; c=0
        self.add(c,r,'nand', 'ee');c+=1
        self.add(c,r,'wire', 'w1');c+=1
        self.add(c,r,'copy', 'n1w0');c+=1
        self.add(c,r,'wire', 'w');c+=1
        r-=1; c=2
        self.add(c,r,'nand', 'n1n');c+=1
        self.add(c,r,'and', 'n1w0');c+=1
```

```

    self.build_children()

#output = 0111 1111 -->
class control_generator(part):
  def __init__(self, ala, location, swap_out=False, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)
    self.swap_output_location=swap_out

  def build(self):

    if self.swap_output_location:
        c_start = 0
    else:
        c_start = 1

    r=3; c=c_start
    self.add(c,r,'wire', 'e0');c+=1
    self.add(c,r,'nand', 'ww');c+=1
    r-=1; c=c_start
    self.add(c,r,'copy', 'n1e1');c+=1
    self.add(c,r,'wire', 'e0');c+=1
    self.add(c,r,'nand', 'ww');c+=1
    r-=1; c=c_start-1
    self.add(c,r,'wire', 'e');c+=1
    self.add(c,r,'copy', 'n0e1');c+=1
    self.add(c,r,'wire', 'e0');c+=1
    self.add(c,r,'nand', 'ww');c+=1
    r-=1; c=c_start-1
    if self.swap_output_location:
        self.add(c,r,'nand', 'n0n');c+=1
        self.add(c,r,'or', 'n0w1');c+=1
    else:
        self.add(c,r,'or', 'n0e1');c+=1
        self.add(c,r,'nand', 'n0n');c+=1

    self.build_children()

class bit_rotate_l(part):
  def __init__(self, ala, location, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)

  def build(self):
    r, c = 0, 0

    c += 2
    self.add(c,r,'nand', 'w0w');c+=1
    self.add(c,r,'wire', 'w');c+=1
    self.add(c,r,'wire', 'w');c+=1
    r-=1; c=0

    del_in_loc = (c-1,r)
    self.add(c,r,'delete', 'sw');c+=1
    copy_in_loc = (c,r+1)
    self.add(c,r,'copy', 'wn');c+=1
```

```
        self.add(c,r,'and', 'wn');c+=1
        self.add(c,r,'or', 'ws');c+=1
        self.add(c,r,'cross', 'wn');c+=1
        r-=1; c=0

        self.add(c,r,'wire', 'w');c+=1
        self.add(c,r,'wire', 's');c+=1
        self.add(c,r,'wire', 'w');c+=1
        self.add(c,r,'and', 's0e');c+=1
        self.add(c,r,'nand', 'nn');c+=1
        r-=1; c=0

        self.add(c,r,'wire', 'n');c+=1
        self.add(c,r,'wire', 's');c+=1
        self.add(c,r,'wire', 'n');c+=1
        self.add(c,r,'wire', 's');c+=1
        r-=1; c=0

        self.add(c,r,'wire', 'n');c+=1
        self.add(c,r,'wire', 'w');c+=1
        self.add(c,r,'wire', 'n');c+=1
        self.add(c,r,'wire', 'w');c+=1
        r-=1; c=0

        control_generator(self.ala, del_in_loc, parent=self,
            rotate=90);
        control_generator(self.ala, copy_in_loc, swap_out=True,
            parent=self, rotate=0);

        self.build_children()
```

```

#permutes side input as if word rotate was done before passsing in input (and output was rotated back)
class xor_and_rotate_block(part):
  def __init__(self, ala, location, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)

  def build(self):
    r, c = 0, 0

    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'xor', 'ne');c+=1
    r-=1; c=0
    self.add(c,r,'xor', 'ne');c+=1
    self.add(c,r,'cross', 'ne');c+=1
    self.add(c,r,'cross', 'ne');c+=1
    r-=1; c=0
    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'xor', 'ne');c+=1
    self.add(c,r,'cross', 'ne');c+=1
    r-=1; c=0

    self.build_children()


#permutes side input as if word rotate was done before passsing in input (and output was rotated back)
```

```
class xor_2_0(part):
  def __init__(self, ala, location, parent=None, mirror_x=
      False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y,
        rotate, parent=parent, color=color)
  def build(self):
    r, c = 0, 0
    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'xor', 'ne');c+=1
    self.build_children()

class xor_0_1(part):
  def __init__(self, ala, location, parent=None, mirror_x=
      False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y,
        rotate, parent=parent, color=color)
  def build(self):
    r, c = 0, 0
    self.add(c,r,'xor', 'ne');c+=1
    self.add(c,r,'cross', 'ne');c+=1
    self.add(c,r,'cross', 'ne');c+=1
    self.build_children()

class xor_1_2(part):
  def __init__(self, ala, location, parent=None, mirror_x=
      False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y,
        rotate, parent=parent, color=color)
  def build(self):
    r, c = 0, 0
    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'xor', 'ne');c+=1
    self.add(c,r,'cross', 'ne');c+=1
    self.build_children()

class fanout_1(part):
  def __init__(self, ala, location, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)
  def build(self):
    r, c = 0, 0
    self.add(c,r,'cross', 'ne');c+=1
    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'wire', 'n');c+=1
    self.build_children()

class fanout_2(part):
  def __init__(self, ala, location, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)
  def build(self):
    r, c = 0, 0
    self.add(c,r,'cross', 'ne');c+=1
    self.add(c,r,'cross', 'ne');c+=1
    self.add(c,r,'wire', 'n');c+=1
    self.build_children()

```

```

class fanout_block(part):
  def __init__(self, ala, location, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)

  def build(self):
    r, c = 0, 0
    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'wire', 'n');c+=1
    r-=1; c=0
    self.add(c,r,'cross', 'ne');c+=1
    self.add(c,r,'wire', 'n');c+=1
    self.add(c,r,'wire', 'n');c+=1
    r-=1; c=0
    self.add(c,r,'cross', 'ne');c+=1
    self.add(c,r,'cross', 'ne');c+=1
    self.add(c,r,'wire', 'n');c+=1

    self.build_children()

#adder with minimal carry loop and overflow prevention
class adder(part):
  def __init__(self, ala, location, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)

  def build(self):
    c, r = 0+1, 0
    self.add(c, r, 'wire', 'w'); c += 1
    self.add(c, r, 'wire', 'w'); c += 1

    c, r = 0, r-1
    self.add(c, r, 'wire', 'w'); c += 1
    self.add(c, r, 'cross', 'wn'); c += 1
    self.add(c, r, 'xor', 'wn'); c += 1
    self.add(c, r, 'xor', 'ws0'); c += 1

    c, r = 0, r-1
    self.add(c, r, 'wire', 'n'); c += 1
    self.add(c, r, 'and', 'wn'); c += 1
    self.add(c, r, 'and', 'ne0'); c += 1
    self.add(c, r, 'or', 'ws'); c += 1

    c, r = 0+1, r-1
    self.add(c, r, 'wire', 'n'); c += 1
    self.add(c, r, 'wire', 'w'); c += 1
    carry_kill = (c, r-1)
    self.add(c, r, 'and', 'ws'); c += 1

    control_generator(self.ala, carry_kill, parent=self, rotate=180);

    self.build_children()

#adder with minimal carry loop and overflow prevention
class adder_no_carry_gen(part):
  def __init__(self, ala, location, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)

  def build(self):
    c, r = 0+1, 0
    self.add(c, r, 'wire', 'w'); c += 1
    self.add(c, r, 'wire', 'w'); c += 1

    c, r = 0, r-1
    self.add(c, r, 'wire', 'w'); c += 1
    self.add(c, r, 'cross', 'wn'); c += 1
    self.add(c, r, 'xor', 'wn'); c += 1
    self.add(c, r, 'xor', 'ws0'); c += 1

    c, r = 0, r-1
    self.add(c, r, 'wire', 'n'); c += 1
    self.add(c, r, 'and', 'wn'); c += 1
    self.add(c, r, 'and', 'ne0'); c += 1
    self.add(c, r, 'or', 'ws'); c += 1

    c, r = 0+1, r-1
    self.add(c, r, 'wire', 'n'); c += 1
    self.add(c, r, 'wire', 'w'); c += 1
    carry_kill = (c, r-1)
    self.add(c, r, 'and', 'we'); c += 1

    self.build_children()

```

```

class s_box(part):
  def __init__(self, ala, location, parent=None, mirror_x=
      False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y,
        rotate, parent=parent, color=color)

  def build(self):
    r, c = 0, 0

    self.add(c,r,'xor', 'ws');c+=1
    self.add(c,r,'wire', 'w');c+=1
    self.add(c,r,'wire', 'w');c+=1
    self.add(c,r,'wire', 'w');c+=1
    self.add(c,r,'wire', 'w');c+=1
    r-=1; c=0

    self.add(c,r,'and', 'se');c+=1
    self.add(c,r,'wire', 's');c+=1
    self.add(c,r,'and', 'wn');c+=1
    self.add(c,r,'or', 'sn');c+=1
    self.add(c,r,'wire', 'w');c+=1
    r-=1; c=0

    self.add(c,r,'wire', 's');c+=1
    self.add(c,r,'cross', 'ws');c+=1
    self.add(c,r,'xor', 'wn');c+=1
    self.add(c,r,'wire', 'w');c+=1
    self.add(c,r,'cross', 'wn');c+=1
    r-=1; c=0
    self.add(c,r,'wire', 's');c+=1
    self.add(c,r,'wire', 's');c+=1
    self.add(c,r,'wire', 'w');c+=1
    self.add(c,r,'wire', 'w');c+=1
    self.add(c,r,'xor', 'wn');c+=1
    r-=1; c=0

    self.build_children()


class sea_round(part):
  def __init__(self, ala, location, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)

  def build(self):
    r, c = 0, 0
    xor_2_0(self.ala,(r,c+1), parent=self, color='#ffe0e0')
    xor_0_1(self.ala,(r,c-1), parent=self, color='#e0ffe0')
    xor_1_2(self.ala,(r,c-5), parent=self, color='#e0e0ee')

    r += 3
    bit_rotate_r(self.ala,(r+4,c+3), parent=self, mirror_x=True, color='#ffe0ff')
    bit_rotate_l(self.ala,(r+4,c-6), parent=self, mirror_x=True, mirror_y=True, color='#ffffe0')
    s_box(self.ala, (r+9, c+1), parent=self, mirror_x=True, color='#e0ffff')
```

```
        adder(self.ala, (r+14, c-0), parent=self,rotate=180, color='#ffe0e0')
        adder_no_carry_gen(self.ala, (r+14, c-4), parent=self, rotate=180, color='#e0ffe0')
        adder_no_carry_gen(self.ala, (r+14, c-8), parent=self, rotate=180, color='#e0e0ff')
        fanout_1(self.ala, (r+18, c-3), parent=self,rotate=0, color='#e0e0ff')
        fanout_2(self.ala, (r+18, c-7), parent=self,rotate=0, color='#e0e0ff')

        fanout_1(self.ala, (r+15, c-4), parent=self,rotate=0, color='#e0e0ff')
        fanout_2(self.ala, (r+15, c-8), parent=self,rotate=0, color='#e0e0ff')

        for i in range(-9, 8):
            #left side path skipping over xor blocks
            if i not in [-5, -1, 1]:
                self.add(0, i, 'wire', 'n')
                self.add(1, i, 'wire', 'n')
                self.add(2, i, 'wire', 'n')

            #key path
            if i not in [-7, -3, 1]:
                self.add(18, i, 'wire', 'n')
                self.add(19, i, 'wire', 'n')
                self.add(20, i, 'wire', 'n')
            elif i not in [-4, -8]:
                self.add(18, i, 'cross', 'en')
                self.add(19, i, 'cross', 'en')
                self.add(20, i, 'cross', 'en')

            #right side path
            if i not in [-7, -3]:
                self.add(21, i, 'wire', 'n')
                self.add(22, i, 'wire', 'n')
                self.add(23, i, 'wire', 'n')

        #wire from sbox to xor (signal 1)
        for i in range(3, 8):
            self.add(i, -1, 'wire', 'e')

        #wire from sbox to rotate left (signal 2)
        self.add(8, -3, 'wire', 'n')
        self.add(8, -4, 'wire', 'n')

        #wire from sbox to rotate right
        self.add(8, 2, 'wire', 's')

        #control signal path for adders
        for i in range(-5, 4):
            if i not in [-3, 1]:
                self.add(13, i, 'wire', 'n')
            else:
                self.add(13, i, 'cross', 'ne')
        self.add(13, 4, 'wire', 'e')

        #route signal 1 from key addition to sbox
        self.add(12, -3, 'wire', 'e')
```

```

    #route signal 2 from key addition to sbox
    self.add(11, -3, 'wire', 's')
    self.add(11, -4, 'wire', 'e')
    self.add(12, -4, 'wire', 's')
    self.add(12, -5, 'wire', 's')
    self.add(12, -6, 'wire', 's')
    self.add(12, -7, 'wire', 'e')
    self.add(13, -7, 'wire', 'e')

    #wires from key to addition blocks
    self.add(17, 0, 'wire', 'e')
    self.add(17, -4, 'wire', 'e')
    self.add(17, -8, 'wire', 'e')

    self.build_children()

def build(my_ala, my_view):
    test = sea_round(my_ala, (0, 0))
    test.build()
```

Appendix B

Parametric Design Example: Arbitrary Period Oscillator

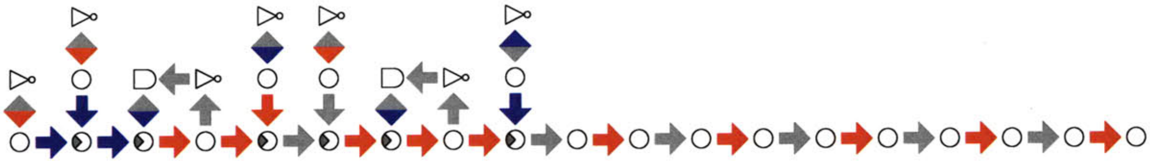

Figure B-1: 42 State Oscillator

```python
from part import part
from part_lib import line

class osc(part):
  def __init__(self, ala, location, value, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y, rotate, parent=parent, color=color)
    self.value = value

  def build(self):
    assert self.value >= 2

    s = 0
    v = self.value

    while v > 2:
        if v & 1:
            v -= 1
            self.add(s-1,1,"and","se")
            self.add(s-1,0,"copy","wn0")
            self.add(s,1,"nand","s0s")
            self.add(s,0,"wire","w")
            s -= 2

        if v > 2:
            v = v >> 1
            self.add(s,2,"nand","ss")
            self.add(s,1,"and", "n1n")
            self.add(s,0,"copy", "wn")
            s -= 1

    self.add(s,1,"nand","ss")
    self.add(s,0,"and","n1n")

    self.build_children()

class test(part):
  def __init__(self, ala, location, parent=None, mirror_x=False, mirror_y=False, rotate=0, color=None):
    part.__init__(self, ala, location, mirror_x, mirror_y,
        rotate, parent=parent, color=color)

  def build(self):
    osc(self.ala, (0, 0), 42, parent=self)
    line(self.ala, (1, 0), 1, 0, 10, 'w', parent=self, values=[], cap=False)
    self.build_children()


def build(my_ala, my_view):
    t = test(my_ala, (0, 0))
    t.build()
```

Bibliography

[1] V. Agarwal, MS Hrishikesh, S.W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. *ACM SIGARCH Computer Architecture News*, 28(2):259, 2000.

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

[3] K. Chen. *Circuit design for logic automata.* PhD thesis, Massachusetts Institute of Technology, 2009.

[4] Kailiang Chen, Forrest Green, and Neil Gershenfeld. Asynchronous logic automata asic design. Preprint, 2010.

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

[6] D. Dalrymple. Asynchronous Logic Automata. Master's thesis, Massachusetts Institute of Technology, 2008.

[7] D.A. Dalrymple, N. Gershenfeld, and K. Chen. Asynchronous logic automata. *Automata-2008. Theory and Applications of Cellular Automata*, pages 313–320.

[8] W. Daniel Hillis. New computer architectures and their relationship to physics or why computer science is no good. *International Journal of Theoretical Physics*, 21(3):255–262, 1982.

[9] AKK Dewdney and AK Dewdeny. *Tinkertoy Computer and Other Machinations.* WH Freeman & Co. New York, NY, USA, 1993.

[10] M. Gardner. Mathematical games: The fantastic combinations of John Conways new solitaire game Life. *Scientific American*, 223(4):120–123, 1970.

[11] N. Gershenfeld, D. Dalrymple, K. Chen, A. Knaian, F. Green, E.D. Demaine, S. Greenwald, and P. Schmidt-Nielsen. Reconfigurable asynchronous logic automata:(RALA). *ACM SIGPLAN Notices*, 45(1):1–6, 2010.

[12] Scott Greenwald, Bernhard Haeupler, and Neil Gershenfeld. Mathematical operations in asynchronous logic automata. Preprint, 2010.

[13] Jia Lee, Ferdinand Peper, Susumu Adachi, Kenichi Morita, and Shinro Mashiko. Reversible computation in asynchronous cellular automata. In *UMC '02: Proceedings of the Third International Conference on Unconventional Models of Computation*, pages 220–229, London, UK, 2002. Springer-Verlag.

[14] N. Margolus, T. Toffoli, and G. Vichniac. Cellular-automata supercomputers for fluid-dynamics modeling. *Physical Review Letters*, 56(16):1694–1696, April 1986.

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

[16] M. Shams, JC Ebergen, and MI Elmasry. Modeling and comparing CMOS implementations of the C-element. *IEEE transactions on very large scale integration (VLSI) systems*, 6(4):563–567, 1998.

[17] G. Taubes. The rise and fall of Thinking Machines. *Inc. Magazine*, September 1995.

[18] C.J. Vieri. *Pendulum: A reversible computer architecture*. PhD thesis, Citeseer, 1995.

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

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

[21] E. Yahya and M. Renaudin. Qdi latches characteristics and asynchronous linear-pipeline performance analysis. *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 583–592.

[22] S.G. Younis and T.F. Knight Jr. Practical implementation of charge recovering asymptotically zero power CMOS. In *Proceedings of the 1993 symposium on Research on integrated systems*, page 250. MIT Press, 1993.