Circuit Design for Logic Automata

by

Kailiang Chen

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2009

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

Accepted by . . . .
Terry P. Orlando
Chairman, EECS Committee on Graduate Students

Circuit Design for Logic Automata

by

Kailiang Chen

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

The Logic Automata model is a universal distributed computing structure which pushes parallelism to the bit-level extreme. This new model drastically differs from conventional computer architectures in that it exposes, rather than hides, the physics underlying the computation by accomodating data processing and storage in a local and distributed manner. Based on Logic Automata, highly scalable computing structrues for digital and analog processing have been developed; and they are verified at the transistor level in this thesis.

The Asynchronous Logic Automata (ALA) model is derived by adding the temporal locality, i.e., the asynchrony in data exchanges, in addition to the spacial locality of the Logic Automata model. As a demonstration of this incrementally extensible, clockless structure, we designed an ALA cell library in 90 nm CMOS technology and established a "pick-and-place" design flow for fast ALA circuit layout. The work flow gracefully aligns the description of computer programs and circuit realizations, providing a simpler and more scalable solution for Application Specific Integrated Circuit (ASIC) designs, which are currently limited by global contraints such as the clock and long interconnects. The potential of the ALA circuit design flow is tested with example applications for mathematical operations.

The same Logic Automata model can also be augmented by relaxing the digital states into analog ones for interesting analog computations. The Analog Logic Automata (AnLA) model is a merge of the Analog Logic principle and the Logic Automata arhitecture, in which efficient processing is embedded onto a scalable construction. In order to study the unique property of this mixed-signal computing structure, we designed and fabricated an AnLA test chip in AMI 0.5μm CMOS technology. Chip tests of an AnLA Noise-Locked Loop (NLL) circuit as well as application tests of AnLA image processing and Error-Correcting Code (ECC) decoding, show large potential of the AnLA structure.

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

Acknowledgments

I would like to acknowledge the support of MIT's Center for Bits and Atoms and its sponsors.

Thank you to my thesis supervisor, Neil Gershenfeld, for his intriguing guidance and encouragement over the past two years. His wide knowledge span and deep insight have been inspiring me to keep learning and thinking; and his openness to ideas has encouraged me to always seek better solutions to problems I encounter. Thank you for directing the Center for Bits and Atoms and the Physics and Media Research Group, which provide me great intellectual freedom and wide vision. He has been a wonderful mentor and friend who helped me smoothly adapt to the research life in MIT. I cannot overstate my gratitude for his support.

Thank you to those who have worked with me on the Logic Automata research team, including Kenny Cheung, David Dalrymple, Erik Demaine, Forrest Green, Scott Greenwald, Mariam Hoseini, Scott Kirkpatrick, Ara Knaian, Luis Lafeunte Molinero, Mark Pavicic, and Chao You. It is an exciting work experience and I enjoy every discussion with them.

Thank you to Physics and Media Group members: Manu Prakash, Amy Sun, Sherry Lassiter, Ben Vigoda, Joe Murphy, Brandon Gorham, Susan Bottari, John Difrancesco, and Tom Lutz. Their diligent work and kind help have been an indispensable source of support for my work and life here.

Thank you to Jonathan Leu, Soumyajit Mandal, Andy Sun, Yogesh Ramadass, Rahul Sarpeshkar, and Anantha Chandrakasan, who have taught me many useful techniques in circuit design and have offered generous help throughout my study and research.

Thank you to all my friends for their encouragement to me and share of happiness with me whenever possible.

Finally thank you to my parents for their love and confidence in whatever I do. Without them I would not be.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Presently, computing systems are usually built with a hierachical and modular approach that tries to hide away the underlying hardware from the upper level software. The venerable von Neumann model [78] is based on such abstraction and has been the dominant architecture choice of almost all computers over the past fifty years. Meanwhile, the hardware scaling trend following Moore's Law [50, 4] has been successful in keeping up with the ever-growing need for better computer performance and lower cost. This exponential technological growth thus plays a critical role in maintaining the von Neumann architecture as a unchallenged canonical way of building computers. However, as the transistor size shrinks beyond 45nm and billions of transistors are crammed onto one chip, challenges like clock/power distribution, interconnection, and heat dissipation are becoming more and more pronounced; and Moore's Law is reaching a point where any additional technology scaling will be accompanied by excessively large design efforts [51]. As a result, people are now eagerly searching for alternative driving forces to back up the transistor scaling. Architectural change from the von Neumann model, which is constrained by many physical limits [56], to some parallelized structures is believed to be promising for future performance increases.

In this work, we push parallelism to extremes to investigate the Logic Automata model as a universal computing structure. The Logic Automata model is an array in which each cell contains a logic gate, stores its state and interacts with its neighboring cells, locally providing the components needed for computational universality [19].

Based on the theoretical and algorithmical developments of this computational model [19, 7, 10, 75, 6, 9, 24], we build transistor-level hardware implementations to examine the computational potential of the Logic Automata model. Before we get to the circuit design for Logic Automata, we first give a review of the past work that leads to the Logic Automata model.

1.1 Past Work

The idea of Logic Automata stems from the Cellular Automata (CA) model [77] proposed by von Neumann. The CA model was designed as a mathematical tool to explore the theory of self-replicating machines; and it came after von Neumann's famous Electronic Discrete Variable Automatic Computer (EDVAC) [78], which is believed to be the ancestor of "the von Neumann architecture." Following the original CA concept, more works can be found in the literature ellaborating on the CA theory [62, 5, 27]. And works such as [2] and more recently the construction of Rule 110 [46] took on a slightly different direction to prove CA's computational universality.

The theoretical works led to useful applications. Margolus and Toffoli, among others, built the Cellular Automata Machine 8 (CAM8) to simulate the cellular automata and other physical systems [20, 43, 71, 63]. Although the hardware was specialized for distributed system simulation, CAM8 was not made with truly extensible cellular structure. Nevertheless, CAM8-emulated cellular automata already showed excellent modeling power for complex physical systems.

It is easy to notice the similarities between the CA structure and many other well known computing media - especially the Field Programmable Gate Arrays (FPGAs) and their variants. The FPGAs include such different flavors as the sum-product networks [61], the sea-of-gates style systems [23], the acyclic, memory-less structures [47], and today's complex, heterogeneous commercial products. Although every FPGA contains a grid of cells with rich interconnections, neither FPGA type is strictly local, nor arbitrarily extensible. The systolic array is another parallel architecture with a network arrangement of processing units [32]. But due to its piped connectivity and directional information propagation between the cells, such a system could not be considered fully scalable either.

Therefore, the CA model is an overly complicated model to implement, as it imposes too many constraints between space and time, in spite of the fact that it is simple in mathematical form and suitable for theoretical research. The CAM8 machine, being an excellent CA simulator, does not have an extensible cellular structure. The FPGAs and systolic arrays have global interconnects and global data flows that introduce non-local dependencies, which again betrays system extensibility. To fill the gaps in the aforementioned models, the Logic Automata model is invented, which faithfully reflects the physics of the system.

In the next section, we continue to elaborate on the motivation for inventing Logic Automata.

1.2 Motivation

We were initially seeking a simple but universal computational model that allows maximum flexibility and extensibility. The original CA model naturally became interesting to us because it is the simplest among the computationally universal models with completely local, scalable construction. But the CA model remains more useful as a mathematical tool for computation theory than a general processing unit for any practical purposes. This is because even basic Boolean logic functions such as NAND or XOR will require many CA cells to implement.

For example, Banks proposed a universal two-state CA model with rectangular nearest neighbor connections [2, 8]. The model operates on three simple rules: a "0" cell surrounded by three or four "1" cells becomes a "1"; a "1" cell which is neighbored on two adjacent sides (north and west, north and east, south and west, or south and east) by "1" cells and on the other sides by "0" cells becomes a "0"; and finally, that any other cell retains its previous state. By implementing a basic universal logic function $\neg A \wedge B$, Banks' CA can be a universal computation machine. However, this primitive function is implemented in a 13×19 rectangular patch of cells, in which 83 cells are actively used [8]. This model is too complicated for any practical purpose.

It is evident that we should increase computational complexity per cell in practical use of a CA-like model. Logic Automata solve this problem by incorporating Boolean logic directly in the cells instead of deriving the logic needed from ensembles of CA cells. Not only would the number of cells needed to achieve a practical design drastically decrease, but the hardware realization would be simpler also. For any digital design that is expressed in terms of a set of complex Boolean logic computations, Logic Automata save one level of mapping effort by directly utilizing two-input Boolean logic functions as building blocks. To be complete, a Logic Automata cell also stores one bit of state just as a CA cell does; and capability other than logic computation is added to the Logic Automata framework to ensure general-purpose control and manipulation.

Furthermore, asynchronous operation is explicitly enforced in Logic Automata to produce the model of Asynchronous Logic Automata (ALA), for the reason that asynchronous cell updates mimic the limited information propagation speed in any physical systems [8, 19]. Additionally, only with the elimination of the global clock and the completely autonomous cell operations, can a truly scalable architecture be maintained.

In addition, we are not confining ourselves solely to the digital computation domain; analog computation can also be incorporated by relaxing the Boolean state of every Logic Automata cell into analog states. The resulting Analog Logic Automata (AnLA) exploit the probablistic Analog Logic principle and find a broad range of applications in signal synchronization, decoding and image processing.

1.3 Thesis Outline

Works in [19, 10, 8, 75, 9, 24] have laid the theoretical and algorithmatic foundation of the Logic Automata model as well as many of its variants. In this thesis, the focus will be on the hardware realization and evaluation of such models [7, 6].

Circuit design and testing for both Asynchronous Logic Automata (ALA) and

Analog Logic Automata (AnLA) are presented. In Chapter 2, we provide background information for the ALA design, including the development of the Logic Automata model and its variants, ALA algorithms, and fundamentals about asynchronous circuit design. Chapter 3 is dedicated to the design of ALA circuits, in which the ALA cell library with a brand new chip design flow is described. In Chapter 4, we test our ALA cells with ALA applications and show the unique advantage of our ALA design flow. In Chapter 5 and Chapter 6, we introduce an analog-computation based Logic Automata, where circuit design and application test are described respectively. We finally summarize our work in Chapter 7.

Chapter 2

Background Information for Asynchronous Logic Automata Design

The Asynchronous Logic Automata (ALA) model sits at the center of the whole Logic Automata development effort. ALA is a computationally universal, simple, and scalable digital computation medium. Before we give a detailed description of hardware design of the ALA model, a brief review of the derivation and definition of the model is presented, which moves from the "plain" Logic CA cells, to the ALA, and to the idea of Reconfigurable Asynchronous Logic Automata (RALA) that adds in-band programmability to the ALA. The Analog Logic Automata (AnLA) model is also derived based on the Logic CA cells and relaxes digital states into analog states. Moreover, the basics of asynchronous circuit design are also summarized briefly in this chapter.

2.1 The Evolution of the Logic Automata Family

2.1.1 Logic CA Cells

A Logic CA cell is a one-bit digital processor which conceptually incorporates a two-input Boolean Logic gate and a one-bit register storing the cell state. A set of selected Logic CA cells is equivalent to a computationally universal CA model, but could be more useful because each Logic CA cell has stronger computation power without considerable hardware complexity increase. A mathematical proof of the equivalence between the Logic CA cells and the cellular automata is given in [8], but readers could understand Logic CA intuitively by looking at the following examplary embodiment of Logic CA cells:

> A Logic CA cell has two inputs, one Boolean logic function and a one-bit state storage. Each cell input can be configured to be from one of the output state values of its four rectangular (North, West, South and East) neighbor cells. The Boolean logic function can be chosen from the function set: {NAND, XOR, AND, OR}. In each time step, a cell receives two input bits from its neighbors, performs the selected Boolean operation on these inputs, and sets its own state to the result.

As can be seen from the definition, the Logic CA cells are still defined to operate synchronously. Secondly, the Boolean function set is chosen such that the universal Boolean computation is ensured (strictly speaking, NAND operation alone suffices universal computation) and the ease of use is also considered by providing redundancy in logical functionalities.

2.1.2 Asynchronous Logic Automata

There are several reasons that let us consider introducing asynchrony into the Logic CA. First of all, Asynchronous Logic Automata (ALA) eliminate the global clock, which could become a bottleneck when a digital system scales up. As the technology is evolving and system designs are becoming more complicated, digital systems suffer more from a whole class of timing problems, including clock skew and interconnect speed limitation. The computation units are processing at such a fast speed that the speed of signal transmission is approaching its physical limit in order to keep up. As a result, huge efforts have been invested into clock distribution techniques and interconnect technologies to fight this physical limit. However, clock deskewing circuitry [18, 68] usually takes up a significant portion of a system's power budget, chip area and design time. And the non-conventional interconnect technologies, for instance the optical interconnect [34, 73], are still far from mature enough to push the physical speed limit to a higher level. In contrast, ALA circuits avoid such efforts completely as they require no global clock, dismissing the need for a clock distribution; and ALA cells only communicate locally, mitigating signal transmission delays over long interconnects.

Secondly, making the Logic CA asynchronous could eliminate the "delay lines." Delay lines exist in some synchronous Logic CA applications only to match multiple convergent paths so that their signals arrive at the same time under the global clock update. These would become unnecessary in an ALA application circuitry because the relative timing of the signals at the merging point is taken care of by the asynchronous communication protocol. This saves space, time, and power, and simplifies ALA algorithm design.

The asynchronous behavior can be added into Logic Automata by exploiting the Marked Graph formulation [52] in Petri net theory [55]. Similar construction of asynchronous circuits using Petri nets can be found in [12, 48], but not with a cellular architecture. In our ALA modeling, the global clock is removed and the state storage in each cell is replaced with a "token" [55] which is broadcast to its neighbor cells. Between each pair of cells that has data transmission, tokens are propagated as information bits. A token can be encoded by two data wires and represent 3 distinct states, i.e., a "0" token corresponding to a bit "0", a "1" token corresponding to a bit "1", and an "empty" token indicating empty state storage. Therefore, tokens reside on the edges between cells and each cell conceptually becomes stateless. For a Logic Automata cell to fire, it waits until its two input edges have tokens ready,

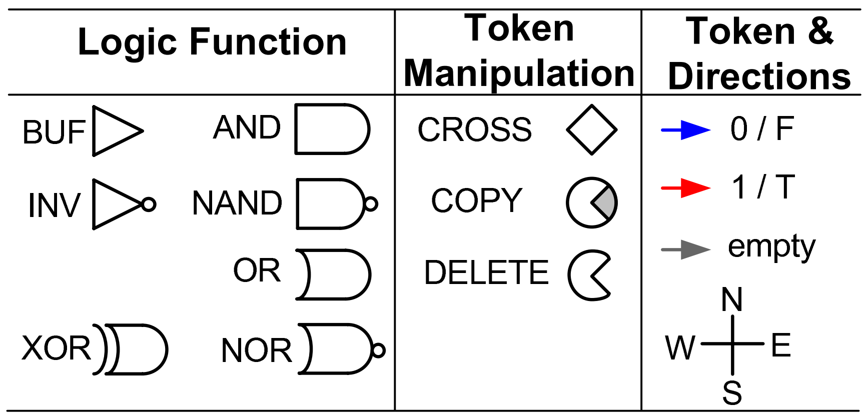


Figure 2-1: ALA cell types and definition.

i.e., non-empty tokens (tokens of either "0" or "1"). Then it consumes the input tokens (making tokens on input edges become "empty" tokens), performs its configured function, and puts the result token onto its output edges. As it is a Marked Graph, the behavior of this model is well-defined even without any assumptions regarding the timing of the computations, except that each computation will fire in some finite length of time after the preconditions are met [8]. From a circuit design point of view, such asynchronous operations can be robustly implemented with "handshake" protocols introduced later.

Figure 2-1 summarizes the ALA cells that we are going to implement and use in the ALA applications. As compared to the Logic CA definition in the last section, the rectangular interconnection is kept unchanged. But ALA cells communicate asynchronously rather than assuming a global clock and more Boolean logic functions are added to the cell library. Moreover, we add the CROSSOVER (or CROSS), COPY and DELETE cells for manipulating token propagation, creation and destruction.

2.1.3 Reconfigurable Asynchronous Logic Automata

Reconfigurable Asynchronous Logic Automata (RALA) are obtained by introducing reconfigurable property into each ALA cell to select the gate type and its input sources. Based on the ALA definition in Figure 2-1, RALA is defined with a selected cell function set and an additional "stem" cell, as shown in Figure 2-2. The nam-

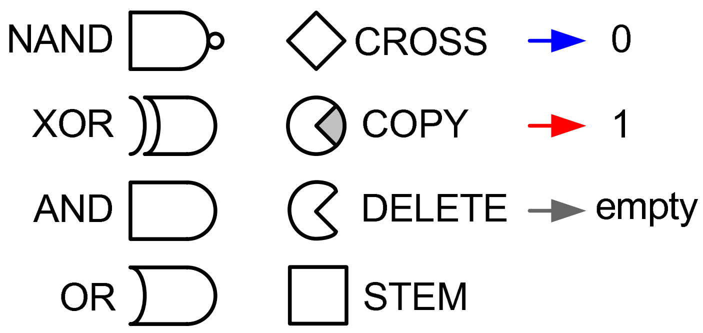


Figure 2-2: RALA cell types and definition.

ing clearly draws analogy from biology: similar to the universal folding behavior in molecular biology, in which a linear code is converted into a shape [57], the RALA model takes in instruction bit string to specify either the creation of a new stem cell in a relative folding direction to its input, or the configuration of the stem cell into one of the 7 other cell types [19]. The instruction bits are tokens streamed in along a linear chain of stem cells; the terminal stem cell at the end of the chain accumulates the bit string to form an instruction.

Just as the machinery of molecular biology is an operating system for life, RALA suggest an analogous way of information propagation to form an operating system and programmable computation. To program a patch of stem cells into a useful ALA application, we first fold a path for instruction streaming, and then as instructions are streamed in, stem cells are specified for appropriate gate types and inputs, one by one along the path. But after stem cells become active cells, they cannot fire before they have valid tokens on their inputs and empty connections on their outputs. This constraint could ensure that each cell turns on consistently without requiring any other kind of coordination.

A different way to look at the RALA model is that it is a multicore processor or an FPGA taken to the extreme. Each processor stores and computes one bit and its functionality is programmable. In addition, RALA push locality to extremes because both the programming phase and the processing phase adhere to strict spatial local rules. Therefore, the RALA model is the first model of computation to balance computation, data, and instructions, while simultaneously permitting efficient implementation in the physical world and enabling practical and efficient algorithmic design. This is both necessary and sufficient for computing to scale optimally according to the laws of physics [9].

A lot of modeling work has been established and all ALA applications could be easily laid out and run by streaming a bit string into a patch of stem cells [9]. In the near future, we will also implement RALA circuits on the transistor level.

2.1.4 Analog Logic Automata

The Logic Automata model is not confined to only digital computations. Analog Logic Automata (AnLA) could be of interest when it comes to statistical signal processing or any other kind of computation that is dealing with continuous numbers directly.

By storing a continuous value in each Logic Automata cell and deriving "soft" versions of Boolean logic functions according to the Analog Logic principle [75, 74, 66], AnLA could perform efficient analog computations and remain local in signal exchange. We will cover AnLA design and application in much more detail in Chapter 5 and Chapter 6.

2.2 Asynchronous Logic Automata Applications

2.2.1 Mathematical Operations

ALA will not be attractive as a general-purpose processor without the capability of integer mathematical operations. We could implement on ALA a family of calculations including addition, subtraction, multiplication, division, as well as matrix-matrix (or matrix-vector) multiplication. [24] is dedicated to the explanations of the implementations and performance evaluation. For the purpose of our work, a summary of all available mathematical operations are listed and described briefly.

1. Addition

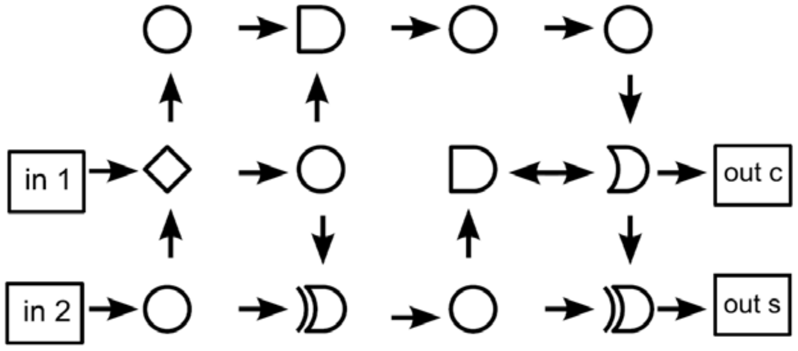


Figure 2-3: A serial adder schematic implemented on ALA, courtesy of [24].

The two-input adder on ALA is best implemented as a serial adder. The two addends are streamed into the adder serially, with the Least Siginificant Bit (LSB) comes in first and Most Significant Bit (MSB) comes last. The core of the serial adder is a one-bit full adder, but the carry out of the full adder is connected back to the adder's carry in. In this way, a multi-bit addition could be performed recursively. The result is also streamed out with its LSB coming out first. Figure 2-3 shows the ALA schematic of the adder.

One might ask why serial adder is more advantageous in ALA. In a conventional computer, addition is best performed by special-purpose hardware such as a Kogge-Stone Tree Adder, which feeds in addends in a parallel manner and uses a Carry Look-Ahead (CLA) strategy [29]. Such kind of CLA adder generates the carry signal in $O(log\ n)$ time and is faster than the ALA serial adder, which takes $O(n)$ time to produce the carry signal. However we choose to avoid a CLA adder in ALA environment because the CLA structure is not suitable for local data communication: a carry signal of lower bits needs to penetrate through the neighboring bits to reach a higher bit, in order to produce a "look-ahead" carry signal. The non-local data exchange will lead to unnecessary long wirings in ALA environment and poor power efficiency. Therefore, the serial adder is a good trade-off for ALA, in which a little bit of speed loss wins back a considerable gain in power and area.

2. Subtraction

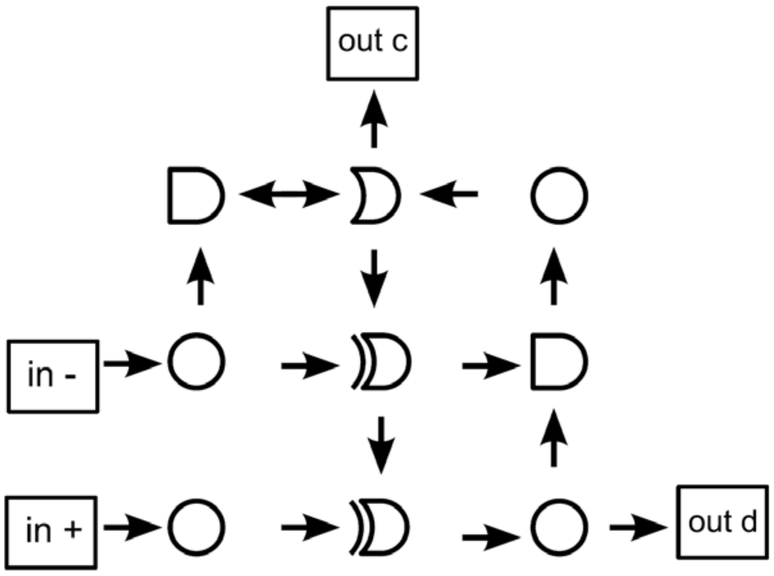


Figure 2-4: A subtracter schematic implemented on ALA, courtesy of [24].

The subtraction operation (schematic shown in Figure 2-4) is analogous to the addition. The carry bits become the borrow bits. The borrow bit being residual indicates a negative result. Ignoring the residual bit gives us two's complement representation. As an example, subtracting $(0001)_2$ from $(0000)_2$ yields $(1111)_2$ and a residual borrow bit of 1.

3. Multiplication

The multiplication on ALA is implemented by mimicking the familiar pen-and-paper approach of multiplying, which has compact and local computing structure by nature. Figure 2-5 shows an exemplary ALA multiplying circuit. This circuit multiplies a 4-bit multiplier (IN 1) with a multiplicand of flexible bit length (IN 2), in which the length is set by the length of the bit buffer at the bottom of the figure.

For ease of explaining, we refer to vertical stripes as *stages* and horizontal ones as *tracks*. The multiplier (IN 1) is first fanned out (the *fan out* stage) and each of the 4 bits is routed to its appropriate track at the *select* stage. The bits are copied (the *duplicate* stage) and bit-wise multiplied with the multiplicand (IN 2) bit string at the *multiply* stage. The resultant bit strings at each track are shifted and aligned by

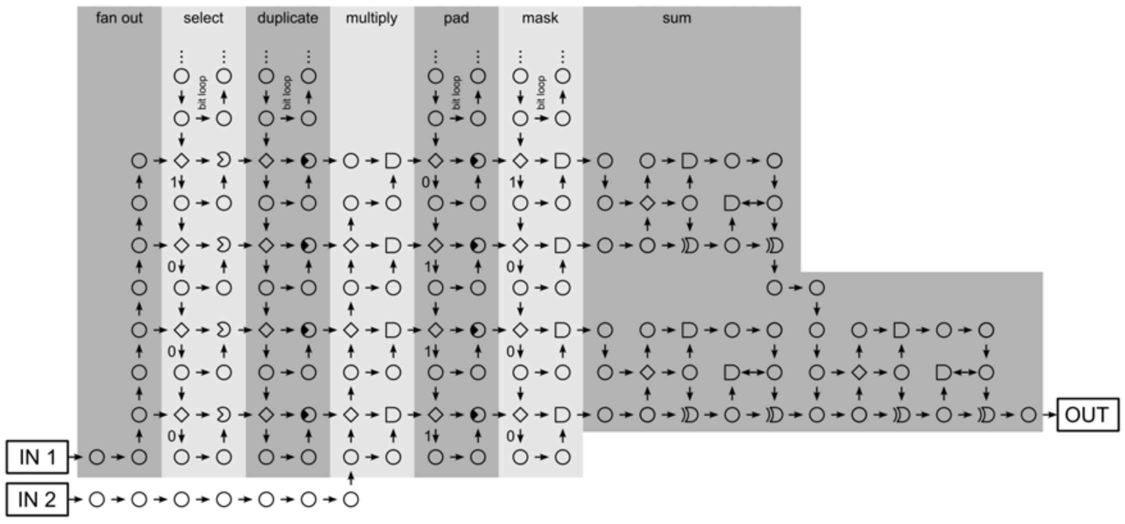


Figure 2-5: A multiplying schematic implemented on ALA, courtesy of [24].

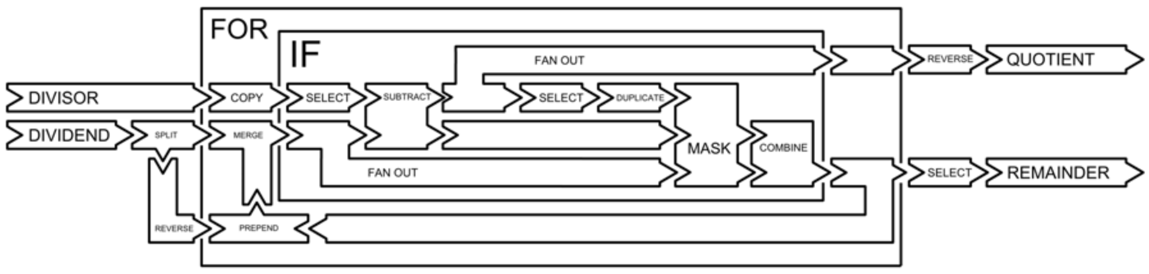


Figure 2-6: A integer divider schematic implemented on ALA, courtesy of [24].

an appropriate amount at the *pad* and *mask* stages. Finally, a tree of serial adders calculate the sum and stream out the multiplication result at the *sum* stage.

4. Integer Division

The integer division algorithm on ALA is more complex than the operations introduced so far, because it requires conditional and loop flow control constructs similar to "FOR" and "IF" statements in C. We will give a global structure of the ALA integer divider here in Figure 2-6, and readers could refer to [24] for the remaining details of each functional blocks.

5. Matrix-matrix Multiplication

The matrix-matrix multiplication is implemented with a matrix of multipliers and

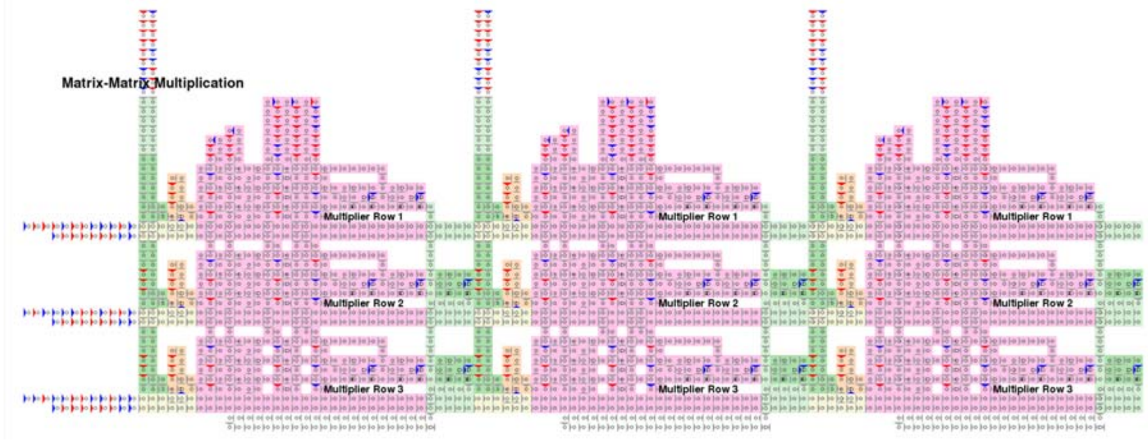


Figure 2-7: An ALA schematic implementing multiplication between two 3×3 matrices, each element of the matrices is a 3-bit number, courtesy of [24].

adders. Figure 2-7 shows an example in which two 3×3 matrices containing 3-bit numbers are multiplied together. The elements of the first matrix is serially fed into the input ports on the left, column by column. And the elements of the second matrix is serially fed into the input ports at the top, row by row. They are multiplied with each other at the ALA multiplier matrix and the result is summed up to produce an output matrix at the bottom part of the schematic. The matrix-vector multiplication is a subset of the matrix-matrix multiplication, which can be implemented by one column of the schematic in Figure 2-7.

This matrix multiplier takes the same hardware space as the matrix being multiplied, and the time complexity is also linear, thanks to the parallel architecture.

2.2.2 Bubble Sort

The well known “Bubble Sort” algorithm could be implemented in ALA at a computational time complexity of $O(n)$ and a computational space complexity of $O(n)$. The bubble sort can be intuitively understood as the unsorted elements gradually “bubbling” up to their sorted location through a series of nearest-neighbor transpositions. The basic operation is to repeatedly check each neighboring pair of elements and swap them if they are out of order [28].

Although the bubble sort is a simple and classical algorithm, it has $O(n^2)$ time

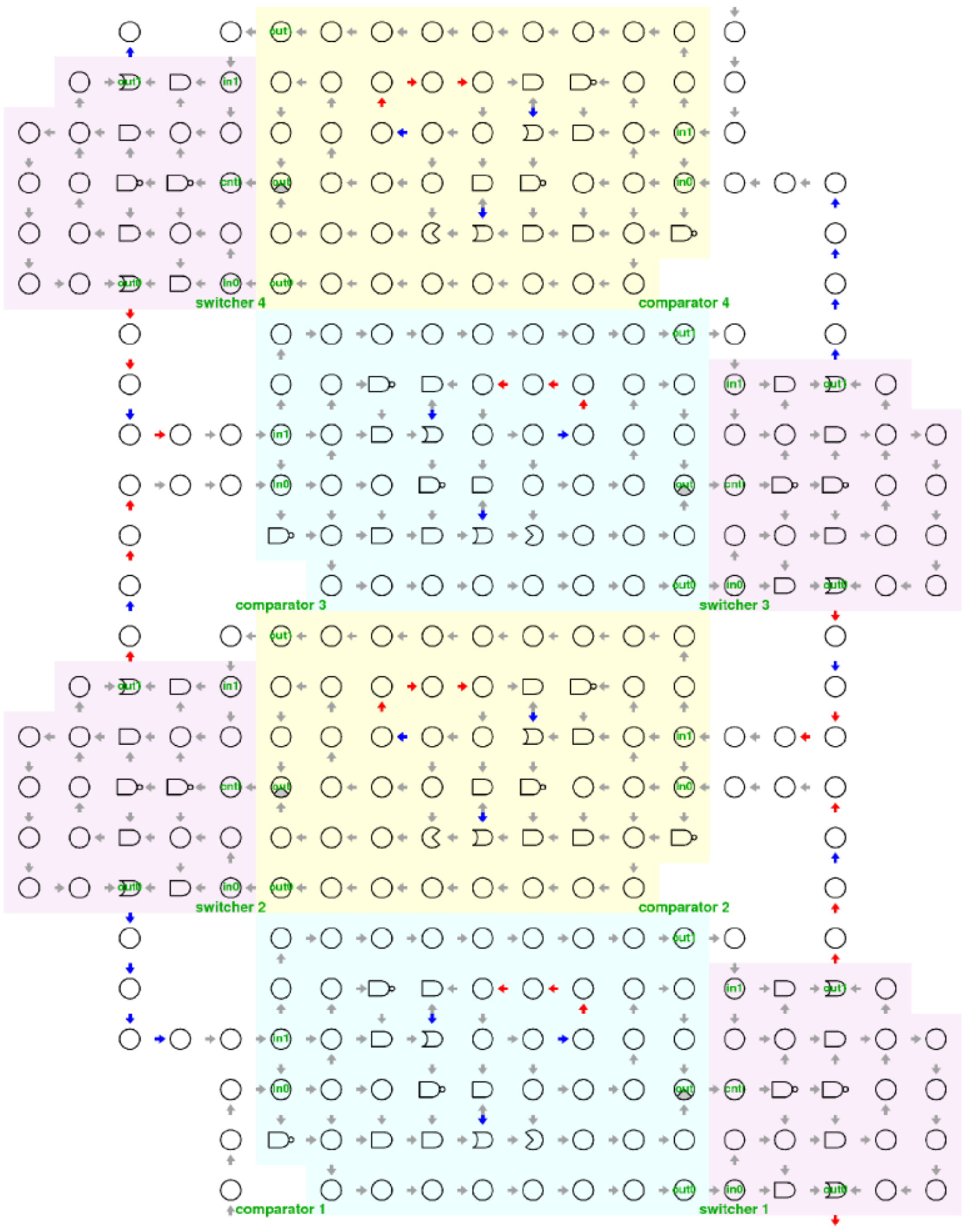


Figure 2-8: Linear-time ALA sorting circuit, courtesy of [8].

complexity if implemented on a sequential computer and is inferior to other faster algorithms such as quicksort ($O(n\ log\ n)$ time complexity). This is not the case on a highly-parallelized ALA computer, where all non-overlapping pairs of elements can be compared simultaneously at no extra cost. The parallelism in turn leads to $O(n)$ time complexity, which is superior than the best possible sequential sorting algorithm. This kind of linear speed-up in a parallel processor is common in special-purpose array processors [33], but our implementation on ALA is different because the ALA model is a general-purpose universal computation.

Now we will take a close look at the ALA bubble sort algorithm [8]. Two function blocks called the "switchyard" and the "comparator" are key to the implementation. The elements to be sorted are represented as bit strings and are serially streamed into the sorting machine. Each comparator operates on two input bit strings of the elements to be sorted and outputs them unmodified, along with a single-bit control line which indicates whether the elements are out of order. A corresponding switchyard receives both the bit strings and the control signal from the comparator. Depending on the control signal, it transposes two bit strings if they are out of order or passes them back out unmodified for the next round of comparisons.

By interleaving switchyards and comparators together, we are able to assemble a bubble sort machine. At any time instance, half the comparators or half the switchyards are active since all the pairs being compared simultaneously are non-overlapping. Figure 2-8 shows a portion of an ALA sorting implementation. It is easy to identify that the circuit size grows linearly with the number of elements to be sorted. The execution time is also linear because in worst-case, an element has to travel from one end to the other, which is linear as the number of elements. Note that here in asynchronous operation, the cost of a computation is not counted in the number of operations, but the distance that information travels.

2.2.3 SEA: Scalable Encryption Algorithm

The Scalable Encryption Algorithm for Small Embedded Applications (SEA) [65] is a scalable encryption algorithm targeted for small embedded applications, where low

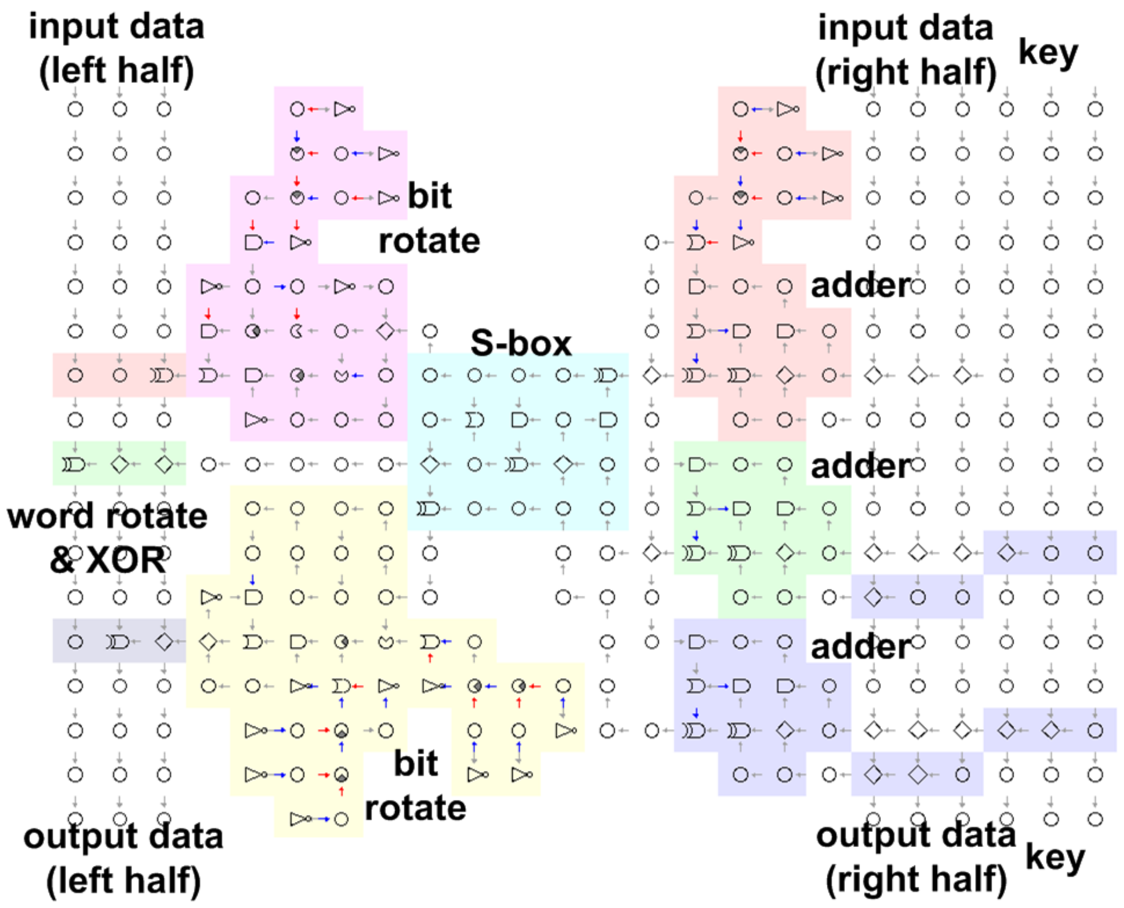


Figure 2-9: ALA SEA encryption circuit, courtesy of [8].

cost performance and maximum parameter flexibility are highly desirable [39, 40]. It is particularly well-suited to ALA because the cipher structures can operate on streams and take advantage of an arbitrary degree of parallelism to deliver a corresponding degree of security.

The SEA implementation is a little more complicated than the bubble sort and the primitive operations include [8]:

1. Bitwise XOR

2. Substitution box (S-box)

3. Word rotate

4. Bit rotate

5. One-bit serial adder

Given the above building blocks, a SEA circuit can be constructed. Figure 2-9 shows one round of a 48-bit SEA encryption process (seven rounds are needed for a practically useful SEA encryption). The rightmost three columns are carrying the encryption key bit streams (48-bit key); the leftmost three column and the three columns left to the key streams are carrying input data streams (every 48-bit block is a data segment). In one round of SEA encryption, half of the data block and half of the key bits are combined with three adders. The combination is processed by the S-box and then bit-rotated. The rotated bit stream is then XOR'ed with the right half of the input data. By placing the XOR gates properly, the word rotate operation is achieved implicitly. At the output, the right half data and key is unchanged and propaged to the next round; and the right half data is changed after the XOR operation. The two halves of the data also need to switch places with each other before the next round of processing, which is not shown in the figure.

2.3 Asynchronous Circuit Design Basics

This section provides a very brief overview of asynchronous circuit design techniques. It is not intended to be a complete tutorial, but the techniques discussed here are mostly relevant to our work in ALA cell library design.

2.3.1 Classes of Asynchronous Circuits

Depending on the delay assumptions made for circuit modeling, asynchronous circuits can be classified as being speed-independent (SI), delay-insensitive (DI), quasi-delay-insensitive (QDI), or self-timed [64, 53, 11]. The distinction between these are best illustrated by referring to Figure 2-10.

A SI circuit can operate correctly under the assumption of positive, bounded but unknown delays in gates and ideal zero-delay wires. Take the circuit segment in Figure 2-10 as an example, this means d_A, d_B, and d_C could be arbitrary, but d_1, d_2,

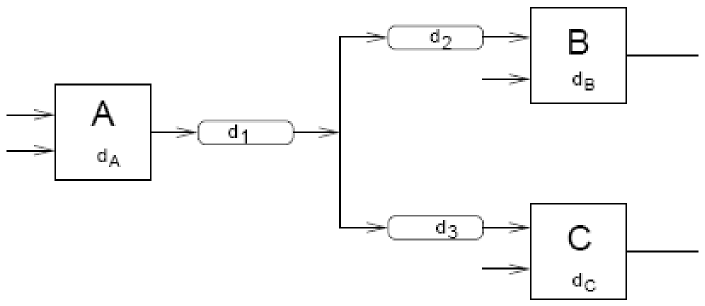


Figure 2-10: A circuit fragment with gate and wire delays for illustration of different asynchronous circuit classes, courtesy of [64].

and d_3 are all zero. Clearly, zero wire delays are not practical, but one way to get an "effectively" SI circuit is to lump wire delays into the gates, by allowing arbitrary d_1 and d_2 and forcing $d_2 = d_3$. This SI model with an additional assumption is actually equivalent to a QDI model, which will be introduced later.

A DI circuit can operate correctly under the assumption of positive, bounded but unknown delays in both gates and wires. This means d_A, d_B, d_C, and d_1, d_2, d_3, are all arbitrary. This is apparently weakest assumption for circuit component behavior, but it is also practically difficult to construct DI circuits. In fact, only circuits composed of C-elements (will be defined in the succeeding section) and inverters can be delay-insensitive [45].

A QDI circuit makes slightly more strict assumption than a DI circuit: the circuit is delay-insensitive, but with the exception of some carefully identified wire forks where $d_2 = d_3$ (Note the equivalence of the QDI definition to the SI model with the addtional assumption). Such wire forks are formally defined by A. J. Martin [45] as isochronic forks, where signal transitions occur at the same time at all end-points. In practice, isochronic forks are usually trivial; or they could also be easily implemented by well-controlled delay lines. As a result, QDI circuits are considered the most balanced asynchronous circuit model, with minimum assumptions to be practically realizable.

Lastly, self-timed circuits are simply referring to circuits that rely on more elaborate timing assumptions, other than those in SI/DI/QDI circuits, to ensure correct

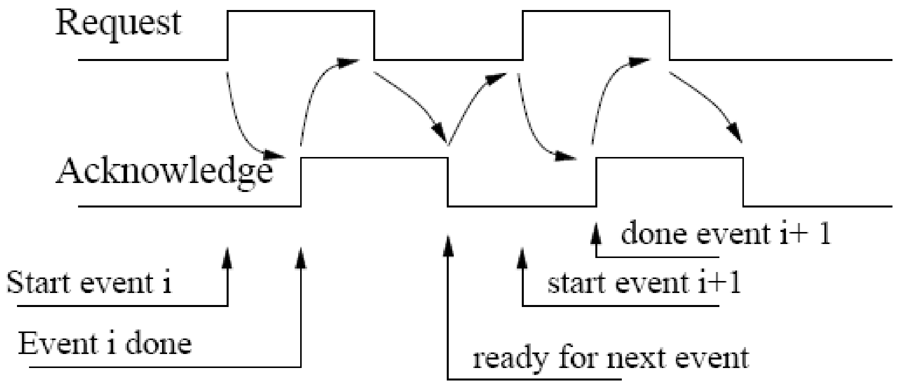


Figure 2-11: The four-phase handshake protocol, courtesy of [11].

operations.

There are many successful projects in the past that used QDI circuits, including TITAC from Tokyo Institute of Technology, MiniMIPS from Caltech, SPA from The University of Manchester and ASPRO-216 from France Telecom. Because of its ease of use, we will focus on QDI asynchronous circuits in following sections and also implement our ALA circuits under QDI assumptions in the next chapter.

2.3.2 The Handshake Protocol and the Data Encoding

The most pervasive signaling protocols for asynchronous systems are the Handshake Protocols. The protocol can be further classified as Four-phase Handshake or Two-phase Handshake.

The Four-phase Handshake is also referred to as return-to-zero (RZ), or level signaling. The typical operation cycles of a Four-phase Handshake Protocol is illustrated in Figure 2-11. In this protocol there are 4 transitions, 2 on the request and 2 on the acknowledge, required to complete a complete event transaction. And by the time one transaction ends, both request and acknowledge signals go back to zero.

The Two-phase Handshake is alternatively called non-return-to-zero (NRZ), or edge signaling. In the operation cycles of a Two-phase Handshake, as illustrated in Figure 2-12, both the falling and rising edges of the request signals indicate a new request. The same is true for transitions on the acknowledge signal.

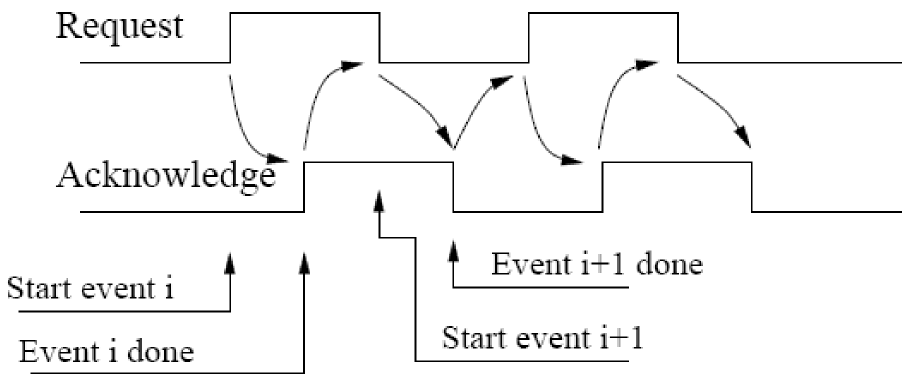
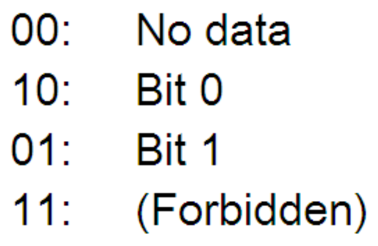


Figure 2-12: The two-phase handshake protocol, courtesy of [11].

Both protocols are heavily used in asychronous circuit design. Usually the four-phase protocol is considered to be simpler than the two-phase protocol because the former responds to the signal levels of the request and the acknowledge signals, leading to simpler logic and smaller circuit implementation; while the latter has to deal with edges. In terms of power and performance, some people think two-phase protocol is better since every transition represents a meaningful event and no transitions or power are consumed in returning to zero. This assertion could be true in theory, but if we take into account that more logic circuits are needed in two-phase protocol, the increased logic complexity, thus increased power, may counteract the power gain from having less transitions. Besides, four-phase proponents argue that the falling (return to zero) transitions are often easily hidden by overlapping them with other actions in the circuit, which means the two-phase protocol is not necessarily faster than the four-phase protocol either. Summing up the above reasoning, we chose to implement the four-phase handshake protocol in our ALA design because of its simplicity and comparable performance and power efficiency to the two-phase protocol.

Following the discussion about data communication protocol, we also need to know ways to encode data. Two methods are widely used for both four-phase and two-phase handshake protocols. One choice is the *bundled data* encoding, in which for an n-bit data value to be transmitted, n bits of data, 1 request bit, and 1 acknowledge bit are needed. Totally n+2 wires will be required.

The alternative choice is the *dual rail* encoding. In this approach, for each trans-



00:	No data
10:	Bit 0
01:	Bit 1
11:	(Forbidden)

Figure 2-13: The dual rail data encoding scheme.

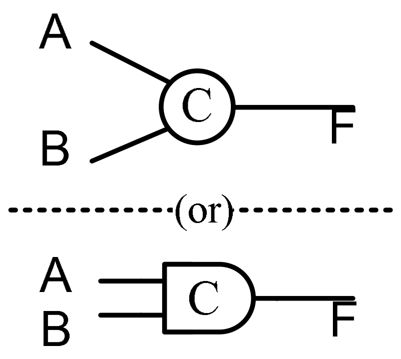
mitted bit, three wires are needed: two of them are used to encode both the data value and the request signal, and the third wire is used as acknowledgement. The data and request encoding scheme is shown in Figure 2-13, in which the request signal could be inferred by judging whether there is data or not on the two wires. In dual rail encoding, an n-bit data chunk will need 2n+1 wires, because only the acknowledge wire could be shared.

Usually the dual rail encoding needs more wires than the bundled data encoding for communication. But because in ALA implementation, we will only transfer 1-bit data in the whole system, the wire cost is 3 wires/bit in both schemes. And due to the fact that the dual rail encoding scheme is more conceptually close to the "token" model, we chose to use the dual rail encoding scheme for our ALA data communication. And in the following development, we will use the word "data" and "token" interchangeably when we are referring to the data communications.

2.3.3 The C-element

A common design requirement in implementing the above two protocols is to conjoin two or more requests to provide a single outgoing request, or conversely to provide a conjunction of acknowledge signals. This is realized by the famous C-element [17, 54], which can be viewed as a state synchronizer in the asynchronous world, or a protocol-preserving conjunctive gate. Figure 2-14 shows the symbol of a C-element as well as its function. This C-element can also be represented in a logic equation as follows.

$$F_{n+1} = A \cdot B + F_n \cdot (A + B) \tag{2.1}$$



A	B	F(n+1)
0	0	0
0	1	F(n)
1	0	F(n)
1	1	1

Figure 2-14: The C-element symbol and function.

The C-element is useful because it acts as a synchronization point which is necessary for protocol preservation. Many consider C-elements to be as fundamental as a NAND gate in asynchronous circuits. However, excess synchronization sometimes will hurt performance when too many C-elements are used in places where simple logic gates might otherwise suffice.

A multiple input C-element is a direct extension to the two-input C-element defined above, in which the C-element output will become "1" ("0") only when all its inputs are "1" ("0"), otherwise the output will hold the current state.

C-element can also have asymmetric inputs. In an asymmetric C-element, a positively asymmetric input will not affect the output's transition from 1 to 0, but this input must be 1 for the output to make the transition from 0 to 1. In other words, it will only affect the positive transition of the output, hence the name. Similarly, a negatively asymmetric input will only affect the negative transition of the output. Figure 2-15 shows some typical asymmetric C-elements and their corresponding logic equations.

2.3.4 The Pre-Charge Half Buffer and the Pre-Charge Full Buffer

The Pre-Charge Half Buffer (PCHB) and the Pre-Charge Full Buffer (PCFB) [80, 13, 14] is an important class of asynchronous circuit that assumes QDI property and

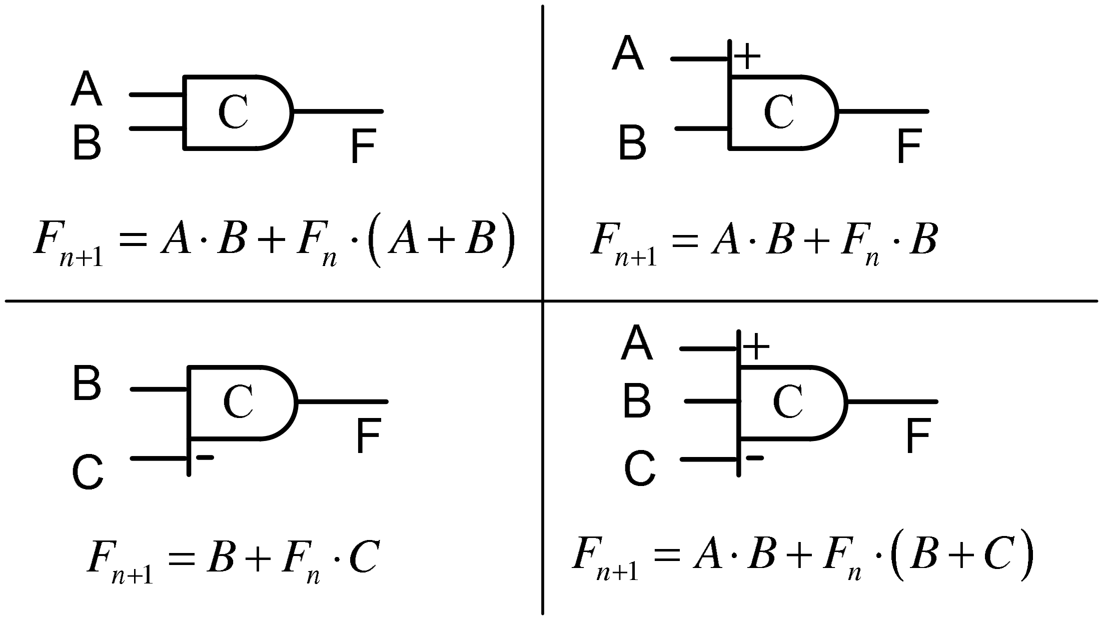

A
B
C
F
$F_{n+1} = A \cdot B + F_n \cdot (A+B)$
A
B
C
F
+
$F_{n+1} = A \cdot B + F_n \cdot B$
B
C
C
F
-
$F_{n+1} = B + F_n \cdot C$
A
B
C
C
F
+
-
$F_{n+1} = A \cdot B + F_n \cdot (B+C)$

Figure 2-15: Variants of C-element, symbols and equations.

communicates with handshake protocols. They are basicly an asynchronous register; by arranging buffers in a chain, an asynchronous pipeline, or shift register could be formed. Because they include the essential behaviors for asynchronous operations, many variants of asynchronous circuits could be built based on them.

Figure 2-16 and 2-17 show the schematics of the PCHB and PCFB, respectively. The protocol for the two circuits is four-phase handshake and is encoded in dual rail scheme. The difference between the PCHB and the PCFB is in their capacitity to hold tokens. In a chain of n PCHB's, only n/2 tokens could be stored in the chain because PCHB's could only hold every other token in their normal operation. But in a chain of n PCFB's, n tokens could be held. We could see PCFB implements a little more logic in each cell to trade for a higher token capacity. In our ALA design, we want to achieve compact data storage where every cell will hold one bit of state, therefore, PCFB is a better candidate for our design. Acutally, our ALA cell design is also based on PCFB structure.

A key design component of the PCFB cell is to use the outputs of the C_3 and

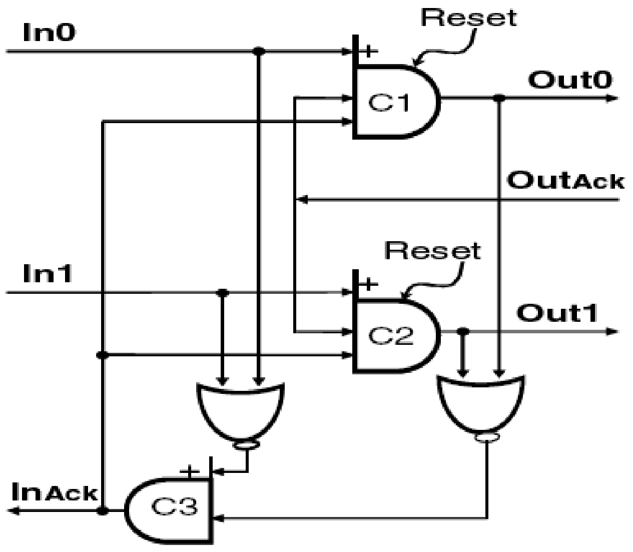


Figure 2-16: The schematic of a PCHB, courtesy of [80].

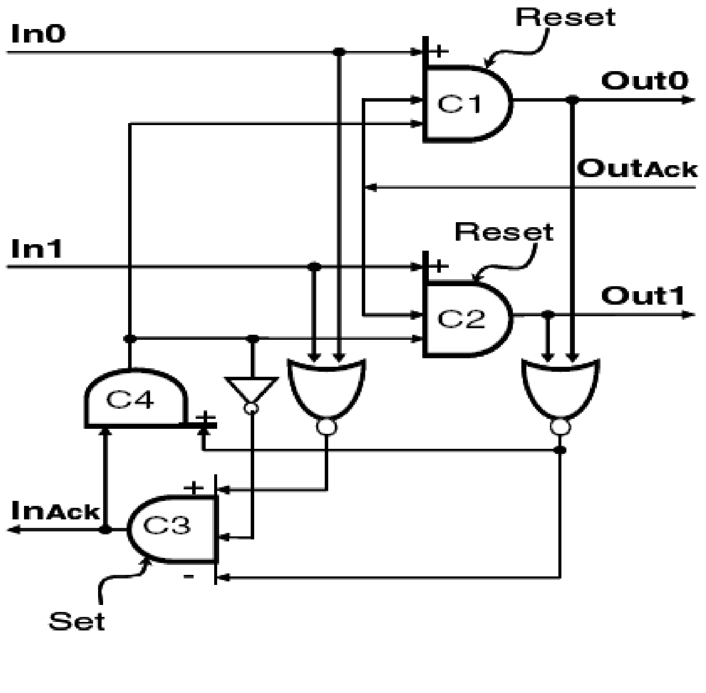


Figure 2-17: The schematic of a PCFB, courtesy of [80].

C_4 C-elements to control the token propagation through C_1 and C_2. In fact, C_3 and C_4 form an asynchronous finite state machine (aFSM) to control the cell operation, while C_1 and C_2 form the computing unit to process and fan-out the token fed into them. Here we will describe the normal Four-phase Handshake operation that is implemented on the PCFB basic cell:

1. We start from the following initial state: there are no tokens on the PCFB cell's input and output; both InAck and OutAck are high, indicating that both the cell and the cell's successor is ready for a new token[1]; and the aFSM's state $\{C_3, C_4\}$ (outputs of C_3 and C_4) is $\{1,1\}$.

2. If a token arrives, either In0 or In1 becomes 1. Because C_4's output and OutAck is high, the token propagates through C_1 and C_2, which makes either Out0 or Out1 becomes 1.

3. The fact that the output is no longer empty is detected by the NOR gate connected to the output. Now the outputs of the inverter and the NOR gate associated with the output become low, which cause the InAck (C_3) to switch from high to low. This is the acknowledgement for the received token.

4. The lowering of InAck causes C_4 switch to low, leading to aFSM's state of $\{0,0\}$. At this point, the cell is held static and waits for either of the two incidents happen: its output token is acknowledged by its successor cell, i.e., OutAck switches from high to low; or its input token is cleared by its predecessor cell as a response to the lowering of InAck.

5. If the OutAck turns low at some point, both C_1 and C_2 are cleared to 0 because OutAck and C_4 are 0. This effectively empties the output token and the NOR gate associated with the output becomes high.

6. If the input token is cleared at some point, the NOR gate associated with input becomes high. InAck switches back to high as a result, indicating that the cell

[1]In our paticular realization, we define acknowledge signal to be negatively active, i.e., ack=1 indicates ready for new tokens, and ack=0 indicates the receipt of a token.

is read to take another new token.

7. Until both of the two incidents take place, C_4 becomes high again, and the aFSM state is back to $\{1,1\}$. This finishes a complete cycle of operation.

Chapter 3

Circuit Design for Asynchronous Logic Automata

In this chapter, we will focus on the circuit design and optimization of the ALA cell library in 90nm CMOS process. The establishment of the ALA design flow is also described.

3.1 Architectural Design

Let us first define the ALA cell library and the design flow that we are trying to build.

> The ALA cell library is a collection of cells with different functions. Each cell waits for input token(s) to arrive and then produces a result token that is stored locally, which in turn can be passed to its rectangular nearest neighbors[1].

- Function Set: the cell library implements logic functions of BUF, INV, NAND, AND, NOR, OR, XOR; and token manipulation functions of CROSSOVER, COPY, and DELETE[2].

[1]We confine the number of inputs for one ALA cell is equal or less than two. This restriction is for simplifying cell designs, but it does not affect the general-purpose ALA architecture because multiple (greater than two) input cells can be implemented by a cascade of two-input cells.

[2]A cell by default is initialized to be storing an empty token (no token), but there are also cells initialized to be storing a "0" or "1" token.

- Asynchronous Communication: ALA cells communicate based on the dual rail encoding scheme and a Four-phase Handshake Protocol.

- Interconnection: an ALA cell has interconnections to and from its four nearest neighbors, i.e., North, West, South and East; the two inputs and up to four outputs of a cell are chosen from the four directions.

- Design Flow: an ALA schematic can be assembled through a "pick-and-place" process. Cells with appropriate functions are first instantiated from the ALA cell library and aligned as a grid. Then the interconnections are placed onto the grid as an additional metal "mask" to finish the design.

The definition is also summarized in Figure 2-1 in Chapter 2. From the definition, we see that the ALA design could be viewed as a grid of highly parallelized and fine-grained pipelined asynchronous computation system. In fact, the cell library could be called as an Asynchronous Bitwise Parallel Cell Library. Because there is no global coordination, any ALA circuit is easily assembled once the ALA cell library and the library of interconnection metal masks are set up. Therefore, the design of ALA circuits comes down to the design of the library and the semi-automatic design flow.

Note that similar works could be found in the literature. The most relevant one is from [42], in which a reconfigurable parallel structure for asynchronous processing is proposed. But the building blocks for that work is not uniform and the design flow is completely manual.

The asynchronous communication interface is implemented with the PCFB structure introduced in the previous chapter. The logic cells' communication interface can be easily derived from the PCFB construction. The token manipulation cells are a little more complicated, and more revision to the asynchronous state machine (aFSM) is needed. Now we will describe the development of the block level ALA cell designs.

3.2 Block Level ALA Cell Design

3.2.1 The Design of the Logic Function Cells

We will start from the simplest ALA cells, the BUF cell and the INV cell. Figure 3-1 shows the block level diagram of the BUF cell design. The control logic (the aFSM) and the computing logic of a BUF cell is the same as the PCFB cell. The BUF cell communicates according to the handshake protocol described in the previous chapter. The aFSM composed by C_3 and C_4 is in charge of the coordination of the protocol. The C_1 and C_2 pair enforces the simple relationship:

$$\begin{cases} Z_1 = A_1 \\ Z_0 = A_0 \end{cases}, \tag{3.1}$$

which implemented the BUF logic function under the dual rail encoding representation.

The difference between our BUF cell and the PCFB cell is that the BUF cell has an additional "Fanout Block" as shown in Figure 3-1. This block is only for a BUF cell that needs to feed its state into two neighboring cells. In that case, the cell can not clear its holding state until both of its succeeding cells send acknowledgements back to it. As a result, we need an extra C-element for the coordination of these two acknowledge signals.

The INV cell is a trivial transform from the BUF cell, where the input wires A0 and A1 are crossed relative to the output wires Z0 and Z1 to enforce

$$\begin{cases} Z_1 = A_0 \\ Z_0 = A_1 \end{cases}. \tag{3.2}$$

An INV cell block diagram is shown in Figure 3-2.

Based on the BUF and INV cell design, we could derive slightly complicated logic function cell designs by incorporating logic into the computing C-element pair C_1 and

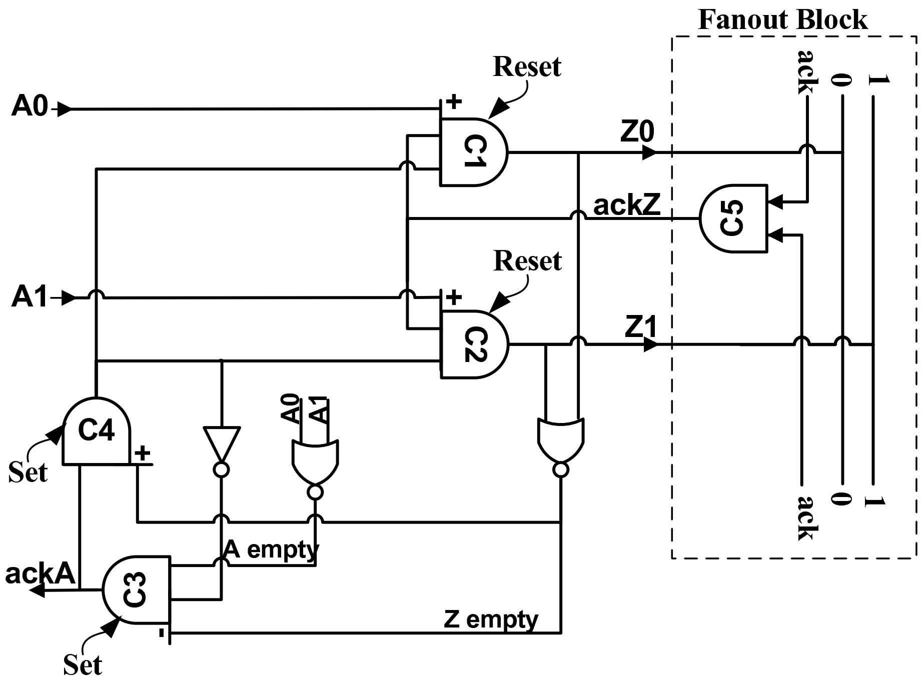


Figure 3-1: Block level design of the BUF cell in the ALA cell library.

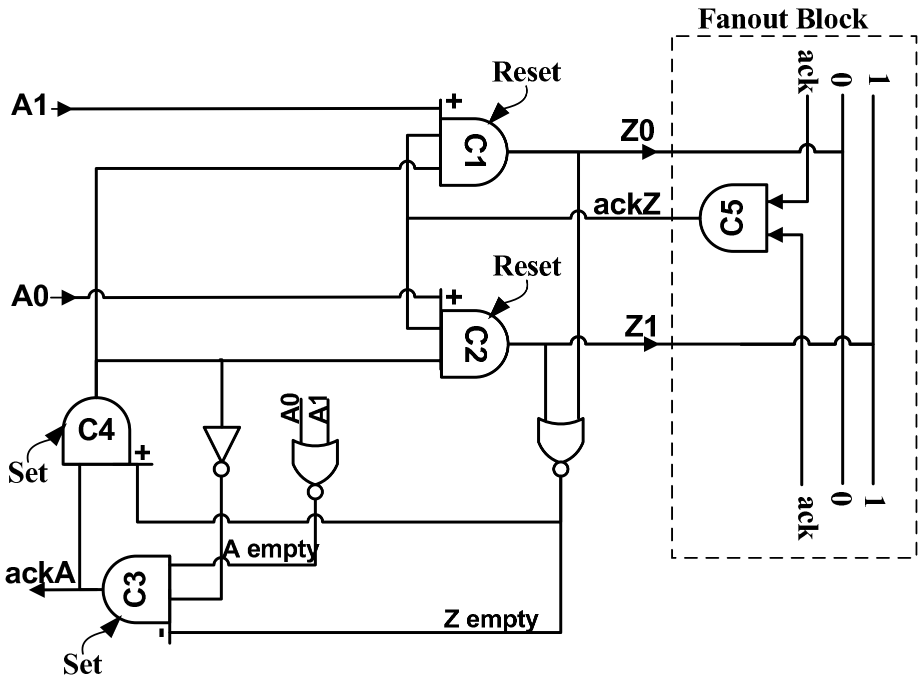


Figure 3-2: Block level design of the INV cell in the ALA cell library.

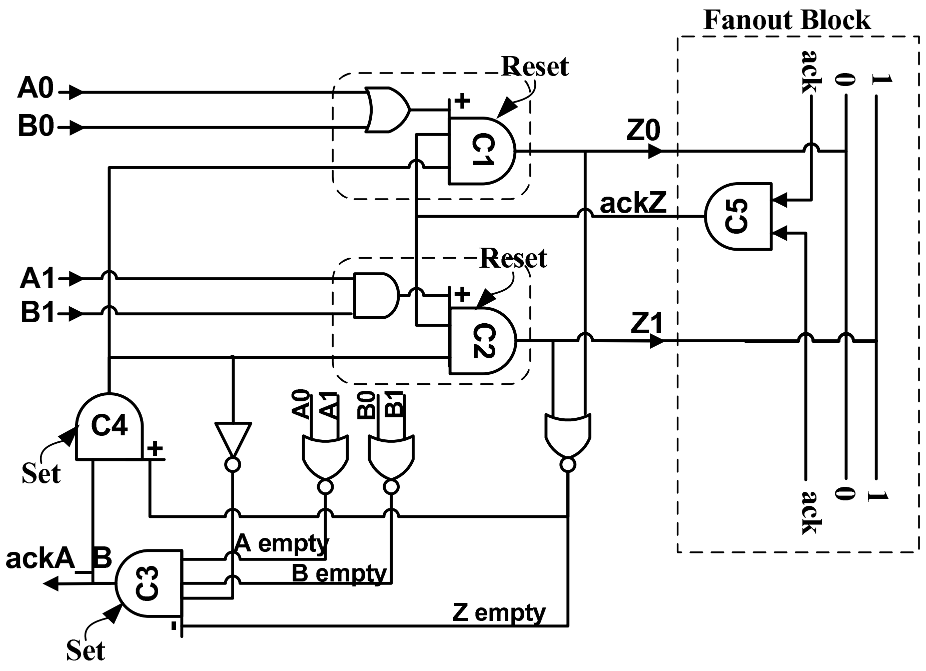


Figure 3-3: Block level design of the AND cell in the ALA cell library.

C_2. An AND cell is shown in Figure 3-3. We could see that this cell enforces

$$\begin{cases} Z_1 = A_1 \cdot B_1 \\ Z_0 = A_0 + B_0 \end{cases} . \tag{3.3}$$

And the added logic could be conceptually represented as the AND and OR gates in front of the C_1 and C_2[3].

Similarly, cells NAND, OR and NOR are constructured by rearranging the logic gates and the inputs combinations. For completeness, the logic expressions and block level design figures are given here.

NAND cell (Figure 3-4):

$$\begin{cases} Z_1 = A_0 + B_0 \\ Z_0 = A_1 \cdot B_1 \end{cases} \tag{3.4}$$

[3]We should point out that the block diagram is only showing the principle. The actual implementation is different, in that there will not be explicit implementations of the logic gates, as will be discussed more in later sections.

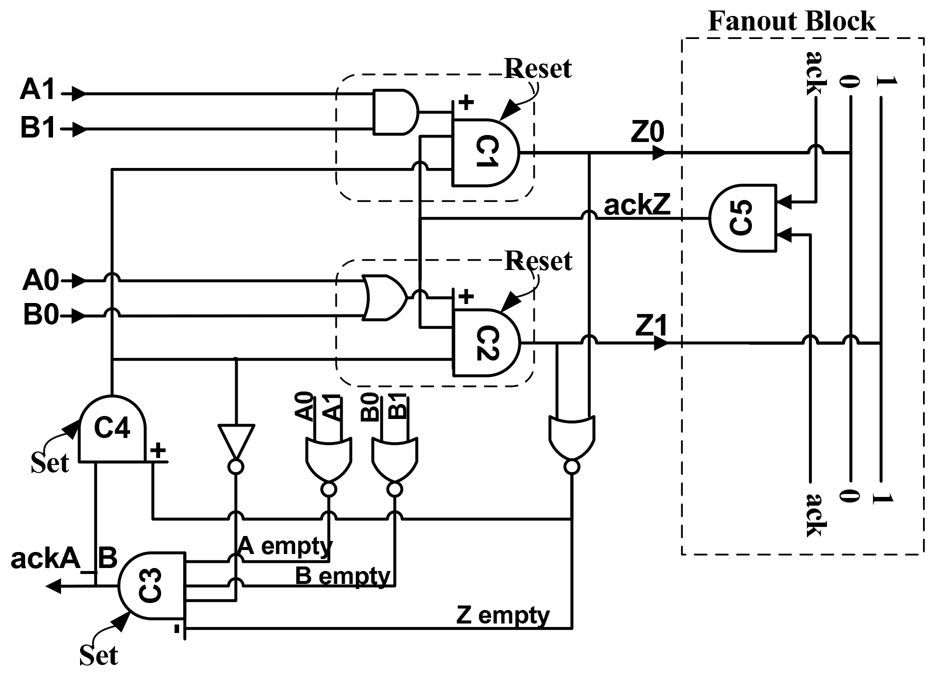


Figure 3-4: Block level design of the NAND cell in the ALA cell library.

OR cell (Figure 3-5):

$$\begin{cases} Z_1 = A_1 + B_1 \\ Z_0 = A_0 \cdot B_0 \end{cases} \tag{3.5}$$

NOR cell (Figure 3-6):

$$\begin{cases} Z_1 = A_0 \cdot B_0 \\ Z_0 = A_1 + B_1 \end{cases} \tag{3.6}$$

The XOR cell is also implemented with more logic gates incorporated in the C-elements, as shown in Figure 3-7. The logic expressions are:

$$\begin{cases} Z_1 = A_1 \cdot B_0 + A_0 \cdot B_1 \\ Z_0 = A_1 \cdot B_1 + A_0 \cdot B_0 \end{cases} . \tag{3.7}$$

Up to now, the cells are all designed to hold empty tokens on initialization. We also implemented cells that are initialized to hold a “1” (also denoted as a “Ture”

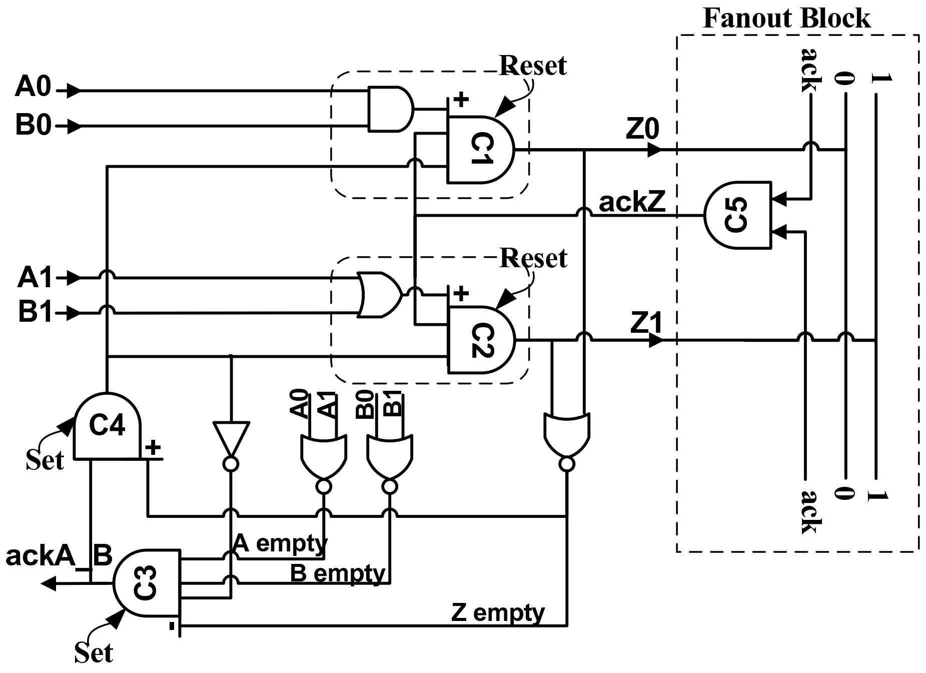


Figure 3-5: Block level design of the OR cell in the ALA cell library.

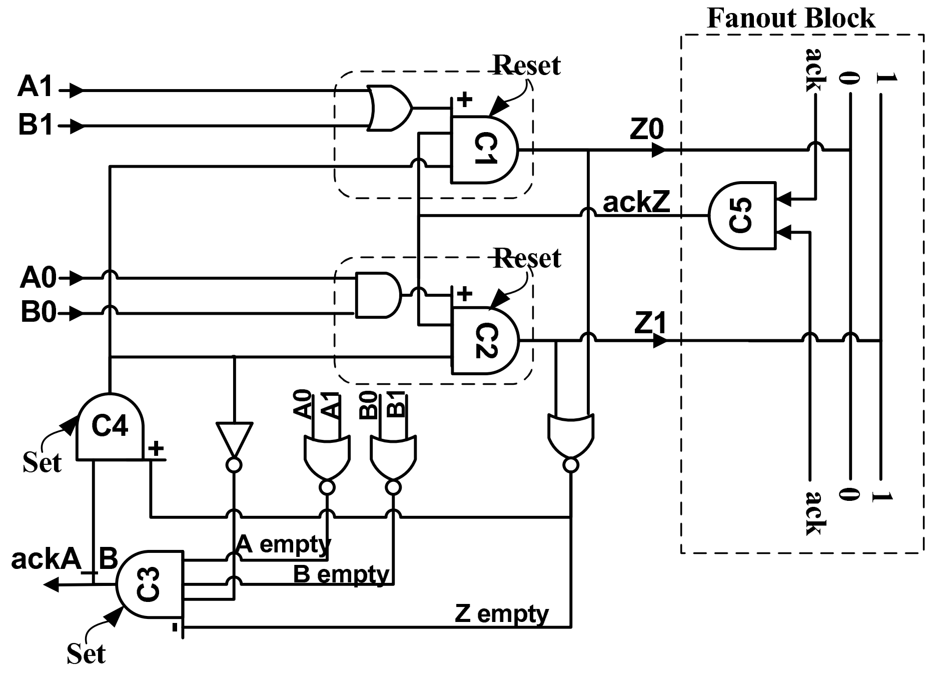


Figure 3-6: Block level design of the NOR cell in the ALA cell library.

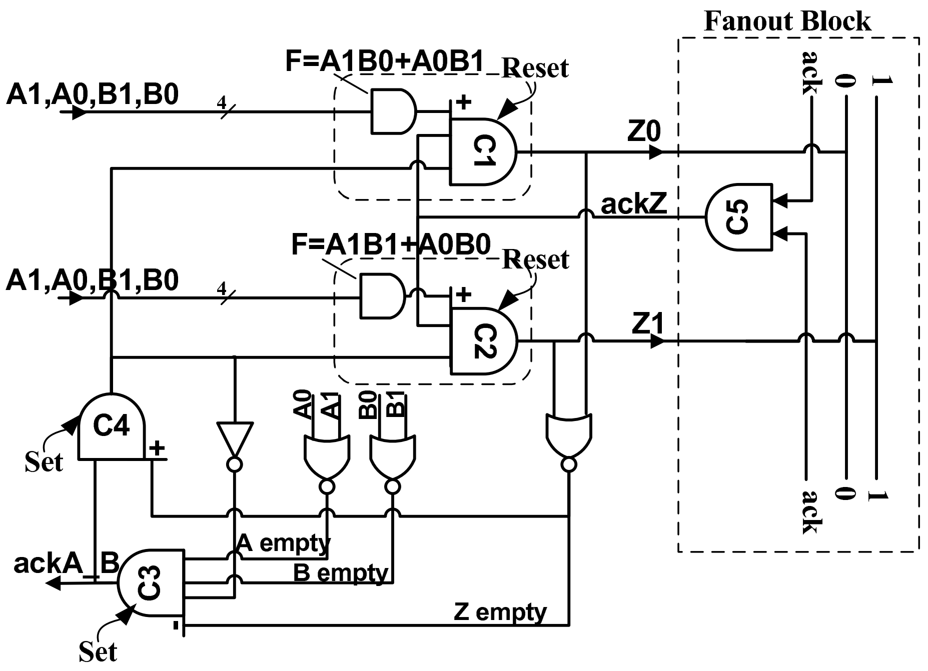


Figure 3-7: Block level design of the XOR cell in the ALA cell library.

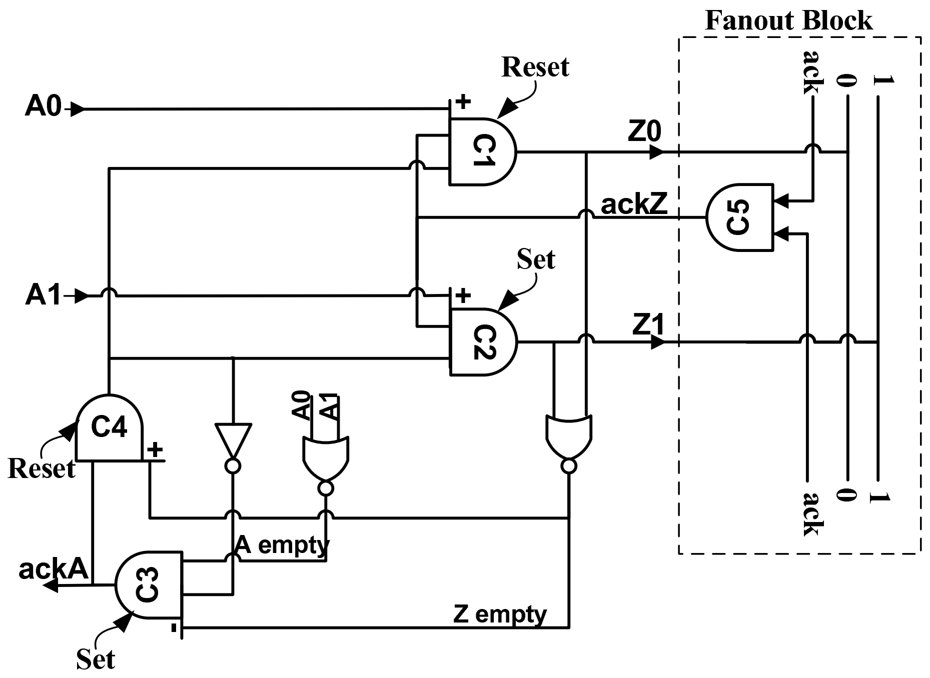


Figure 3-8: Block level design of the BUF cell initialized to "T" (BUF_T) in the ALA cell library.

token or “T”) or “0” (also denoted as a “False” token or “F”) token, in which either C_1 or C_2 is initialized to be high instead of cleared to low. Because this essentially changes the starting state of the cell, the asynchronous state machine state should be initialized differently too, to adapt to the new starting state. We take the BUF cell initialized to “T” (or “1”) as an example to show the consequent block diagram in Figure 3-8.

All logic function cells can be initialized to “T” or “F” in the same way: the C_1 (C_2) element should be initialized to high to represent an initial “F” (“T”) token, and the aFSM ought to be initialized to $\{C_3, C_4\}=\{1, 0\}$ as the starting point of the state machine operation.

3.2.2 The Design of the Token Manipulation Cells

We now describe the design of the CROSSOVER, COPY and DELETE cells, which manipulate token streams.

1. The CROSSOVER Cell

The CROSSOVER cell is dealing with two crossing streams of tokens. It is topologically essential because our ALA cell does not have diagonal interconnections. The cell is very simple to design in that it is actually two BUF cells combined, as shown in Figure 3-9.

Totally there are four topologically different CROSSOVER cells. Following the format of {Input,Input - Output,Output}, they can be represented as {N,W - S,E}; {N,E - S,W}; {S,W - N,E}; and {S,E - N,W}.

2. The DELETE and COPY Cells

DELETE and COPY operations are for the generation and consumption of the tokens. The two inputs to them are no longer symmetric, because there is a control signal and a input signal. The DELETE and COPY cells are defined in Figure 3-10 and 3-11 respectively. The A inputs in both figures are the data inputs and the B inputs are the control signals. When the control signals are “True”, the cells

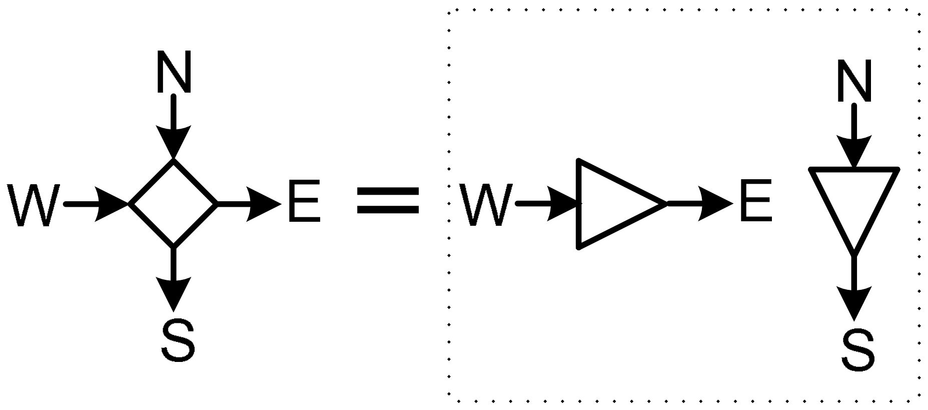
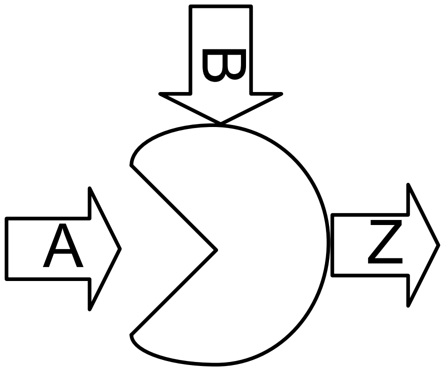


Figure 3-9: Block level design of the CROSSOVER cell in the ALA cell library.

perform DELETE and COPY operations. A DELETE cell consumes the input data (by acknowledging the input token) without propagating to its output (the output remains empty). And a COPY cell replicates the input data to its output without clearing the input data (by not acknowledging the input token). When the control signals are “False”, the cells behave like normal buffers, in which the input data is propagated to the output and the input data is cleared afterwards.

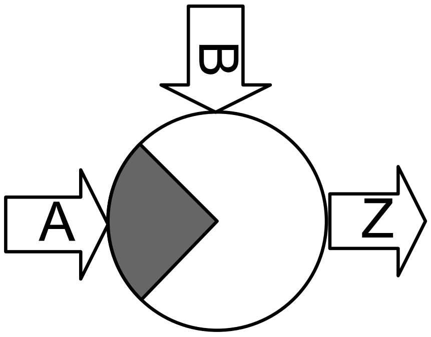
We could realize the desired DELETE and COPY behaviors mainly by modifying the asynchronous state machines as shown in Figure 3-12 and 3-13. For a DELETE cell, the DELETE behavior happens when the input B token is “Ture” (B1=1, B0=0). This behavior requires that the token is not propagated to output Z, which is attained by gating the data input (A1 and A0 wires) with the B0 signal. Secondly, the cell needs to acknowledge the A and B tokens, which is triggered by the $\tilde{\text{B}}$1 signal feeding into the aFSM C_3 element.

The COPY behavior is slightly more complicated. Becasue A and B tokens need to be acknowledged separately, there is an addtional C-element in the aFSM. When the input B token is “True” (B1=1, B0=0), the COPY behavior requires that the token A should not be acknowledged. This is satisfied by the $\tilde{\text{B}}$0 signal feeding into the C_{3a} element. The addtionaly logic at the input of the C_{3b} element keeps the aFSM under correct opertion no matter the value of the input control token B.



A	B	Z	Note
data	0	A	Always ack A and B to clear both
data	1	**empty**	

Figure 3-10: The DELETE cell definition.



A	B	Z	Note
data	0	A	Ack A and B to clear both
data	1	A	**Leave A at input, only clear B**

Figure 3-11: The COPY cell definition.

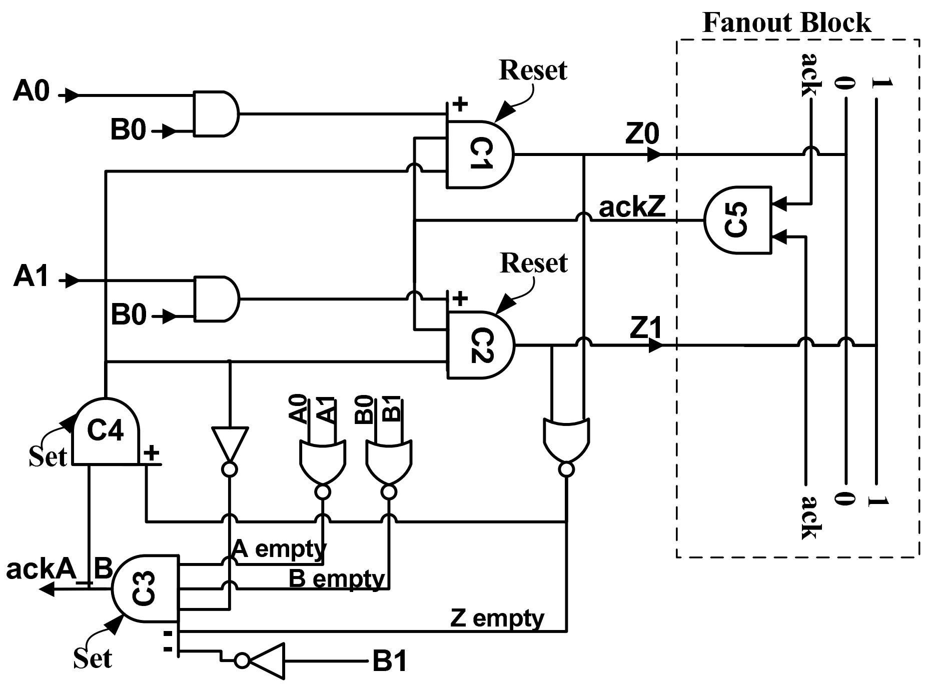


Figure 3-12: Block level design of the DELETE cell in the ALA cell library.

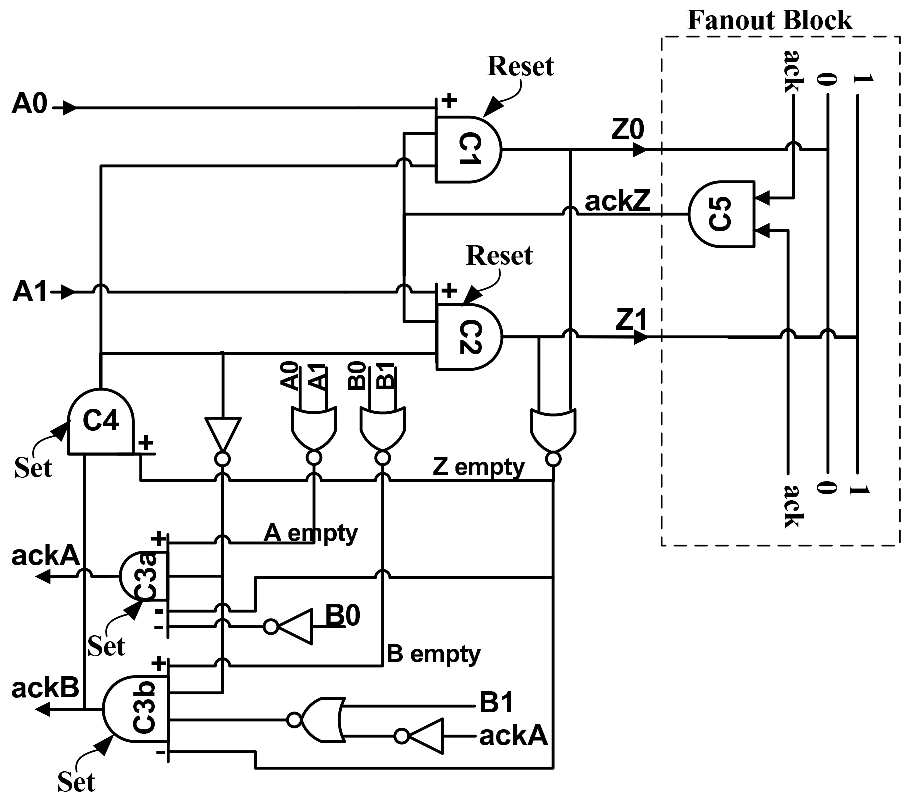


Figure 3-13: Block level design of the COPY cell in the ALA cell library.

3.3 C-element Design

As the block level designs of the ALA cells are primarily composed of different kinds of C-elements, the transistor level design of the ALA cells actually comes down to the design of various C-elements. Not only the state machine control is implemented in the C-elements, but the logic functions the cells are computing are also incorporated inside the C-elements. Therefore we need to find an appropriate way to realize various kinds of C-elements.

3.3.1 Logic Style for C-element Design

There are many different design styles for the Muller C-elements. [59] gives a good summary of the most widely used topologies for the C-element design. Figure 3-14 shows four candidate design styles for C-elements.

In Figure 3-14(a), the C-element is implemented with dynamic logic, in which the node c' is floating when $a \neq b$. Figure 3-14(b) through (d) are static logic style C-elements. But they differ slightly concerning whether they use ratioed transistors to hold and switch states statically. Figure 3-14(b) has been presented in [67] by Sutherland and is termed as the *conventional* static realization of C-element. It is ratioless and transistors N3, N4, N5, P3, P4, P5 form the keeper of the C-element state. Figure 3-14(d) is another static, ratioless C-element proposed by Van Berkel [72]. Transistors N3 and P3 forms the keeper, but the main body transistors are also involved in holding the state of the output. This implementation is called the *symmetric* C-element for its symmetric topology. Figure 3-14(c) is a different kind of static C-element in which the transistor sizes are ratioed in order to correctly hold and switch the output state. The circuit is basically a dynamic C-element added with a *weak* feedback inverter (N4 and P4) to maintain the state of the output. This feedback inverter must be sized weak enough compared to the input stage transistors, so that the C-element could overcome the feedback to change states when necessary. This structure is proposed by Martin in [44].

Among the above four candidates, we choose scheme (c) for a couple of reasons.

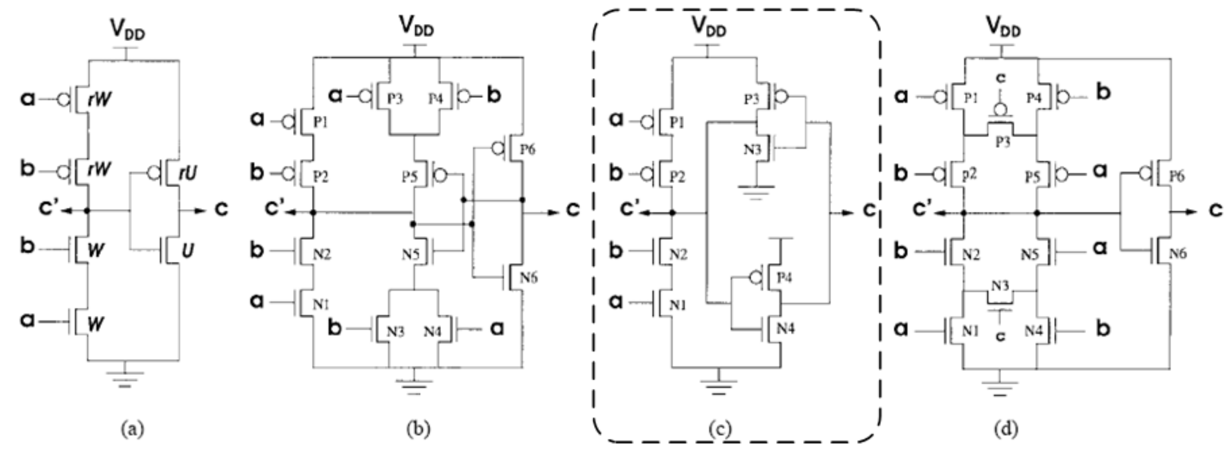


Figure 3-14: Different CMOS implementations of the C-element, courtesy of [59].

First of all, we require a static implementation to ensure reliable state holding, thus the scheme (a) can not be used. Secondly, although static, scheme (b) and scheme (d) are not flexible enough to realize asymmetric C-elements because they are relying on elaborated self locking mechanisms to hold states. They also can not scale well as the number of inputs gets bigger, because for multiple-input C-elements, not only the transistor structure for the keeper logic is difficult to design, but the number of transistors needed would grow exponentially. Lastly, we also want to incorporate logic into the C-elements to achieve compact implementations of the ALA cells. As a result, scheme (c) fits all our requirements. The dynamic logic with a keeper inverter holds state statically; and it provides maximum flexibility to extend a canonical C-element into multiple and/or asymmetric input C-element, as well as to incorporate logic into the latch.

The reason for the ease of extending this logic style is that it decouples the input stage with the keeper. The keeper design is the same for all different input conditions, so we can design the input stage separately to meet design requirements. The input signals to the NMOS transistors are associated with the constraints governing the state transition from 0 to 1; while the signals to the PMOS are associated with the 1-to-0 transition. Every additional symmetric C-element input is obtained by cascoding an extra NMOS and PMOS transistor pair into the input stage transistor chain. And a positively (negatively) asymmetric input is obtained by cascoding a

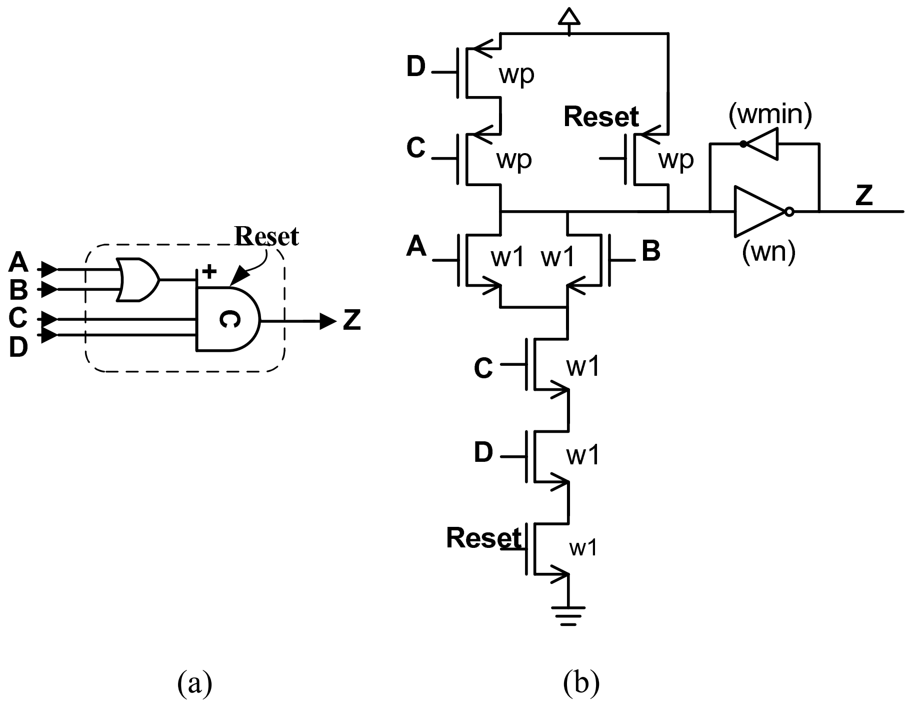


Figure 3-15: An C-element with multiple, asymmetric inputs and OR function: (a)the block level symbol; (b) the CMOS implementation.

NMOS (PMOS) transistor into the chain. Finally, Boolean logic can be incorporated into the input stage in the same way as in dynamic logic circuits [1], i.e., by inserting transistors connected in serial and parallel structures. We will describe the detailed design considerations and optimizations in the next section.

3.3.2 C-element Design and Circuit Optimization

In this section, we take a typical C-element design as an illustration of the C-element design and optimization in 90nm technology. The "OR function C-element" has multiple, asymmetric inputs, and incorporates an OR function inside. This C-element together with the "AND function C-element" is repeatedly used in AND, NAND, OR and NOR ALA cells to compute token values (the dashed boxes in Figure 3-3, 3-4, 3-5, 3-6). Other C-element circuits can be designed and optimized in the same way.

The block level symbol and the CMOS implementation of the "OR function C-element" is shown in Figure 3-15(a) and 3-15(b), respectively.

The NMOS transistors with inputs of A and B are implementing the OR function; and they are also positively asymmetric because A and B inputs are only associated with the NMOS, which govern the 0-to-1 transistion of the output state. The C and D inputs are symmetric C-element inputs and appear at the inputs of both NMOS and PMOS transistors. Additionally, the pair of Reset MOS transistors are used for initialization of the output state to 0. This C-element statically holds the state with a feedback inverter (the inverter labelled with wmin). But the inverter has to be sized weak enough to allow normal state transition. The following paragraphs discuss the optimization of the transistor sizing and other design problems.

(a) Transistor sizing issues in C-element design:

In this ratioed circuit, extra care in sizing is needed for correct functionality of the circuit. Besides, transistor sizes directly affect overall circuit speed/power tradeoff. For example, as the drive transistor get stronger, i.e., w1 increases, the fighting transient would be shorter, decreasing short-circuit dissipations, but at the same time the increase in gate capacitance would increase switching power. Therefore, there must be an optimal point for sizing the drive transistor and output buffer (wn). In addition, we want to save chip area in our cell design. Therefore area of the cell design is added as an component into the overall optimization metrics, which leads to the "$Energy \times Delay \times \sqrt{Area}$" metrics used as the target function to minimize in our design.

We use minimum size NMOS transistor and 2x size PMOS for the feedback inverter to make sure that it is weak enough. We also fix the PMOS size to be twice of the NMOS width in the input stage for two reasons. Firstly this sizing combination guarantees balanced rising and falling transition of the C-element; Secondly this sizing ratio is not very sensitive to the overall performance metrics as is supported by our sweep simulations. Therefore, we fix $wmin = 1$ and $wp = 2 \times w1$, and sweep sizing parameters "wn" and "w1" in the particular sweep simulation that leads to the final sizing choice (Figure 3-16). We can see from the sweeping curves, the performance number reaches optimum on both dimensions at the node indicated by

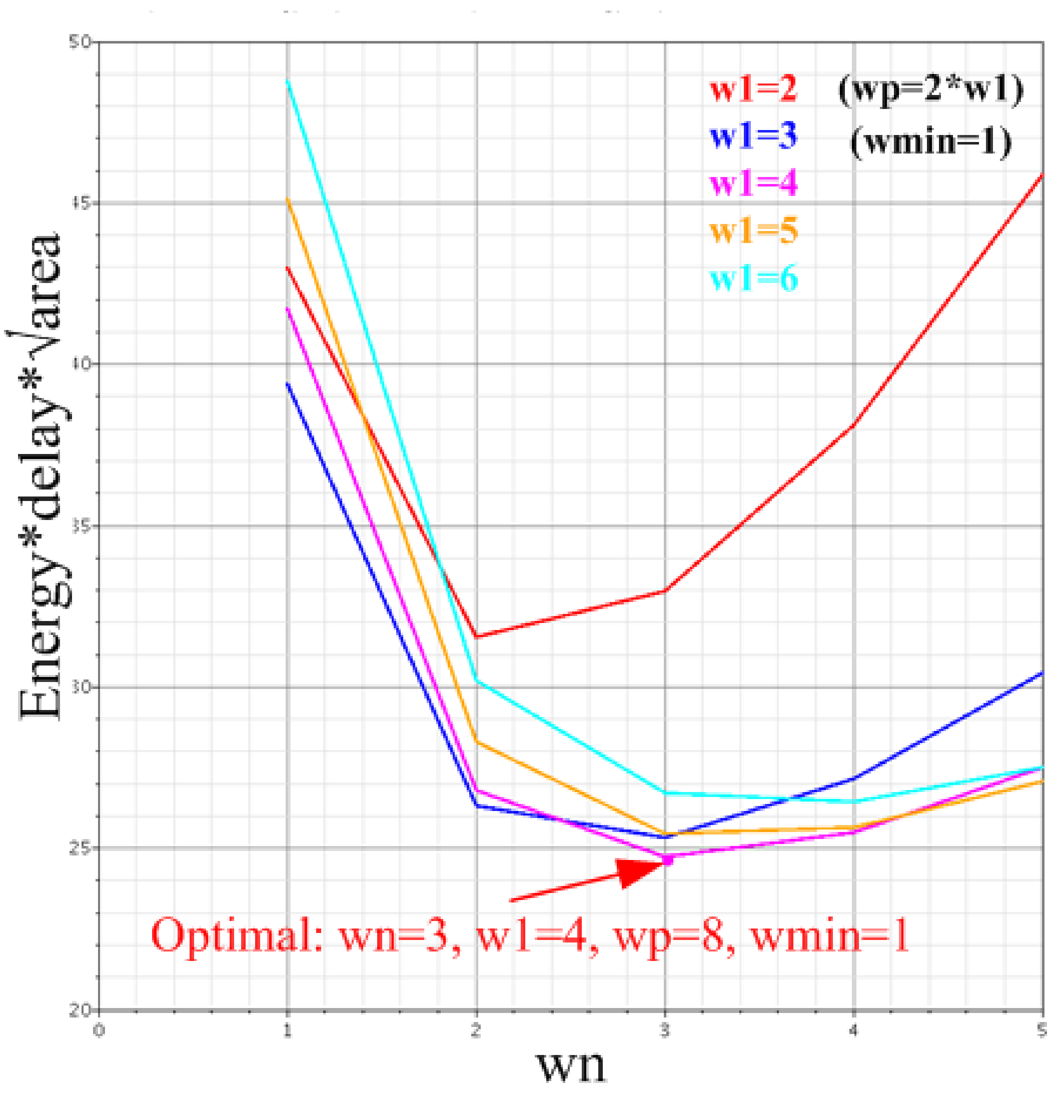


Figure 3-16: Circuit sweep simulation showing the transistor sizing optimum.

the arrow. This optimum is in accordance with our intuition from the circuit. And the final sizing for this C-element design is: $wn = 3,\ w1 = 4,\ wp = 8,\ wmin = 1$. Other C-element designs are optimized in similar ways and the sizing parameters vary slightly.

(b) Charge sharing effects:

Dynamic gates inevitably suffer from charge sharing problems for large fan-in gates. Tapering drain area technique [49] is tried in the design, but the gain is not much. Besides, it adds overhead to layout area due to DRC rules. As a result, uniform sizing is used for a NMOS or PMOS chain finally. Because at most 6 stacked transistors are used in our design, charge sharing is still tolerable in our simulation (about 20% at worst case).

(c) Another design effort for minimizing area:

For some asymmetric C element gates, multiple inputs for stacked PMOS transistor would cause the sizing of PMOS grow to very large numbers ($\sim$10 times of minimum width). We tried to mitigate the stack effect by replacing large MOS chain with only one large MOS gate driven by an equivalent, smaller CMOS logic gate to achieve the same functionality. However, this effort does not obtain satisfactory result because for a large fan-in CMOS gate, the delay increases considerably. Thus small gain in area has to be traded with substantial loss in speed.

Finally, we designed and optimized every C-element that is going to be used in ALA cells. Subsequently, those optimized C-elements are verified to be working at worst cases by running corner simulations to account for transistor mismatches. ALA cells are then assembled and simulated based on the C-element designs. After the layouts of ALA cells are completed, we ran post-layout simulations to make sure that every ALA cell is functioning correctly.

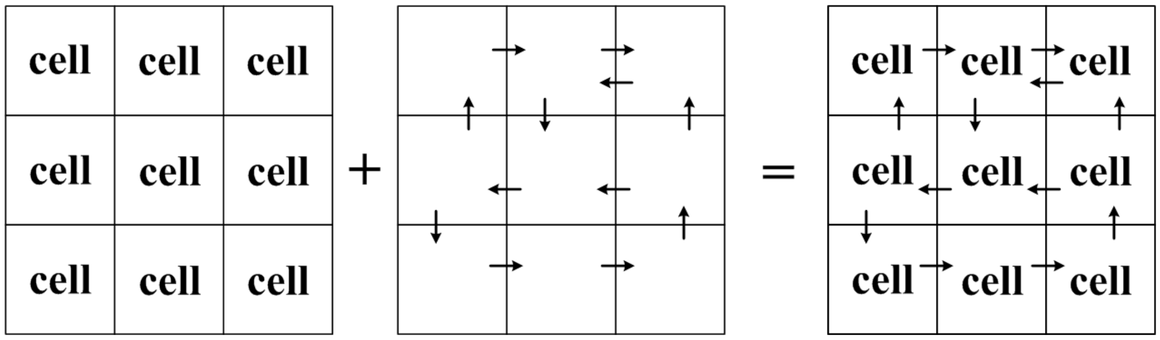


Figure 3-17: The pick-and-place ALA design flow.

3.4 The "Pick-and-Place" Design Flow

Now that we have the ALA cell designs in 90nm process, we continue to establish the "pick-and-place" design work flow at the chip layout phase, in which after placing proper ALA cell layouts onto a grid and aligning them, they automatically connect with each other electrically and form a meaningful ALA application circuit. To establish this work flow, each cell is of the same dimension in layout, and we design the metal interconnections layout which aligns cells naturally using three metal layers. This interconnection layout is then placed as a mask layer onto the ALA cell layout to complete the cell library design. This design flow is also visulized in Figure 3-17.

With this uniform cell library and interconnection mask layer, we could directly map the graphical representation of an ALA schematic into chip layout. This one-to-one mapping essentially merges the Hardware Description Language (HDL) and the resulting hardware implementation of a traditional digital design flow. The pictures of ALA schematics are the HDL description of a digital system in our context; and the mapping from ALA pictures to ALA layouts replaces the whole complicated process of the HDL synthesis, which includes firstly compling HDL codes to the Register Transfer Level (RTL) design, secondly synthesizing RTL design into a transistor-level netlist, and then feeding the netlist into a place-and-route tool to produce a final chip layout.

Finally, the ALA cell library with the interconnection layer is completed in 90nm process and tested under post-layout simulations. Each ALA cell occupies an area

Cell Name	Throughput (GHz)	Energy per Token (pJ)
BUF/INV	2.60	0.144
AND/NAND/OR/NOR	2.35	0.163
XOR	2.28	0.168
CROSSOVER	2.60	0.288
COPY	1.37	0.231
DELETE	2.21	0.170

Table 3.1: Speed and energy consumption summary of the ALA cell library.

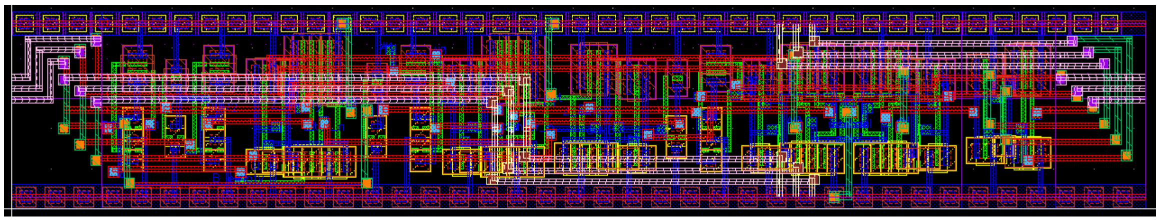


Figure 3-18: The layout of a COPY cell with inputs from "W" and "S"; and outputs to "E" and "N".

of $122\mu m^2$; and Table 3.1 summarizes the speed and power performance numbers obtained from post-layout simulations. Note that because the CROSSOVER cell is two BUF cells combined, its throughput is the same as a BUF cell and the energy consumption for each token's computation doubles the number of a BUF cell. Figure 3-18 shows a layout picture of a COPY cell with inputs from "W" and "S"; and outputs to "E" and "N".

3.5 More Discussion: A Self-timed Asynchronous Interface

So far we have discussed about the circuit design for our ALA cell library, in which the asynchronous communication protocol implementation is the key to the design. In fact, the asynchronous interface logic accounts for a significant part of the overall hardware cost of the cells, since the handshake protocol requires non-trivial amount of logic computation. Although the handshake protocol enforces quasi-delay insensitive asynchronous information exchange, with very weak assumptions on the communication channel, it might be too costly to be implemented in each cell. Meanwhile, we could trade the robustness of the interface for simpler hardware design and lower cost in terms of the transistor count, chip area, energy consumption, etc.

According to the discussion in Chapter 2, self-timed asynchronous interfaces are exactly the class of designs that could make the asynchronous communication protocol simpler. With reasonable assumptions on relative timing of the circuit, self-timed circuits could enforce asynchronous operation with lower hardware complexity. In comparison with QDI interfaces, such as the one we have already implemented in the previous sections, self-timed interfaces are generally less robust in theory. But in practice, if we design the interface carefully to ensure that signals propagate in the right sequence, it could still offer correct operations for all practical situations.

In this section, we make an experiment to test a self-timed asynchronous interface design.

3.5.1 The Self-timed Asynchronous Interface Design: The Basic Circuitry

In our self-timed implementation of ALA cell communication, the data/token representation is still dual rail, which is defined in Chapter 2 (Figure 2-13). Conceptually, we divide the design into blocks for ALA cell state storage and blocks for cell communication interface. For example, Figure 3-19 shows a segment of an ALA cell chain. In this figure, The two square blocks labelled with "Cell 1" and "Cell 2" are state storage blocks. The rectangular block at the center is the asynchronous interface, which not only regulates data exchanges between Cell 1 and Cell 2 (in this case data is propagated from Cell 1 to Cell 2), but also incorporate logic inside (in this case the logic incorporated is "INV").

The cell state storage block either stores a "0" or "1" token, or remains at the empty state. The cell state is reflected at the "Z0" and "Z1" ports. The "IsEmpty" port also indicates whether the cell state is empty or not. To control the cell storage, the "A0" and "A1" ports are used for assigning a "0" or "1" token to the cell; while the "clear" port is used for clearing the cell state, i.e., setting the cell state to be

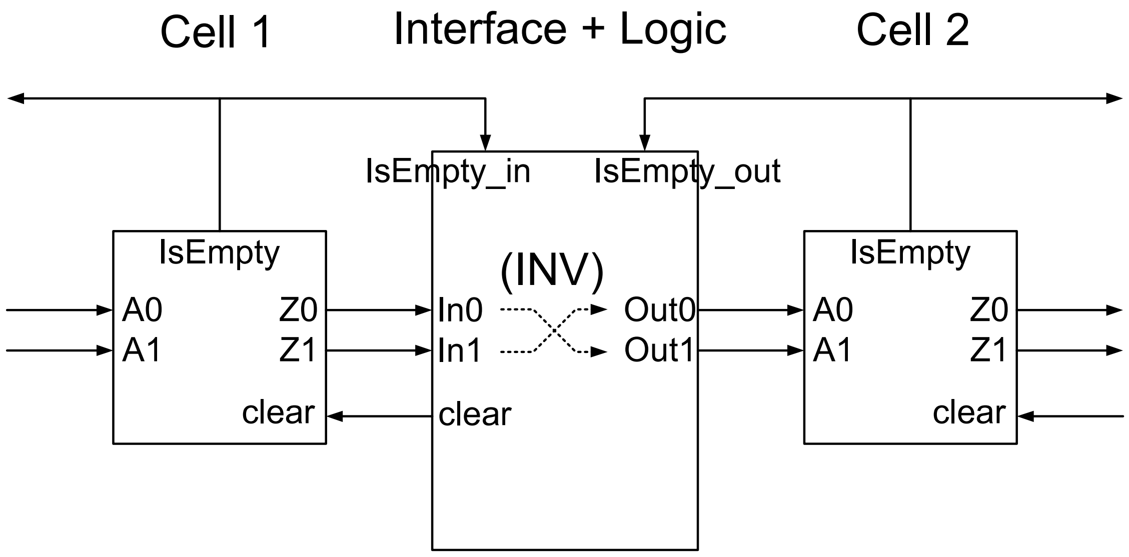


Figure 3-19: A block diagram showing self-timed asynchronous cells and the interface: tokens are inverted by the asynchronous interface and propagated from Cell 1 to Cell 2 in this example.

empty.

The asynchronous interface between Cell 1 and Cell 2 enforces self-timed asynchronous data transfer from Cell 1 to Cell 2 with embedded "INV" function. The interface block waits for the input data to be ready (IsEmpty_in = 0) and the output port to be empty (IsEmpty_out = 1). When this condition is met, the interface trigers a "fire" signal, which in turn controls the circuitry to compute the output data based on the Boolean function and the input data. The output data is pushed onto the receiving cell (Cell 2) and sets its state. The interface circuitry also sends a "clear" signal to the sending cell (Cell 1), which forces the cell to set its state to empty. When the input cell state becomes empty and the output cell state becomes non-empty, the interface circuitry lowers the fire signal and returns to its original state, completing a full cycle of operation.

Figure 3-20 is the transistor-level realization of the cell state storage block. The two-bit cell state is statically held by two SRAM (Static Random Access Memory) units. Each SRAM unit stores one bit in a pair of cross-coupled inverters. To set the value of the bit, two NMOS transistors are connected at both sides of the inverters'

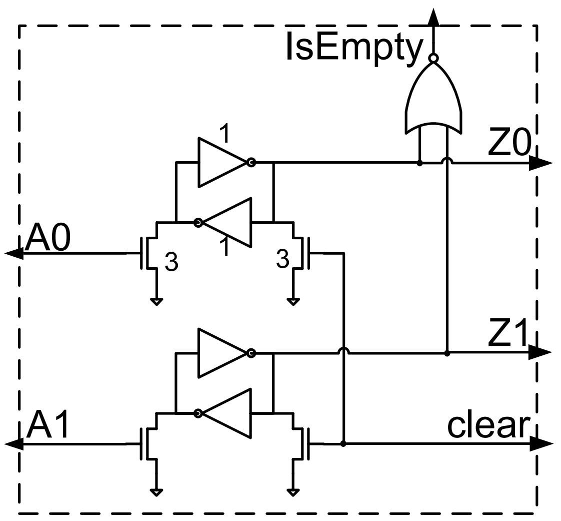


Figure 3-20: The implementation of the cell state storage for self-timed ALA.

outputs. They must be sized stronger than the PMOS's inside the inverters such that they could overcome the inverter outputs and switch. We chose minimum size inverters and the NMOS width is 3x in our implementation. Signals $A0$ and $A1$ are used to set the SRAM units to "1"; signal *clear* is used to set the SRAM units to "0". Furthermore, whether the cell state is empty or not is judged by a NOR gate and the result is output as the signal $IsEmpty$.

Figure 3-21 shows the implementation for the asynchronous interface circuitry with an INV funciton embedded. The C-element takes signals $\overline{IsEmpty\_in}$ and $IsEmpty\_out$ as inputs and produce the $Fire$ signal as an indicator for the condition of "input ready, output empty". When Fire is high, it enables the input token to propagate through the two AND gates. The crossing of input signal wires is effectively doing an INV Boolean computation. The computed token is then present at the output port and sets the state of the receiving cell. The receiving cell holds a new data and its status becomes non-empty, which flips the voltage on the $IsEmpty\_out$ wire. Meanwhile, a delayed version of the $Fire$ signal produces a *clear* signal at the NOR gate output. When *clear* becomes high, it sets the input cell's state to empty. The input cell thus becomes empty and the voltage on the $IsEmpty\_in$ wire is flipped. After both the $IsEmpty\_in$ and $IsEmpty\_out$ flip their value, the $Fire$ signal returns

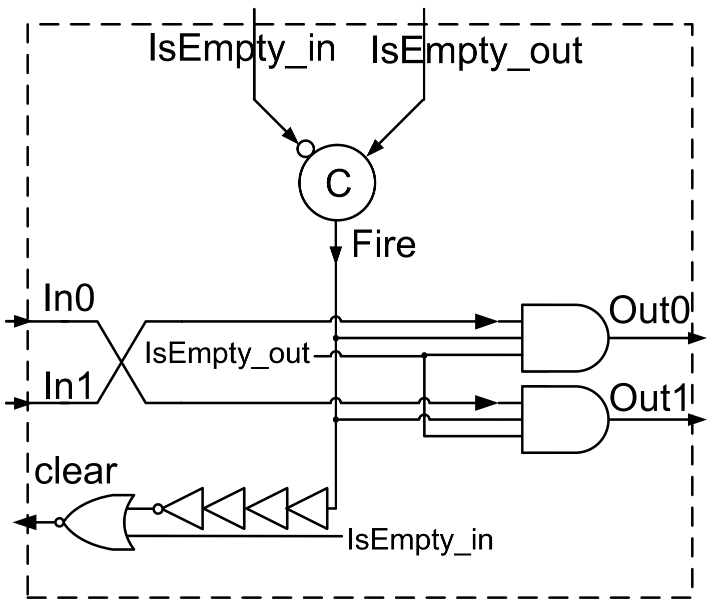


Figure 3-21: The implementation of the cell interface with an INV function embedded.

to low, which completes a full cycle.

There are several design issues in this circuit to ensure desired operation. Firstly, the *clear* signal must be generated (turn high) late enough, so that the input cell could still hold its valid state long enough before it is cleared, for it to propagate to the receiving cell. Therefore we added some buffers with proper gate delay in front of the NOR gate input. Secondly, the *clear* signal must also return back to low early enough to avoid race conditions. Because if the *clear* signal remains high for too long a period, the empty input cell might be able to receive another new token from its predecessor. This could cause both NMOS transistors of a SRAM unit to be on at the same time, shorting the circuit. To prevent this race condition, we feed the $IsEmpty\_in$ signal back to the input of the NOR gate that generates the *clear* signal. In this way, the *clear* signal will come back to low as soon as the cell becomes empty. Thirdly, we need to prevent similar race conditions at the output port: $Out0$ and $Out1$ should come back to low quick enough to stop a short-circuit condition. This is because as soon as the receiving cell becomes a non-empty cell, it might triger another firing and propagates its newly received token to the its succeeding cell, in which it will receive a “clear” command from its succeeding interface circuitry. If either $Out0$

or *Out*1 were still high by the time the receiving cell got a "clear" command, the race condition would take place. As a result, we take similar approach to solve this problem: the *IsEmpty_out* signal is used to gate the two AND gates. When the receiving cell becomes non-empty, the *IsEmpty_out* wire turns low, which forces the outputs of the AND gates to be low.

3.5.2 The Self-timed Asynchronous Interface Design: Multiple Inputs, Multiple Outputs and Other Logic Functions

The last section shows the design for a one-input one-output INV cell design. In this section we scale the design up to deal with multiple inputs, multiple outputs and other logic functions.

To begin with, the cell state storage design is kept unchanged. We only need to make revision on the interface circuitry between cell storage blocks. Figure 3-22 shows a revised asynchronous cell interface block circuitry that could perform logic functions with multiple inputs and fanout the result to multiple output cells.

We could see four major changes in Figure 3-22 as compared to Figure 3-21. The first change is that we augment the number of inputs of the C-element to catch the new firing condition, in which we require both input cells' states are non-empty and both output cells' state are empty.

The second change is that we add a general "logic" block to produce the output tokens depending on the input tokens. For symmetric functions (all Boolean logic operations in ALA definition), the output tokens $OutA'$ and $OutB'$ are the same. And the logic block implementations can be easily derived. For example, an AND function could be expressed as:

$$\begin{cases} OutA_1 = OutB_1 = InA_1 \cdot InB_1 \\ OutA_0 = OutB_0 = InA_0 + InB_0 \end{cases} . \tag{3.8}$$

This is of the same form as in equation (3.3). Similarly, functions NAND, OR, NOR

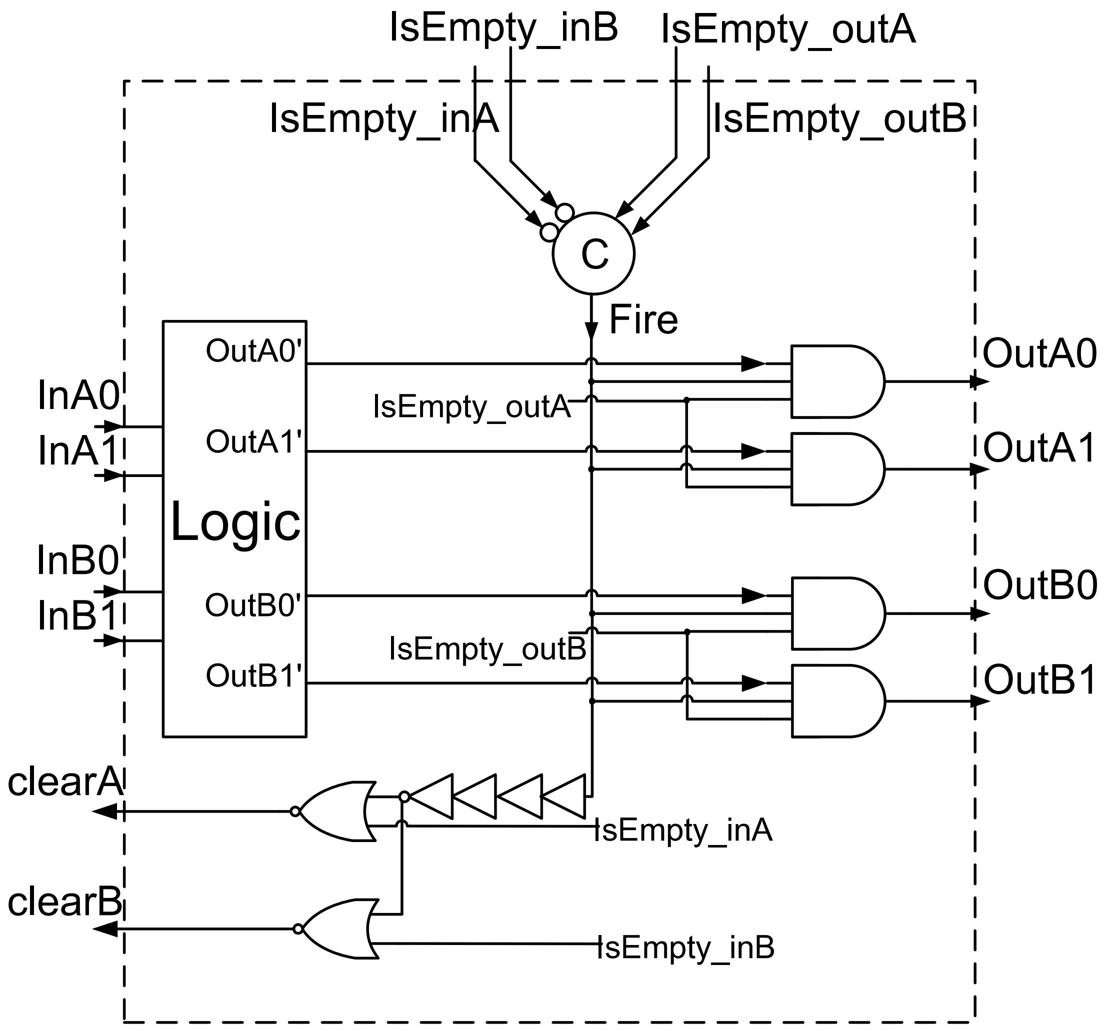


Figure 3-22: The implementation of the cell interface for general functions, two inputs and two outputs.

and XOR can be expressed in the same way as in equations (3.4) through (3.7). Additionally, the CROSSOVER cell function can be obtained by crossing input token wires and output token wires: $OutA = InB$; $OutB = InA$. The COPY and DELETE cell function can not obtained by only changing the logic block. But we could get the desired behaviors by adding proper logic into $Fire$ and $clear$ signal generations, which is similar to the designs discussed in the previous sections.

The Third change is that we have two separate output ports. Each output port is composed of a pair of AND gates gated by the $Fire$ and the $IsEmpty\_outX$ signals (X is either A or B).

The last change is the duplication of the $clear$ signals to separately control the resetting of each input cell state. Each $clear$ signal is also gated by the $IsEmpty\_inX$ signals to prevent race conditions (X is either A or B).

This general asynchronous interface can be connected to two input cells and two output cells to coordinate data exchanges. Changes can be easily made to adapt the general design into specific situations or less inputs/outputs.

3.5.3 Performance Evaluation and Summary

To test the throughput and energy consumption of this design, we run simulation for a chain of INV cells. Because at current design phase, we have not done the layout of the circuit, we did not run post-layout simulation for evaluations here. The simulated throughput of the INV cell is 4.53 GHz, and the energy consumption for each computation is 0.080 $pJ/data$. If we account for 20% performance degradation for the layout, the expected post-layout self-timed INV cell throughput is 3.62 GHz; energy consumption is 0.096 $pJ/data$.

We could compare these numbers with the performance of the handshake ALA INV cell given in Table 3.1. The comparison is summarized in Table 3.2. We could see over 30% improvement in both speed and energy efficiency in the self-timed ALA INV cell. Moreover, even without actually laying out the circuit, we could safely predict large area efficiency improvement in the new design due to the significantly simpler hardware architecture.

Performance Comparison	Throughput	Energy per Token
Handshake INV	2.60 (GHz)	0.144 (pJ)
Self-timed INV (expected)	3.62 (GHz)	0.096 (pJ)
Comparison	+39%	-33%

Table 3.2: A performance comparison between the Handshake ALA INV cell and the Self-timed ALA INV cell.

To sum up, we could see that the self-timed asynchronous circuits have the potential to reduce hardware overhead in ALA design. Although the asynchronous interfaces assume a few timing dependencies, these assumptions are relatively easy to be attained in the design. Therefore, the self-timed asynchronous circuitry could be of interest for a newer version ALA design.

Chapter 4

Applications of Asynchronous Logic Automata

In this chapter, we demonstrate the ALA cell library and the design flow. To evaluate the cell performance and the throughput of this "fine-grained" bitwise asynchronous pipelining structure, we first test traditional Finite Impulse Response (FIR) Digital Filters implemented by ALA. This test also demonstrate that the ALA library is compatible with traditional digital circuits. After that we test ALA based on a few applications described in Chapter 2.

4.1 The FIR Filter Implemented by ALA

Because ALA cells are universal in implementing digital circuits, we start by showing that the ALA library is compatible to design conventional digital circuits. Although a majority of the ALA applications introduced previously enjoy a streaming data representation and a serial computation structure, ALA cells can still be assembled to realize computation in a parallel data representation (data bus) as in many conventional digital circuits. The locality advantage is not evident in some of the traditional circuit structures, but high throughput is observed owing to the bitwise pipelining architecture. Here we take the FIR digital filter as an example. Since the FIR filter is composed of a couple of adders and multipliers, we describe the design of each block

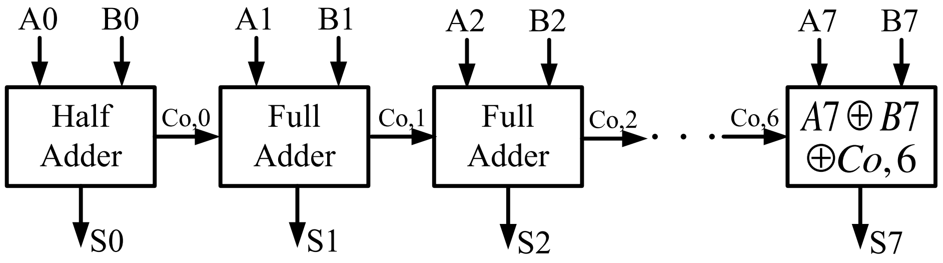


Figure 4-1: The block diagram of the 8-bit ripple adder.

before the test of a complete FIR filter.

4.1.1 The 8-bit Ripple Adder

A ripple adder is the simplest implementation for a multi-bit adder. Several 1-bit full adders are cascaded together to perform bit addition in parallel over bits. But higher bit additions would still need the carry bits to propagate through from lower bit results. In our case, an 8-bit ripple adder is a cascade of one half adder for the LSB addition, six full adders and a XOR unit for the MSB addition. We save the LSB and MSB full adder into simpler logic because of the fact that the carry-in bit is assumed to be zero and the carry-out bit is discarded. Figure 4-1 shows the diagram of the 8-bit adder.

The logic for a half adder and the diagram are in (4.1) and Figure 4-2, respectively.

$$\begin{aligned} S &= A \oplus B \\ Co &= A \cdot B \end{aligned} \tag{4.1}$$

The full adder can be assembled based on two half adders and an OR cell. The logic equations and the diagram are respectively shown in (4.2) and Figure 4-3.

$$\begin{aligned} S &= A \oplus B \oplus Ci \\ Co &= A \cdot B + (A \oplus B) \cdot Ci \end{aligned} \tag{4.2}$$

An 8-bit adder is implemented with the ALA cell library, using 34 cells. The total chip area for this adder is 2728 μm^2. The post-layout simulation shows that the adder

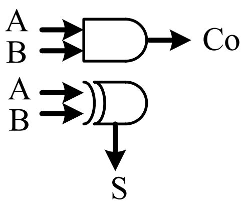


Figure 4-2: The logic diagram of a half adder.

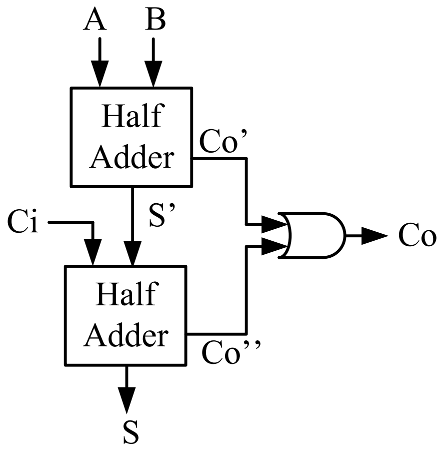


Figure 4-3: The logic diagram of a full adder.

has an average throughput of 1.1 GHz and the computation consumes an average of 6.08 $pJ/data$.

4.1.2 The 4-bit Signed Multiplier

A 4-bit two's complement signed multiplier is implemented in a network of half adders, full adders and AND/NAND cells. Figure 4-4 shows a block diagram of the multiplier.

The 4-bit multiplier is implemented in a total area of 5790 μm^2. The post-layout simulation shows that the multiplier has an average throughput of 653 MHz and the computation consumes an average of 11.68 $pJ/data$.

4.1.3 The 4-bit FIR Filter

After testing all building blocks, we can now implement a FIR filter with 4-bit data bus width. A 4-bit 3-tap FIR filter (Figure 4-5) is first implemented and evaluated.

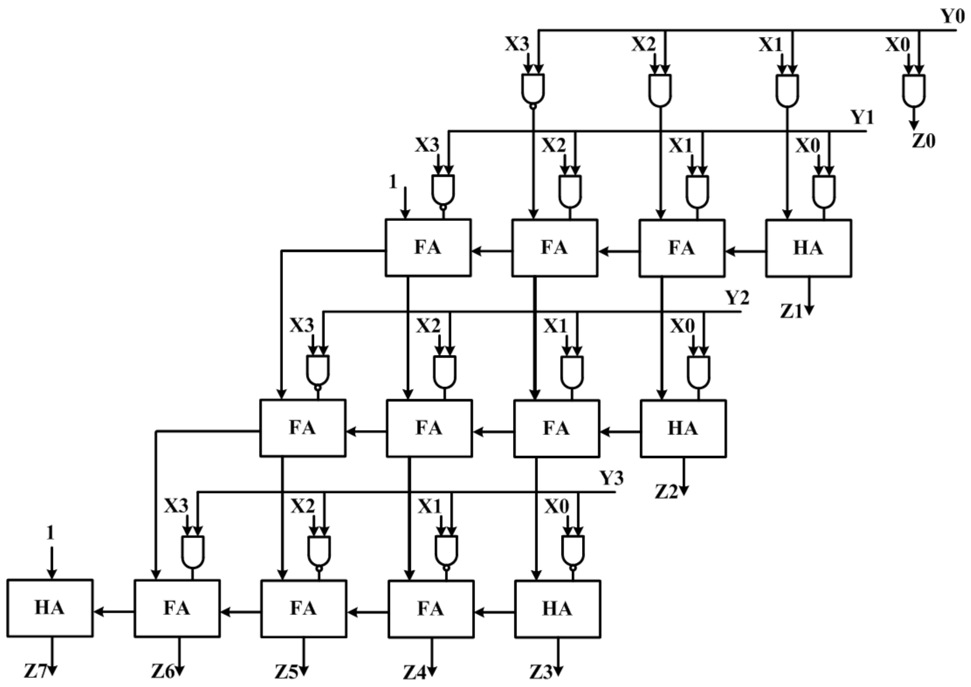


Figure 4-4: The block diagram of a 4-bit signed multiplier.

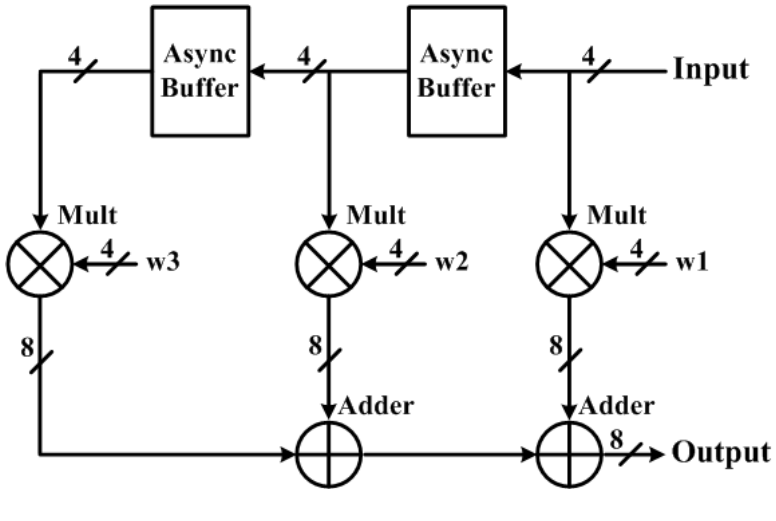


Figure 4-5: The block diagram of a 4-bit 3-tap FIR filter.

As can be seen from Figure 4-5, the Asynchronous Buffer is implemented by 4 BUF cells in parallel, which effectively delay the input data by one step (in the sense of asynchronous updates). The input data and their delayed version are multiplier by three fixed coefficients respectively. The resulting 8-bit products are added together to yield an 8-bit output data.

The layout of the FIR circuit is obtained from the diagram. Note that we did not enforce nearst neighbor connections in this particular layout for efficiency reasons. Although we could implement a FIR filter with nearst neighbor communication, it is not efficient because a lot of cells will be used for long-distance signal wiring due to the non-local structure of the conventional FIR filter. As a result, for the sole purpose of testing cell performance and the asynchronous bitwise pipelining performance, we used global metal wiring in the FIR implementation. But the unique advantage of locality in ALA will become evident in the following sections.

At nominal operating condition (Vdd = 1.2 V), the FIR throughput is simulated to be 402 MHz and the energy consumption is 56.0 $pJ/data$. We also carry out circuit performance scaling measurement by varying power supply voltage to see scaling trends. Figure 4-6 shows that the throughput increases sub-linearly as Vdd increases; and the energy consumption goes slightly more than linearly as Vdd increases in Figure 4-7.

The power supply voltage can be scaled down to as low as 0.6 V in the post-layout simulation for correct operation. But by scaling the voltage further down beyond the limit, we discovered an interesting "sleep mode" circuit operation due to the asynchronous communication. Specifically, if the power supply voltage is lowered down to below 0.6 V, the ALA cell can not switch states any more, but all the cells still retain their digital states as long as the power supply is on. When the supply voltage is turned back above 0.6 V, the ALA circuit could resume operation correctly. This robust dynamic power scaling behavior is due to the robustness of the asynchronous interface. Because when the supply voltage is not high enough to trigger a firing behavior in an ALA cell, the input tokens of that cell are still kept unchanged, by the handshaking protocol. Figure 4-8 shows one example, in which the

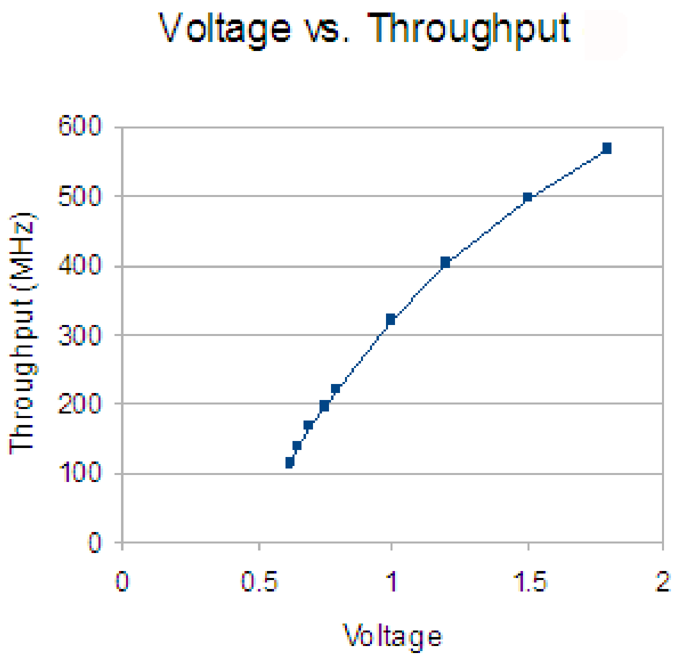


Figure 4-6: Power supply voltage scaling effect on the throughput of the FIR filter.

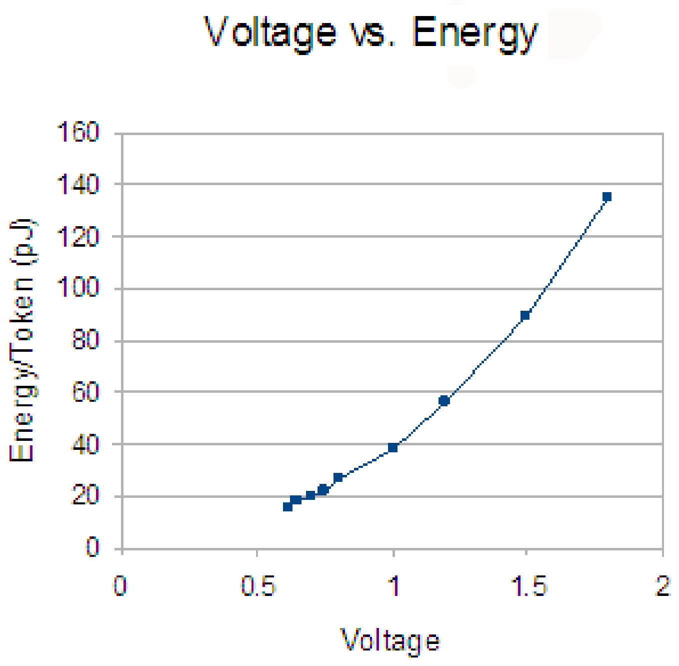


Figure 4-7: Power supply voltage scaling effect on the energy consumption of the FIR filter.

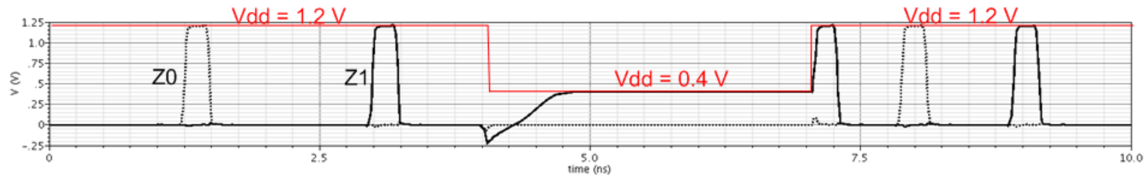


Figure 4-8: Dynamic supply voltage scaling and "sleep mode" for the ALA cell.

power supply voltage drops from 1.2 V to 0.4 V to stop the ALA cells from updating, but when the voltage is restored to 1.2 V, the cell state is recovered and correct operations resume. We could exploit this interesting asynchronous circuit feature to realize dynamic power saving or state holding by dynamically adjusting power supply voltage.

We could also measure circuit performance when the FIR tap number is changed. The bitwise pipelining of the ALA FIR filter ensures that the circuit throughput is not deteriorated as we increase the taps of the FIR filter. Figure 4-9 proves that the FIR throughput does approximately remain unchanged as we increase the tap number from 3-tap to 8-tap and 16-tap. Additionally, Figure 4-10 shows that the circuit energy consumption is still increasing linearly with the tap number, which is a reasonable outcome of the linear hardware complexity increase.

4.2 The ALA Serial Adder

As described in Chapter 2, the ALA schematics are efficient in doing serialized computations because the computing structures can be made local to exploit nearst neighbor computation. Here we first show the serial adder implemented in ALA cells.

Figure 4-11 shows a serial adder implemented with 8 ALA cells. This is of the same functionality but an improved version than the adder shown in Chapter 2, reducing cell count from 12 to 8.

The two addends are streamed in serially from the A and B ports on the left, and the result is streamed out from port Z on the right. The circuit adds up two addends in a bit-by-bit fashion, but the carry-out bit for each bit-addition is feedback to the next bit-addition by the loop on the right half of the circuit. In this way, a

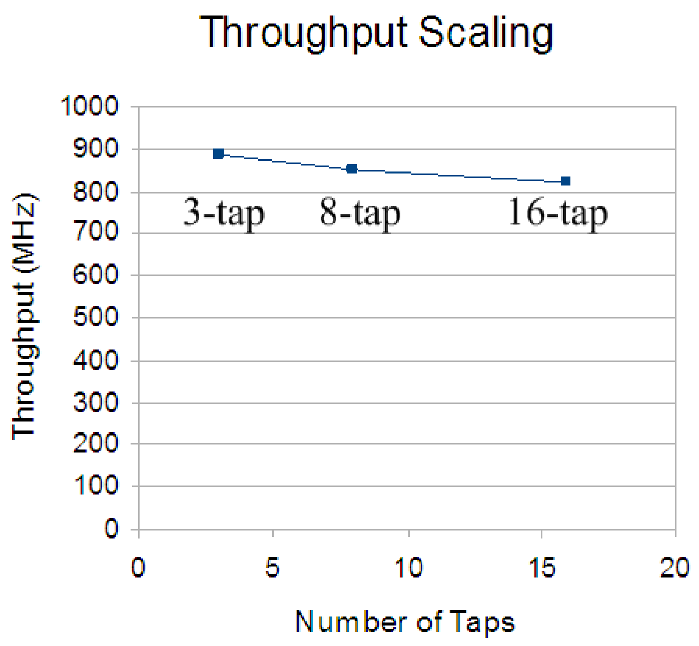


Figure 4-9: Throughput vs. number of taps in the FIR filter.

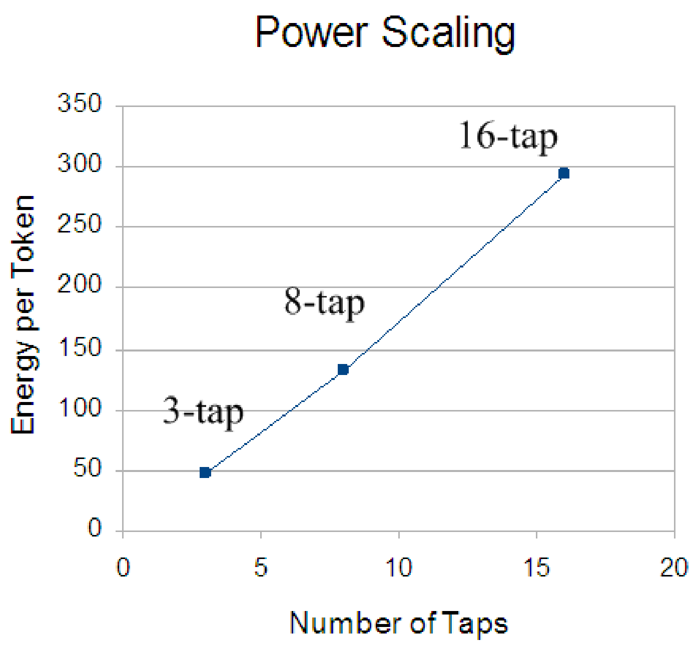


Figure 4-10: Energy consumption vs. number of taps in the FIR filter.

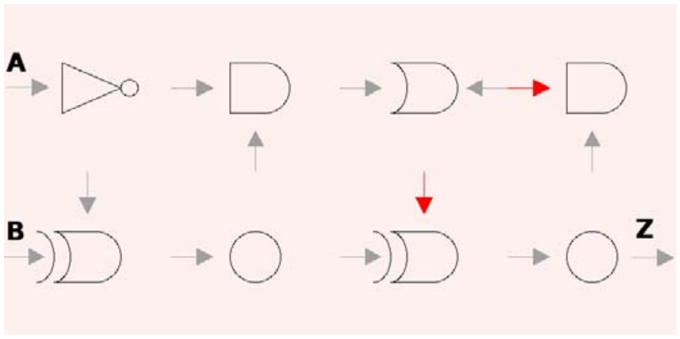


Figure 4-11: Serial adder implemented in ALA.

recursive serial addition is achieved, and the adder is capable of performing addition for arbitrary length addends. The time complexity of the adder is proportional to the bit-length of the addends, i.e., $O(n)$.

The layout of the ALA serial adder is generated very easily using the ALA cell library and the "pick-and-place" design flow. A post-layout simulation is carried out. The ALA serial adder's computation throughput is 664 MHz, and the average energy consumption is 1.69 $pJ/data$.

We could compare this serial adder with the 8-bit ripple adder: to produce an 8-bit word, the seial adder has a word throughput of: 664 / 8 $=$ 83 MHz; an energy consumption of: $1.69 \times 8 \;=\; 13.5$ $pJ/data$. The energy consumption for producing an 8-bit word is comparable between two adders, but the speed of the serial adder is about 13x slower than the parallel ripple adder. However, the serial adder only uses 8 cells to add up numbers with any bit length, whereas the ripple adder uses 34 cells, i.e., 4.25x more hardware is used in the ripple adder. Additionally, the ripple adder hardware cost will scale up linearly as the addend's bit-length increases while the serial adder hardware cost is always 8 cells. Moreover, we need to note that it is not really a fair comparison because the ripple adder uses global interconnects to avoid active cells for signal wiring. Therefore, we could conclude that the serial adder is slower to produce a multiple-bit output due to its serial structure, but much more hardware efficient than the ripple adder.

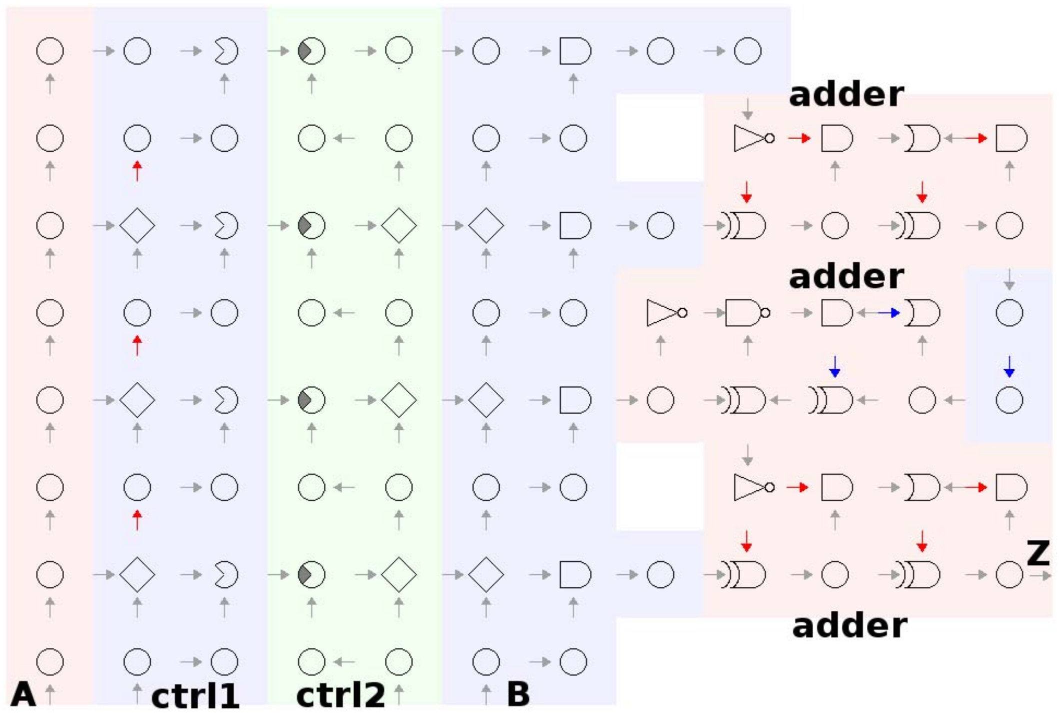


Figure 4-12: Serial 4-bit multiplier implemented in ALA.

4.3 The ALA Multiplier

The ALA multiplier has been described in Chapter 2. For the same reason as in the ALA serial adder, the ALA multiplier also takes up the serial representation for the data and a serialized computation structure. Figure 4-12 shows the ALA schematic for a multiplier. This 4-bit multiplier is also of the same functionality but an improved version than the multiplier shown in Chapter 2; the main improvement is at the more compact serial adder implementation.

The input data is streamed in serially from the A and B port. In addition, the multiplier needs two control sequence input from the ctrl1 and ctrl2 port. These two control sequences defines input data format, in this case ctrl1 is a repeating bit pattern of “0111”; and ctrl2 is repeating “11111110”, which means that each A input data is a 4-bit long data while B is an 8-bit long data. The multiplier does multiplication for a 4-bit input A and an 8-bit input B, producing the output data at the Z port. The multiplication can be performed recursively when multiple pairs of data are present at the inputs.

The layout of the ALA multiplier is generated using the ALA design flow. A post-layout simulation is carried out afterward. The multiplier has a throughput of 409 MHz, and the average energy consumption is 16.2 $pJ/data$.

We could roughly compare this serial multiplier against the 4-bit signed multiplier, but again it is not a fair comparison because the 4-bit signed multiplier uses global interconnects to avoid active cells for signal wiring. To produce an 8-bit word, the seial multiplier has a word throughput of: $409\ /\ 8\ =\ 51.1\ MHz$; an energy consumption of: $16.2 \times 8\ =\ 129.6\ pJ/data$. The energy consumption for producing an 8-bit word for the serial multiplier is 11x of the 4-bit signed multiplier; the speed is about 13x slower. And the area used is approximately the same in both designs. Because the signed multiplier is a fixed design for 4-bit multiplication, it is not surprising that its performance is better. The serial multiplier, on the other hand, is a more general-purpose architecture, which can be reconfigured easily to perform multiplication other than the 4-bit multiplication. Therefore, what we are gaining here is actually design flexibility and extensibility, which is exactly the goal of the Logic Automata model.

4.4 Summary

More ALA circuits could be developed easily with the ALA cell library and the “pick-and-place” design flow. This is a promising approach for fast chip development because it completely gets rid of global design constraints such as the clock and the central processing/control unit and only relies on local designs. This incremental, distributed and scalable design methodology could make impact and change current IC design flow.

Chapter 5

Circuit Design for Analog Logic Automata

As a variant of the Logic Automata family, Analog Logic Automata (AnLA) take continuous values as the states in the computing array, based on the Analog Logic principles [74, 75, 37, 66]. On one hand, Analog Logic circuits work on digital problems using an analog representation of the digital variables, relaxing the state space of the digital system from the vertices of a hypercube to the interior. This lets us gain speed, power, and accuracy over digital implementations. On the other hand, Logic Automata are distributed, scalable and programmable digital computation media with local connections and logic operations. Therefore, here we propose Analog Logic Automata to try to combine advantages from both sides, which relaxes binary constraints on Logic Automata states and introduces programmability into Analog Logic circuits. The localized interaction and scalability of the AnLA enable systematic designs in a digital work flow and provide a new way to solve problems in many different fields, spanning decoding, communication and (biomedical) image processing [7].

5.1 The Analog Logic Principle

Digital computation avoids and corrects errors by sacrificing continuous degrees of freedom. Analog Logic circuits recover this freedom by relaxing the digital states, with each device doing computation in the analog domain, and only quantizing at the output [74]. The analog representations come from either describing digital (binary) random variables with their probability distributions in a digital signal processing problem, or from relaxing binary constraints of an integer programming problem. The preserved information from this analog computation scheme for digital problems gives rise to robust, high-speed, low-power, and cost-effective hardware. Circuit realization examples include decoders [37] and the Noise-Locked Loop (NLL) for direct-sequence spread-spectrum (DSSS) acquisition and tracking, which promise order-of-magnitude improvement over digital realizations [75].

5.1.1 General Description

In essence, Analog Logic is a method for statistical signal processing, in which an associated statistical inference problem is solved by dynamically and locally propagating probabilities in a message-passing algorithm (also known as the sum-product algorithm or the belief propagation algorithm) [41]. To obtain an Analog Logic solution to a practical signal processing problem, three major steps are needed.

Firstly, the statistical signal processing problem is mathematically formulated into a graph. To achieve this, the probabilistic graphical model called Factor Graphs [30] is used as a formal description of the problem, both defining the random variables used in the problem formulation and the relationships between those variables. Note that other graphical models equivalent to Factor Graphs can also be used to model the system under study, among them are Bayesian Networks, Markov Random Fields and Forney Factor Graphs, etc. A detailed tutorial of these mathematical tools can be found in [74], where a comparative summary is provided.

A message-passing algorithm is then derived based on the Factor Graph representation of the problem. By iteratively passing and processing messages on the graph without cycles, the solution of the problem can be guaranteed to converge. Even if the graph is not cycle free, the generalized belief propagation (GBP) algorithm [83] could be used, offering a converging result. The messages in the algorithm are probabilistic information of the random variables in the problem. The information could be the probability distribution of one random variable or the joint probability distribution of several random variables. The processing of the messages, as a result, is actually the marginalization operation over incoming probability distribution functions of some random variables on the corresponding Factor Graph. The marginalization process can in turn be reduced to a series of multiplications and summations. This is also called sum-product operation.

Finally, the units that perform marginalizations are abstracted into Analog Logic gates, or soft gates [38], and the messages become inputs to the soft gates. Subsequently, the signal processing problem is turned into a graph of Analog Logic gates together with interconnections between the gates. The iterative operations on the derived Analog Logic schematic could then lead to a converged result, which is the solution to the original problem.

For a thorough theoretical development of the Analog Logic principles, readers could refer to [74, 66]. And in the following chapters on AnLA applications, we will also give detailed derivations based on specific applications to illustrate the general process of turning a problem into Analog Logic formulations.

5.1.2 Analog Logic Gates

The key to the realizations of Analog Logic principle is the representation of the probability distributions of random variables (messages) and the marginalization over probability distributions (sum-product operations) in the message-passing algorithms. For practice purpose, we only consider problems containing exclusively binary random variables from now on, as digital signal processing tasks are in fact the most common tasks; and binary random variables in the message-passing algorithms lead to simple representations for the mapped Analog Logic operations, as will be seen in the examples provided in the succeeding sections. However, it is important to point

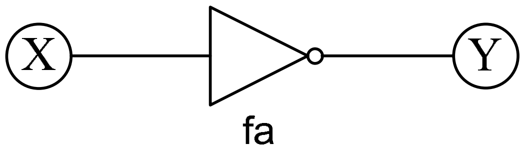


Figure 5-1: Factor graph representation of a soft inverter.

out that in principle, the random variables in the message-passing algorithms could be of any kind.

In a problem that only involves binary random variables, a message that describes the probability distribution of a binary variable could be simply represented by a real number in the interval $[0, 1]$, which could be the probability of the variable being a "0" or a "1".

The message being a real number, there are many different sum-product operations used for the marginalization of the messages. Different marginalization operations correspond to different Analog Logic gates. In additional, some of the Analog Logic gates have their conventional Digital Logic gate counterparts, while some others do not. Here we will introduce some of the mostly used Analog Logic gates (soft gates) in practical applications.

1. Soft Inverter

   The soft inverter (Figure 5-1) is the simplest soft gate. It takes in a probability distribution and inverts the distribution to be its output probability distribution.

   Therefore, if the incoming message is:

$$\mu_{X->f_A}(x) = P_X(x) = \begin{cases} P_X(1), x = 1 \\ P_X(0), x = 0 \end{cases} \tag{5.1}$$

   The output message would be:

$$u_{fA->Y}(y) = P_Y(y) = \begin{cases} P_Y(1), y = 1 \\ P_Y(0), y = 0 \end{cases} = \begin{cases} P_X(0), y = 1 \\ P_X(1), y = 0 \end{cases} \tag{5.2}$$

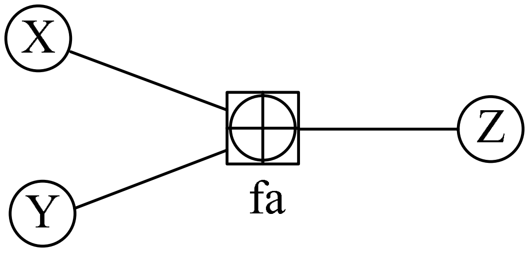


Figure 5-2: Factor graph representation of a soft XOR gate.

2. Soft XOR Gate

Figure 5-2 is the factor graph representing XOR relationship between three random variables. The soft gate XOR (f_a in the figure) connecting the random variables can be expressed as an indication function:

$$f_A(x,y,z) = I(x \oplus y \oplus z = 0) = \begin{cases} 1, x \oplus y \oplus z = 0 \\ 0, x \oplus y \oplus z = 1 \end{cases} \tag{5.3}$$

According to the message-passing algorithm, the messages propagated from variable nodes x and y to the function node f_a are the probability distribution of x and y respectively. The probability distribution for a binary random variable is denoted as:

$$\mu_{X->f_A}(x) = P_X(x) = \begin{cases} P_X(1), x = 1 \\ P_X(0), x = 0 \end{cases} \tag{5.4}$$

$$\mu_{Y->f_A}(y) = P_Y(y) = \begin{cases} P_Y(1), y = 1 \\ P_Y(0), y = 0 \end{cases} \tag{5.5}$$

To compute the resulting message propagated from node f_a to variable node z, we simply do marginalization over the joint probability distribution:

$$\mu_{f_A->Z}(z) = \sum_{x,y \in \{0,1\}} f_A(x,y,z) \cdot \mu_{X->f_A}(x) \cdot \mu_{Y->f_A}(y) \tag{5.6}$$

The message is a function of $z \in \{0,1\}$, for $z = 1$ and $z = 0$, the message is computed:

$$\begin{aligned}\mu_{f_A->Z}(1) &= \sum_{x,y\in\{0,1\}} f_A(x,y,1)\cdot\mu_{X->f_A}(x)\cdot\mu_{Y->f_A}(y)\\ &= \sum_{x,y\in\{0,1\}} I(x\oplus y\oplus 1=0)\cdot\mu_{X->f_A}(x)\cdot\mu_{Y->f_A}(y)\\ &= \sum_{x,y\in\{0,1\}} I(x\oplus y=1)\cdot\mu_{X->f_A}(x)\cdot\mu_{Y->f_A}(y)\\ &= \mu_{X->f_A}(1)\cdot\mu_{Y->f_A}(0)+\mu_{X->f_A}(0)\cdot\mu_{Y->f_A}(1)\\ &= P_X(1)\cdot P_Y(0)+P_X(0)\cdot P_Y(1)\end{aligned} \tag{5.7}$$

$$\begin{aligned}\mu_{f_A->Z}(0) &= \sum_{x,y\in\{0,1\}} f_A(x,y,0)\cdot\mu_{X->f_A}(x)\cdot\mu_{Y->f_A}(y)\\ &= \sum_{x,y\in\{0,1\}} I(x\oplus y\oplus 0=0)\cdot\mu_{X->f_A}(x)\cdot\mu_{Y->f_A}(y)\\ &= \sum_{x,y\in\{0,1\}} I(x\oplus y=0)\cdot\mu_{X->f_A}(x)\cdot\mu_{Y->f_A}(y)\\ &= \mu_{X->f_A}(1)\cdot\mu_{Y->f_A}(1)+\mu_{X->f_A}(0)\cdot\mu_{Y->f_A}(0)\\ &= P_X(1)\cdot P_Y(1)+P_X(0)\cdot P_Y(0)\end{aligned} \tag{5.8}$$

We can see clearly from the above result that the message sent to random variable z is a probability distribution:

$$\mu_{f_A->Z}(z) = \begin{cases} P_X(1)\cdot P_Y(0)+P_X(0)\cdot P_Y(1), z=1\\ P_X(1)\cdot P_Y(1)+P_X(0)\cdot P_Y(0), \mathrm{z}=0\end{cases} \tag{5.9}$$

This probability distribution coincides with the probability distribution of the random variable $Z' = X\oplus Y$. It is exactly because of this "coincidence" that facilitates the concept of soft XOR gate, i.e., we can mimic the process of message propagating and marginalization by computing the probability distribution of the equivalent random variable Z', given the probability distributions of input random variables X and Y.

Another issue that facilitates the realization of soft gates is the fact that we are dealing with binary random variables. It is relatively easy to represent and send the probability distribution information because either the value of $P_X(1)$ or $P_X(0)$ suffices the task of describing the probability distribution of the binary

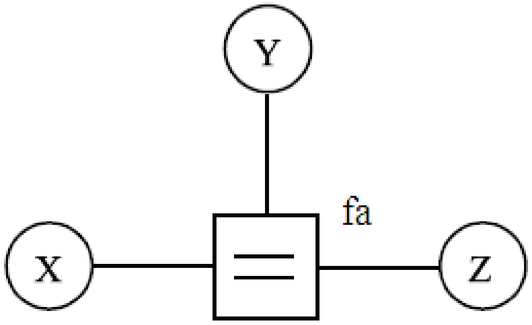


Figure 5-3: Factor graph representation of a soft EQ gate.

random variable X. If it were a discrete random variable with several possible values, we would have to send a vector of values expressing the probability distribution function; if it were a continuous random variable, then the whole method of expressing the distribution function would have to be re-designed.

3. Soft EQ Gate

   The factor graph representing the "EQUAL" (EQ) relationship between three random variables is drawn in Figure 5-3. The soft EQ (f_a in the figure) connecting the random variables is defined as:

$$f_A(x, y, z) = I(x = y = z) = \begin{cases} 1, & x = y = z \\ 0, & x, y, z \text{ not all equal} \end{cases} \tag{5.10}$$

   The messages propagated from variable nodes x and y to the function node f_a are still the probability distribution of x and y respectively.

   The message propagated from node f_a to variable node z is obtained similarly by marginalize over the joint probability distribution as follows:

$$\begin{aligned} \mu_{f_A->Z}(1) &= \sum_{x,y\in\{0,1\}} f_A(x, y, 1) \cdot \mu_{X->f_A}(x) \cdot \mu_{Y->f_A}(y) \\ &= \sum_{x,y\in\{0,1\}} I(x = y = 1) \cdot \mu_{X->f_A}(x) \cdot \mu_{Y->f_A}(y) \\ &= \mu_{X->f_A}(1) \cdot \mu_{Y->f_A}(1) \\ &= P_X(1) \cdot P_Y(1) \end{aligned} \tag{5.11}$$

$$\begin{aligned} \mu_{f_A->Z}(0) &= \sum_{x,y\in\{0,1\}} f_A(x,y,0)\cdot \mu_{X->f_A}(x)\cdot \mu_{Y->f_A}(y) \\ &= \sum_{x,y\in\{0,1\}} I(x=y=0)\cdot \mu_{X->f_A}(x)\cdot \mu_{Y->f_A}(y) \\ &= \mu_{X->f_A}(0)\cdot \mu_{Y->f_A}(0) \\ &= P_X(0)\cdot P_Y(0) \end{aligned} \tag{5.12}$$

If we normalize the message sent to random variable z so that the message looks like a probability distribution, we would get the following expression:

$$\mu_{f_A->Z}(z) = \begin{cases} P_X(1)\cdot P_Y(1)/\gamma, z=1 \\ P_X(0)\cdot P_Y(0)/\gamma, \mathrm{z}=0 \end{cases} \tag{5.13}$$

Where the normalization factor is: $\gamma = P_X(1)\cdot P_Y(1) + P_X(0)\cdot P_Y(0)$.

The above three soft gates are quite representative for what we will be using later. Other soft gates could be easily derived following the same approaches. In particular, common digital logic gates such as AND, OR, NAND, and NOR could be “softened” in the same way as the soft XOR gate. And there is a soft “Unequal” (UNEQ) gate frequented in use, which could be obtained by cascading a soft gate EQ and a soft inverter.

Finally, we need to point out that all above soft gate derivations are based on the assumption that the input random variables are statistically independent. If the input variables are not independent, as might be the case in some loopy factor graphs, the formulation would be much more complicated.

5.2 Analog Logic Automata = Analog Logic + Logic Automata

As discussed in the preceding section, because Analog Logic hardware directly maps the underlying message-passing algorithms, it is capable of solving a wide range of different inference and statistical problems. However, to the best of our knowledge, most of the hardware realizations in the literature are in an ad-hoc fashion.

5.2.1 Embedding Analog Logic into the Reconfigurable Logic Automata framework

Logic Automata as a programming model has already shown great potential in previous chapters, for its scalable and universal computation power, while reflecting the nature of many complex physical systems. But previous Logic Automata implementations were exclusively dealing with digital computations.

In an attempt to embed Analog Logic processing into the generic framework of Logic Automata to implement programmable message-passing algorithms, we are able to invent the new computational model of Analog Logic Automata (AnLA). And of course, AnLA differs from conventional digital computational models in that it has continuous states, which preserve the information contained between "0" and "1".

The new model exploits virtues from both Analog Logic and Logic Automata gracefully. Firstly, message-passing algorithms naturally indicate that computations going on locally. This locality is in perfect agreement with Logic Automata discipline, leading to local and distributive computation behavior for AnLA. Secondly, Reconfigurable Logic Automata empowers programmability. As message-passing algorithms are very broad in terms of application, this allows AnLA to become a versatile processor. Thirdly, because signal processing is done in analog domain, AnLA is able to avoid discarding huge amount of potentially useful information from the beginning, which leads to large power/cost savings, and/or extremely high speed operations, and/or satisfactory computation capability with very limited physical resources, etc.

5.2.2 Relaxing Logic Automata with Analog States

Another way to view the novelty of the AnLA model is from the mathematical programming point of view. AnLA is an extension and relaxation from the traditional Cellular Automata (CA) [2] architecture. First of all, the extension means that Logic Automata is a special case of the CA model, in that each Logic gate of the Logic Automata can be equivalently replaced by a patch of CA elements with certain local interconnections and update rules [2]. Both Logic Automata and CA are universal and equivalent to each other, but Logic Automata has a more compact representation and smaller form factor. Secondly, AnLA relaxes the Boolean states of the old CA model. Historically, CA was formulated and implemented as a completely discrete-state (mostly Boolean), discrete-time model. Programming on a Boolean CA array with a certain update rule for a particular signal processing problem is the mathematical equivalence of a constraint optimization problem with all the state variables being binary and constraints being combinatorial. However, this discrete operation mode is not the fundamental reason for the success of CA in modeling the physical world. Instead, CA gains its power largely from the local and distributed interaction. The introduction of Analog Logic and message-passing algorithm replaces binary state variables with probabilistic information of those variables, or more specifically, probability distribution of the binary state variables, while still preserving locality and distributed organization of CA model. Correspondently, the combinatorial constraints are relaxed into continuous constraints; and the discrete optimization problem is transformed into a relaxed optimization problem.

5.2.3 More Discussion

Finally, I would like to further clarify some of the easily mistaken concepts around AnLA, which probably would be confusing.

1. AnLA is always compatible with traditional digital signal processing hardware, because output signal can always be easily converted back into digital domain. And in most of the times, the digitized final results are all what people actually need.

2. The principles described here apply equally to clocked and un-clocked (asynchronous) automata. While in the work that follows, we focus on the clocked AnLA, in which all elements update synchronously at the command of a global clock; we are not excluding the possibility of an improved asynchronous version of AnLA, i.e., Asynchronous Analog Logic Automata (AAnLA). The asynchronous model keeps the same states and computation, but further localizes

time by removing the global clock.

3. AnLA is different from a traditional Analog Field Programmable Gate Arrays (AFPGA) [26] in that AnLA conceptually work in digital space with analog representations. Even the Logic Automata architecture is different from Field Programmable Gate Arrays (FPGA) because Logic Automata interconnects are completely local, as compared to the global connections in a FPGA.

5.3 Circuit Design for the Analog Logic Automata

The hardware realization for the AnLA matches upper level schematic nicely, thanks to the locality enforced by the Logic Automata model and the natural analog representation of the message from the Analog Logic principle. In general, the AnLA model is a continuous state, discrete time model with reconfigurable connectivity and functionalities. A global clock serves as a uniform reference for cell updates in the array, and the cell circuits works in current mode to perform analog computation. A target message-passing algorithm modeled with a factor graph is firstly represented by the programmed local connections of the AnLA array. Then the probabilistic messages propagating on the factor graph are mapped into physical degrees of freedom as electrical currents. The analog computations could be reduced to a series of multiplication and summation operations, as mentioned before. These operations can in turn be abstracted to a family of soft gates, which will be realized by each of the programmable Analog Logic cell on the AnLA grid. Because we employ current mode operations in underlying circuits, a summation could be trivially attained by a wire joint; and the multiplications are implemented with analog Gilbert Multipliers [21], exploiting the well-known Translinear principle [22] by MOS transistors in weak inversion or Bipolar Junction Transistors (BJTs).

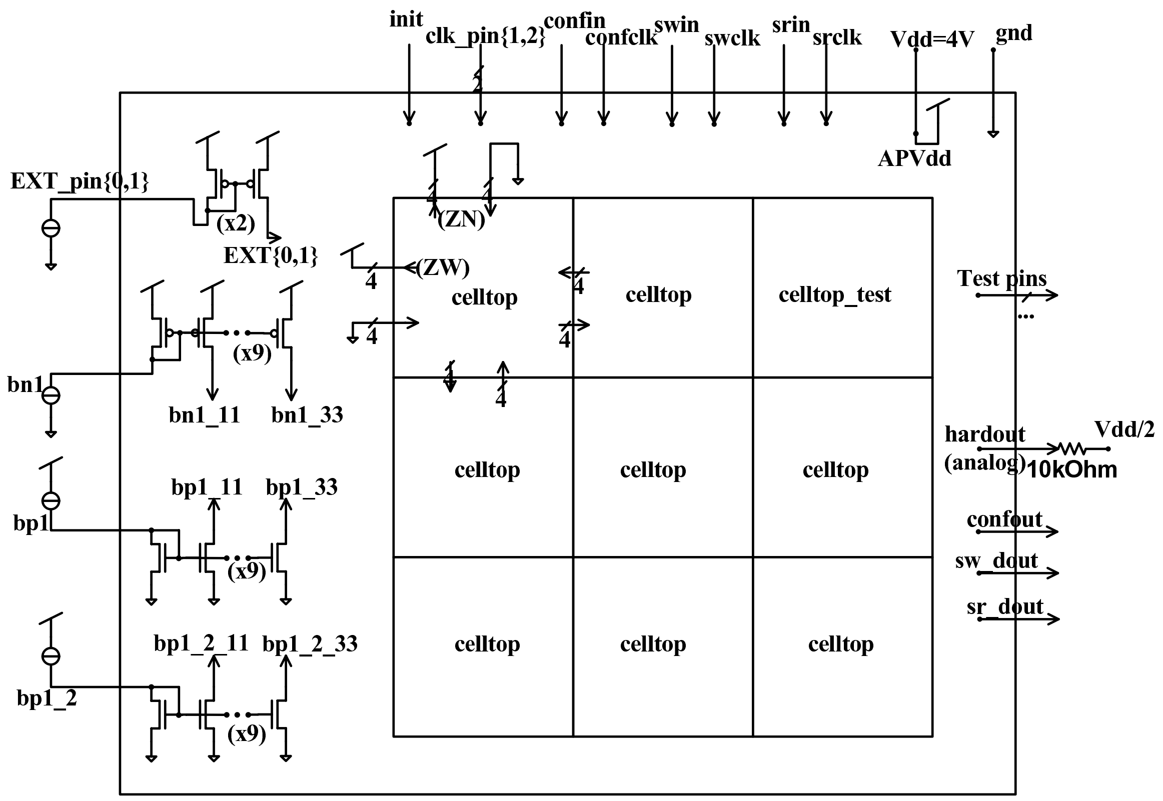


Figure 5-4: Architecture of a 3×3 AnLA array.

5.3.1 Architectural Overview

The AnLA is composed of a grid of reconfigurable Analog Logic cells and operates synchronously with a global clock. Each cell in the array is able to communicate with its North, East, South, West neighbors in the rectangular grid, based on the programmed connectivity. Figure 5-4 shows the architecture of a 3×3 AnLA array that we prototyped in silicon.

Figure 5-5 goes on to show the block diagram of one AnLA cell in the array. Each cell is implemented in mixed-signal circuits, with analog circuitry (blocks shown to the right of the dashed line) doing Analog Logic computation and digital control circuitry (blocks shown to the left of the dashed line) taking care of the configurations in every single cell. The cell stores an analog state (denoted as Z) and interacts with its rectangular neighbors. The two analog inputs (denoted as X and Y) of the cell could be any combination of the outputs from its four neighbors, the current state of the

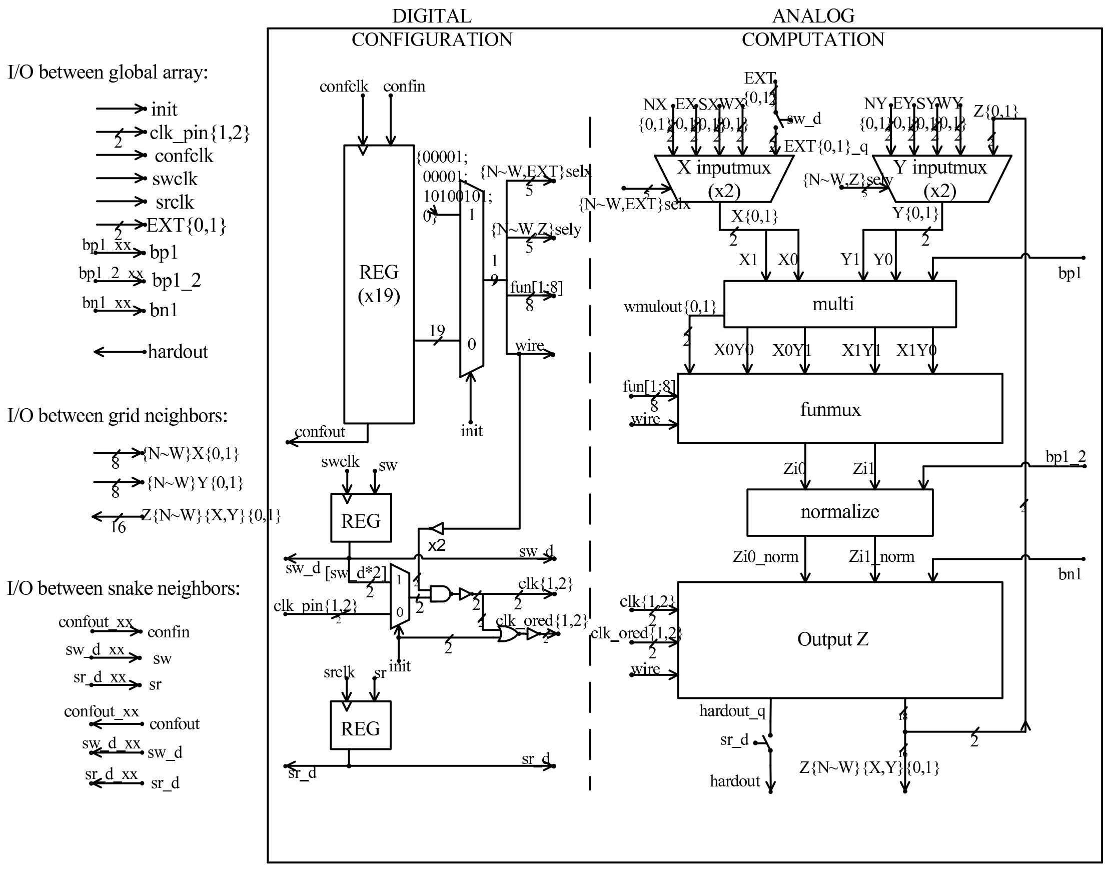


Figure 5-5: Block diagram of one AnLA cell.

cell itself, or external inputs, depending on the connection configuration (as indicated by the X and Y "input mux" block). In every clock phase, each cell performs an Analog Logic computation according to its function configuration. The cell state is updated and accessible to neighboring cells in the next clock phase.

As specified in the model, all state variables, e.g. X, Y and Z, can be viewed as binary random variables. In current-mode circuits, the probability distributions are represented by

$$\begin{aligned} I_{Z1} &\propto P(Z=1) \equiv P_Z(1), \\ I_{Z0} &\propto P(Z=0) \equiv P_Z(0). \end{aligned} \tag{5.14}$$

With the above representation, the message-passing algorithms are reduced to a series of summations and multiplications. The summation over several variables is implemented by merging their respective currents, which effectively takes average on probability distributions of those random variables. The multiplication units are programmable soft gates implemented with Gilbert Multipliers (as indicated by the "multi" block). Note that only 2-input soft gates are used in our AnLA architecture because they suffice all computations.

As an example to help understanding, a 2-input soft XOR gate, as mentioned previously, performs a statistical version of the XOR operation. The probability distribution of Z is derived from incoming probability distributions of X and Y as:

$$\begin{bmatrix} P_Z(1) \\ P_Z(0) \end{bmatrix} = \begin{bmatrix} P_X(1) \cdot P_Y(0) + P_X(0) \cdot P_Y(1) \\ P_X(0) \cdot P_Y(0) + P_X(1) \cdot P_Y(1) \end{bmatrix} \tag{5.15}$$

Similarly, many more soft gates, including 2-input soft AND, NAND, OR, NOR, and 1-input soft Inverter, can be derived from digital gates. But soft gates without digital counterparts also exist. For example, the 2-input soft EQUAL (EQ) gate, which is frequently used when independent information of two random variables is combined, indicating how similar these variables are, is defined as

$$\begin{bmatrix} P_Z(1) \\ P_Z(0) \end{bmatrix} = \gamma \begin{bmatrix} P_X(1) \cdot P_Y(1) \\ P_X(0) \cdot P_Y(0) \end{bmatrix} \tag{5.16}$$

Its complementary, soft UNEQUAL (UNEQ) is defined as

$$\begin{bmatrix} P_Z(1) \\ P_Z(0) \end{bmatrix} = \gamma \begin{bmatrix} P_X(1) \cdot P_Y(0) \\ P_X(0) \cdot P_Y(1) \end{bmatrix} \tag{5.17}$$

Where γ in (5.16) and (5.17) is the normalization factor satisfying $P_Z(1)+P_Z(0) = 1$.

All the above equations indicate that a programmable soft gate can be made by selectively steering and merging the Gilbert Multiplier output currents with switches (the "funmux" block) before the normalization (the "normalize" block) and gate outputs (the "output" block).

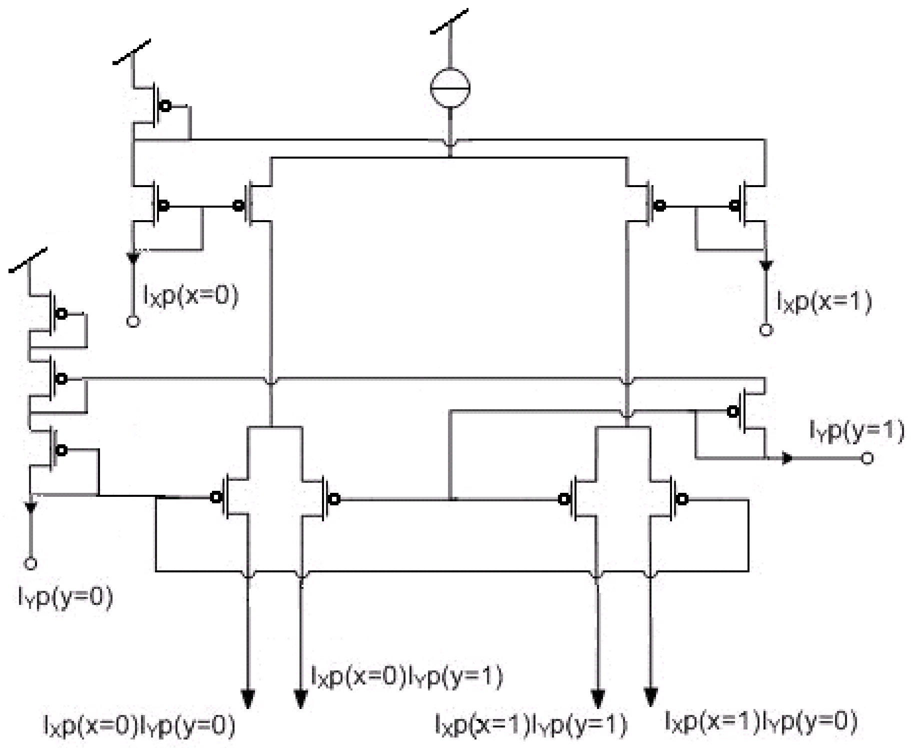


Figure 5-6: Schematic of a Gilbert Multiplier.

5.3.2 Detailed Design Descriptions for the Core Computation Circuit

1. Analog Multiplier

   The core schematic of the analog multiplier is shown in Figure 5-6. We use sub-threshold MOS transistors in a translinear configuration. This implies that the current density must be small, thus the tail current of the multiplier and the transistor sizes are designed accordingly. Also, diode connected transistors are added to the sources of the input current mirrors, so that all transistors in the translinear circuit are saturated for accurate multiplication.

   The multiplier's four current inputs are proportional to the four probability terms: P(X=1), P(X=0), P(Y=1), P(Y=0), corresponding to the two input messages of the soft gate. Consequently, the output currents are proportional to the products of the probability terms, as indicated in the figure. This allows us to implement various Analog Logic functions:

By connecting wires $I_X p\left(X=0\right) \cdot I_Y p\left(Y=0\right)$ and $I_X p\left(X=1\right) \cdot I_Y p\left(Y=1\right)$ together as the output $I_Z p\left(Z=0\right)$; and wires $I_X p\left(X=0\right) \cdot I_Y p\left(Y=1\right)$ and $I_X p\left(X=1\right) \cdot I_Y p\left(Y=0\right)$ together as the output $I_Z p\left(Z=1\right)$, the gate performs XOR operation:

$$\begin{cases} P_Z\left(0\right) = P_X\left(0\right) \cdot P_Y\left(0\right) + P_X\left(1\right) \cdot P_Y\left(1\right) \\ P_Z\left(1\right) = P_X\left(1\right) \cdot P_Y\left(0\right) + P_X\left(0\right) \cdot P_Y\left(1\right) \end{cases} \tag{5.18}$$

Similarly, the soft AND gate with the following expression:

$$\begin{cases} P_Z\left(0\right) = P_X\left(0\right) \cdot P_Y\left(0\right) + P_X\left(0\right) \cdot P_Y\left(1\right) + P_X\left(1\right) \cdot P_Y\left(0\right) \\ P_Z\left(1\right) = P_X\left(1\right) \cdot P_Y\left(1\right) \end{cases} \tag{5.19}$$

Could be realized by connecting wires $I_X p\left(X=0\right) \cdot I_Y p\left(Y=0\right)$, $I_X p\left(X=0\right) \cdot I_Y p\left(Y=1\right)$ and $I_X p\left(X=1\right) \cdot I_Y p\left(Y=0\right)$ together as the output $I_Z p\left(Z=0\right)$; wire $I_X p\left(X=1\right) \cdot I_Y p\left(Y=1\right)$ as the output $I_Z p\left(Z=1\right)$.

Furthermore, the soft EQ gate is realized by only connecting the wire $I_X p\left(X=0\right) \cdot I_Y p\left(Y=0\right)$ to output $I_Z p\left(Z=0\right)$; and the wire $I_X p\left(X=1\right) \cdot I_Y p\left(Y=1\right)$ to output $I_Z p\left(Z=1\right)$; and followed by a normalization process. Other current outputs are discarded. Additionally, the inverse, soft UNEQ, is obtained by simply flipping the connections of soft EQ.

However, a full collection of all possible Analog Logic functions can be implemented in a programmable fashion with the introduction of a switching block, which is to be discussed next.

2. Switching Function Mux

All Analog Logic functions can be implemented as a programmable unit with the switching structure as shown in Figure 5-7. The structure is used for selectively steering output currents from the analog multiplier towards the normalization and output circuitry. The control signals (fun1∼fun8) decide where the currents are steered out to. Currents are steered to either z0 or z1; or simply discarded.

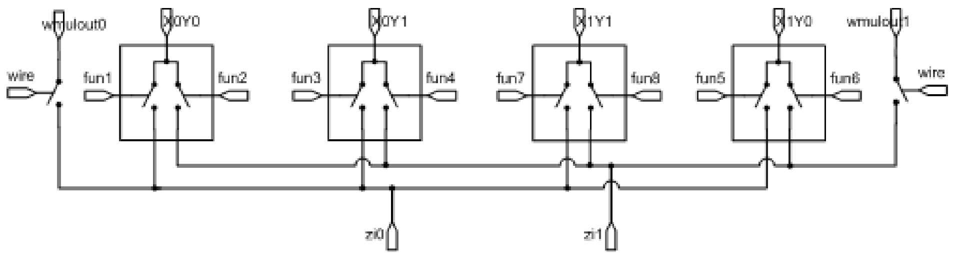


Figure 5-7: Switching function mux.

Turning on two switches from the same input branch is prohibited.

We would like to note here that although soft gates descended from digital gates usually normalize their output currents automatically, soft gates like EQ and UNEQ do require normalization to ensure inter-cell correctness. Therefore, after passing through the 8 switches that determine the functionality, the two output currents proportional to the cell state should always go through a normalization circuitry to make sure the cell states on the entire array are in accordance to each other in terms of the absolute magnitude.

3. Analog State Storage and Output Stage

The computed analog state needs to be stored locally; and be driven to the neighboring cells at the next clock phase. Figure 5-8 shows the schematic implementing analog storage in log-domain and output stage. In the schematic, M_0 and M_1 must be well matched, and the capacitor that stores the gate voltage must be much greater than the gate parasitic capacitance of M_0 or M_1. In order to charge and discharge the large capacitor within a clock phase, M_2 and M_3 are added, to form a "super-buffer" with low output impedance. When the current going into M_0 suddenly increases, the gate voltage of M_2 jumps up. Now $M2$ puts more current into the capacitor than M_3 draws, charging it up. When the current going into M_0 suddenly decreases, the gate voltage of M_2 drops, which weakens M_2, so the capacitor discharges. In our test chip, the gate voltage of M_3 is adjustable by external bias current bn_1 to ensure the stability of the super-buffer.

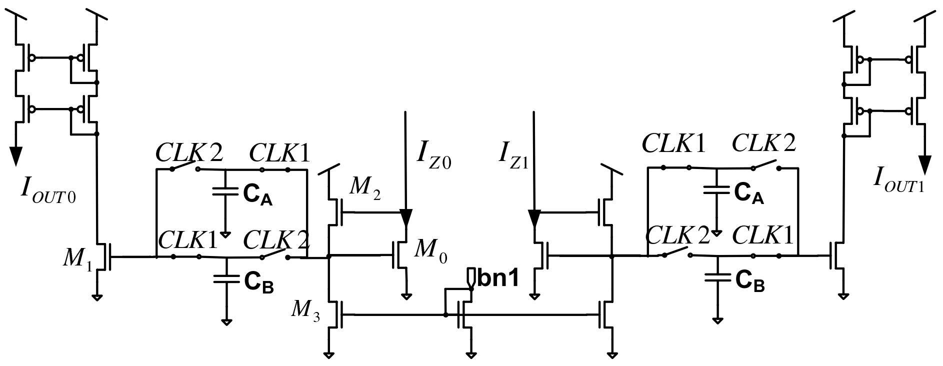


Figure 5-8: Analog storage and output stage.

> The output stage operates on the alternating clocks CLK_1 and CLK_2. In the first clock phase, capacitor A (C_A) is being written into, and capacitor B (C_B) is connected to the gate of M_1, which goes through cascode current mirrors to send the output currents to the neighbor cells. In the next clock phase, C_B is being written into, and C_A is connected to M_1. This results in the functionality described in the cell architecture.

Overall, the complete circuit schematic of one AnLA cell is shown in Figure 5-9. As a proof-of-concept experiment, we fabricated a 3×3 AnLA chip in the AMI 0.5μm CMOS process[1], with an area of 1.5×1.5mm^2 and $4V$ voltage supply. The array can work at $50kHz$ and the power consumption is 64μW, including both digital and analog circuits. The chip layout and die photo are shown in Figure 5-10 and Figure 5-11, respectively.

[1]As the first prototyping chip, we used a relatively old process technology. As a result, only 9 AnLA cells are embedded and almost half of the chip area is occupied by digital configuration circuitry (19 D Flip-Flops per cell) to get full programmability. However, high density AnLA array would be possible if we scale down the silicon process and reduce configuration bits.

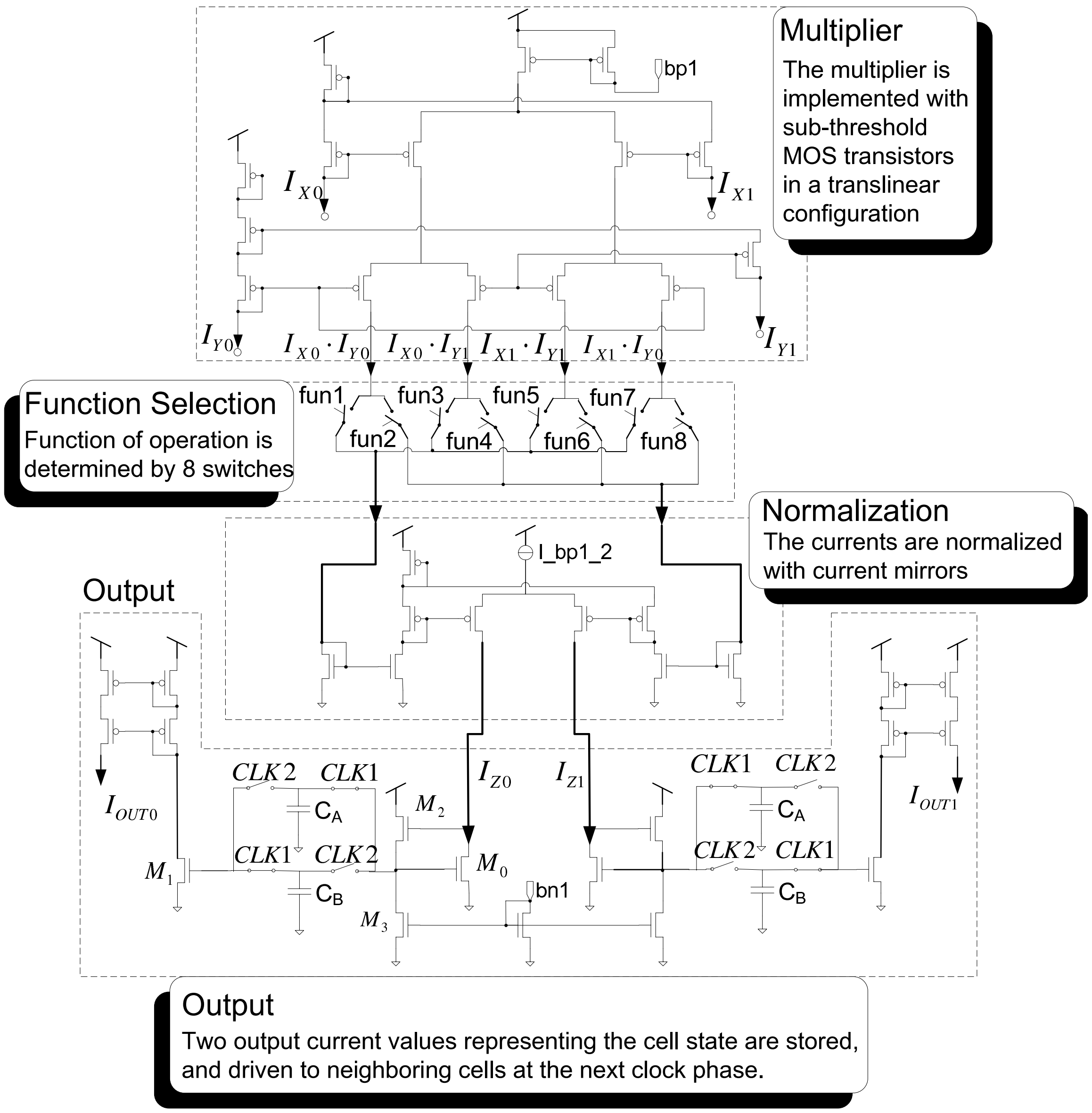

Multiplier
The multiplier is implemented with sub-threshold MOS transistors in a translinear configuration
bp1
I_{X0}
I_{X1}
I_{Y0}
I_{Y1}
$I_{X0} \cdot I_{Y0}$
$I_{X0} \cdot I_{Y1}$
$I_{X1} \cdot I_{Y1}$
$I_{X1} \cdot I_{Y0}$
Function Selection
Function of operation is determined by 8 switches
fun1
fun2
fun3
fun4
fun5
fun6
fun7
fun8
Normalization
The currents are normalized with current mirrors
I_bp1_2
Output
I_{OUT0}
I_{OUT1}
$CLK2$
$CLK1$
C_A
C_B
M_1
M_2
M_0
M_3
I_{Z0}
I_{Z1}
bn1
Output
Two output current values representing the cell state are stored, and driven to neighboring cells at the next clock phase.

Figure 5-9: Core schematic of an AnLA cell.

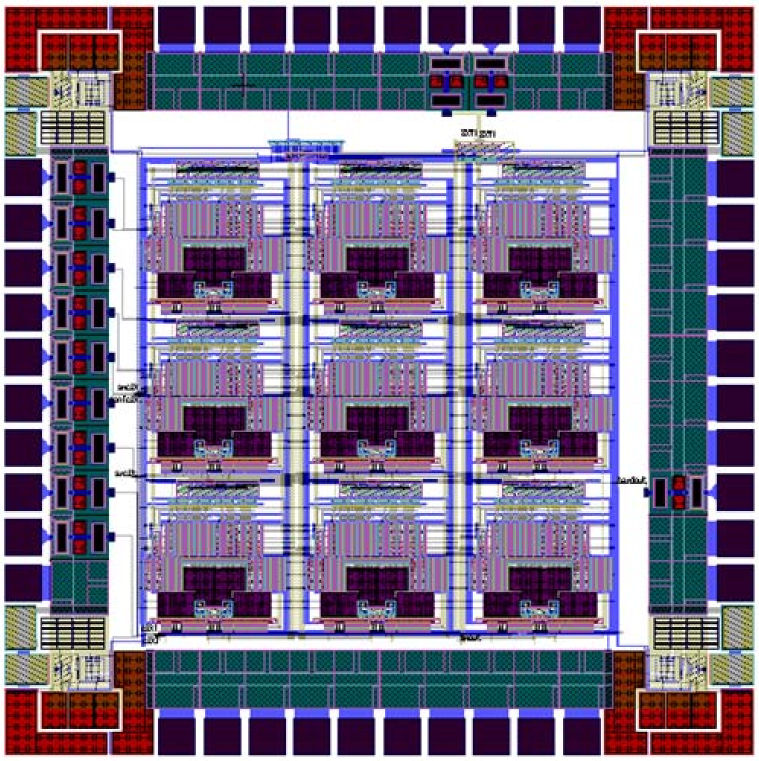


Figure 5-10: Chip layout.

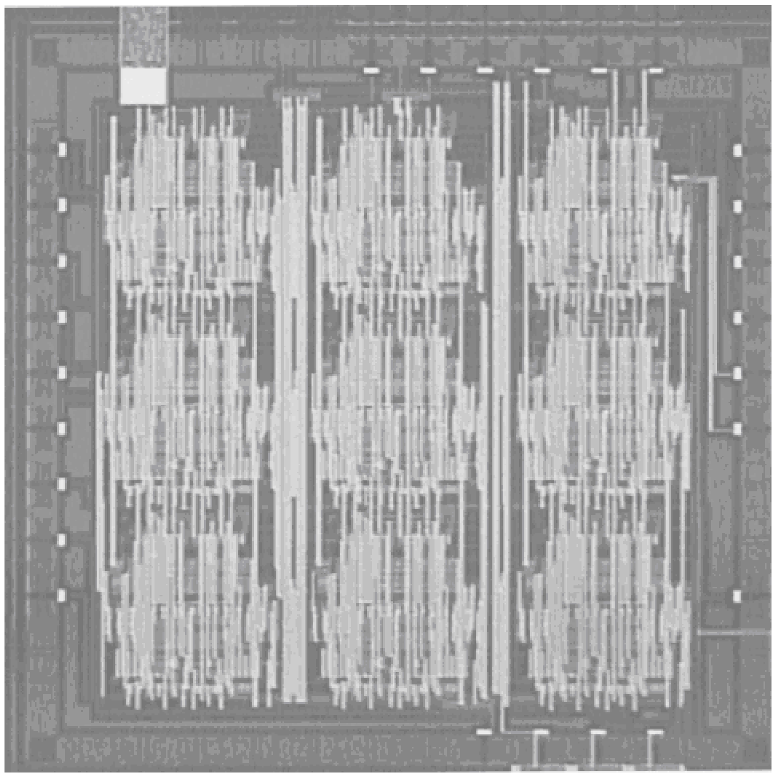


Figure 5-11: Chip die photo.

Chapter 6

Applications of Analog Logic Automata

Reconfigurable Analog Logic Automata implementing message-passing algorithms have the potential to solve many kinds of statistical inference and signal processing problems with much more efficiency than their digital counterparts. Some active fields include pseudorandom signal (also called PN sequence or m-sequence) synchronization, error-correcting code (ECC) decoding and statistical image processing.

Historically, Hagenauer et al. [25] proposed the idea of analog implementation for the maximum a posteriori (MAP) decoding algorithm, but without actual transistor implementations. The scheme for iterative m-sequence synchronization by soft sequential estimation was also reported [81, 82]. The analog implementations of the Viterbi algorithm were observed in [58, 38], with BiCMOS and sub-threshold CMOS hardware realizations, respectively. Later, the theoretical work done by Loeliger et al. generalized various statistical inference algorithms into the generic message-passing framework called the sum-product algorithm, which operates on a certain graphical model [36, 30]. This generalization include so broad areas of interest as decoding algorithms, Bayesian networks, Kalman filtering and other complex detection and estimation algorithms. Research on computational models and simulations for statistical image processing [69, 70], early machine vision [16, 15] and 2-D Gaussian Channel Estimation [60] indicated large performance win over traditional methods. Discrete component analog circuit realization of the noise-locked loop (NLL) algorithm for direct-sequence spread-spectrum acquisition and tracking also demonstrated orders-of-magnitude speed/power advantage over digital implementation [75]. Much work [74, 66, 75, 38] has used the concept of Analog Logic in the mapping from system level algorithm to transistor level implementation. We continue to take advantage of the Analog Logic features with added strength of programmability as well as a notion of mathematic programming in this work.

6.1 Reconfigurable Noise-Locked Loop

6.1.1 General Description

The Noise-Locked Loop (NLL) is a generalization of the Phase-Locked Loop (PLL) [74]. Instead of synchronizing to a sinusoidal waveform, an NLL relaxes this constraint and can synchronize to a more complex periodic pattern produced by a given Linear Feedback Shift Register (LFSR).

The non-programmable NLL for pseudorandom signal synchronization was reported in [74, 75, 66], where theoretical derivation of NLL as forward-only message passing in the corresponding factor graph can be found in detail. Intuitively, to infer and be synchronized to the pseudorandom signal from the transmitter, the NLL receiver provides a locking mechanism that only reinforces the correct pseudorandom signal pattern. This locking mechanism is achieved by mimicking the process that generates the pseudorandom signal in the transmitter system and producing a local replica. This is analogous to a PLL system in that the VCO in a PLL is used for producing a local replica pseudorandom signal of the sinusoid signal, whose phase is compared with the incoming signal in a Phase Detector (PD); while in a NLL system, we use a LFSR of the same structure as in the transmitter for the reproduction of the pseudorandom signal, which is compared with the incoming pseudorandom signal through the use of a soft EQ gate. Such PLL-inspired systems could also be seen in other interesting applications [3, 79].

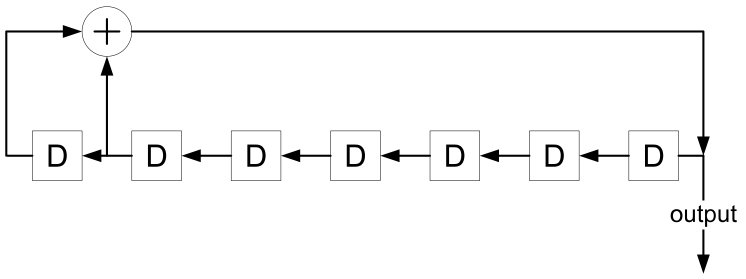


Figure 6-1: 7-bit LFSR transmitter.

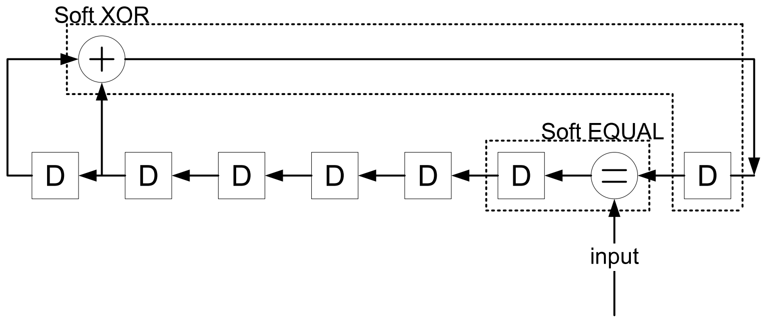


Figure 6-2: 7-bit NLL receiver.

To gain understandings of how the NLL could be constructed, assume that a LFSR of a fixed structure is used as a transmitter to generate a digital pseudorandom signal; the NLL receiver for the synchronization to the input signal can be obtained by the following modifications on the original digital LFSR:

1. Transform all digital gates of the LFSR into corresponding soft gates, i.e., the digital delay elements become analog delay elements, delaying analog states by one clock cycle; and the XOR gate becomes the soft XOR gate.

2. Insert a soft EQ gate into the soft LFSR at a proper position[1], for the comparison of the difference between the synthesized and input pseudorandom signal.

By softening the components in the LFSR and adding a soft EQ gate, we obtain the corresponding NLL that synchronizes to a pseudorandom signal. And within the

[1]Although in principle can be placed at anywhere of the delay train, soft EQ gate should avoid being placed right after the output of a soft XOR gate in actual implementation because each AnLA cell intrinsically contains a soft delay element.

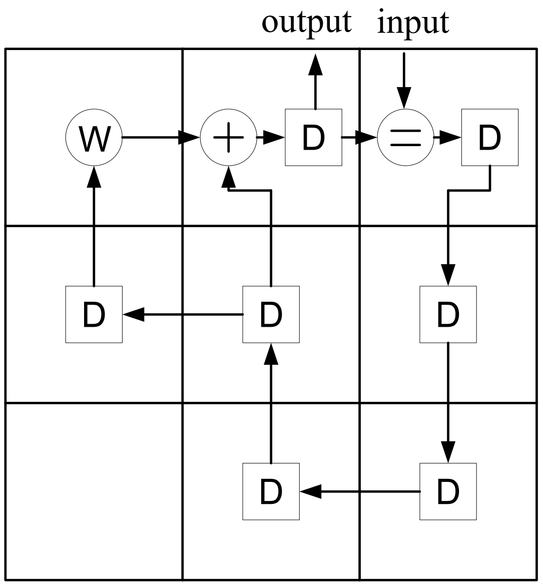


Figure 6-3: 7-bit NLL receiver implemented on 3×3 AnLA.

3×3 AnLA framework, we can match different LFSR transmitters with corresponding NLL receivers up to 7-bit long by changing the array configuration[2]. Figure 6-1 and 6-2 show the 7-bit LFSR transmitter and its corresponding NLL receiver, respectively. The box denoted by "D" is representing a unit delay. And the dashed boxes indicate AnLA cells performing soft XOR and EQ functions, with a unit delay. The actual implementation on the AnLA is shown in Figure 6-3, where the top left cell is configured as a WIRE gate, denoted by "W". The WIRE function bypasses the input directly to the output without any delay. It is introduced for more routing flexibility in the rectangular-connection-only array.

6.1.2 Test Results

The 3×3 AnLA chip was tested for the NLL applications. Different bit numbers from 3-bit to 7-bit are tested and proved to be working correctly. The 7-bit NLL locking and tracking dynamics are shown in Figure 6-4, in which the three waveforms are (from top to bottom): An attenuated version of the clean pseudorandom signal generated by the 7-bit LFSR; the noisy pseudorandom signal corrupted by white

[2]The reason that we are unable to implement a 8-bit or 9-bit NLL is due to the routing overhead generated by the lack of diagonal connections in the array.

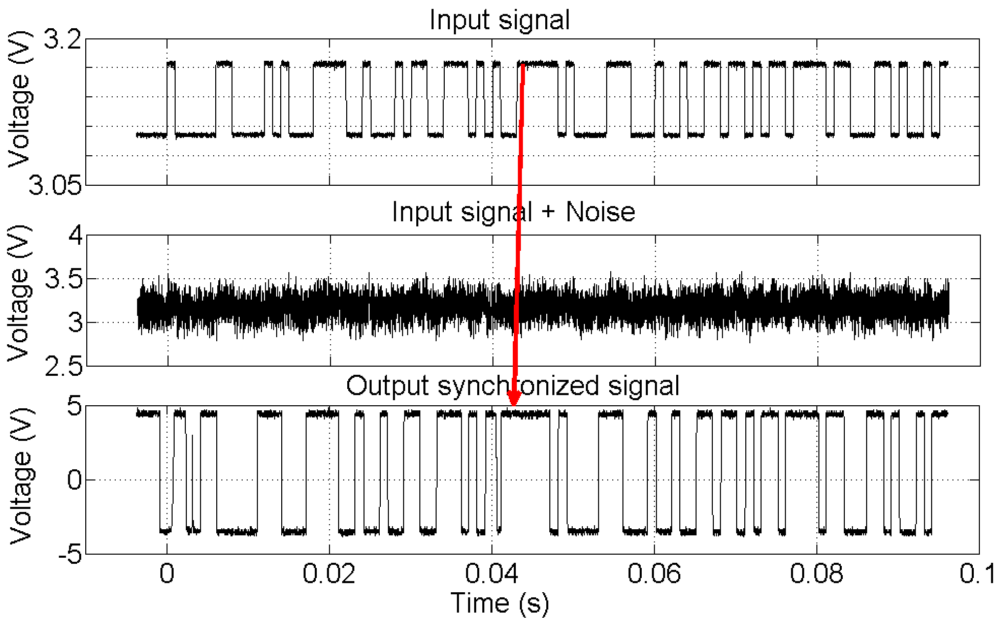


Figure 6-4: 7-bit NLL test results: the locking and tracking dynamics.

Gaussian noise, which is the input of the 7-bit NLL; and the 7-bit NLL output signal, which is clearly synchronized to the LFSR. We can see the NLL locks onto the input signal after 43 clock phases. In this measurement, the input signal current swing is $47.4nA$ and the measured lowest SNR is -6.87dB. A plot of the Bit Error Rate (BER) as a function of input Signal-to-Noise Ratio (SNR) is also given in Figure 6-5, which indicates correct working even when the signal power is less than the white noise power.

6.2 Error Correcting Code Decoding

As already have been mentioned in the preceding chapter, the Error Correction Code (ECC) decoding algorithms such as the Forward-backward algorithm (FBA), Viterbi algorithm, iterative turbo code decoding and Kalman Filter, could be typically mapped into some kind of Bayesian inference problems. Relevant research [83] generalized them with some kind of local message-passing algorithms, thus embedding them into the context of statistical inference algorithm research and message-passing

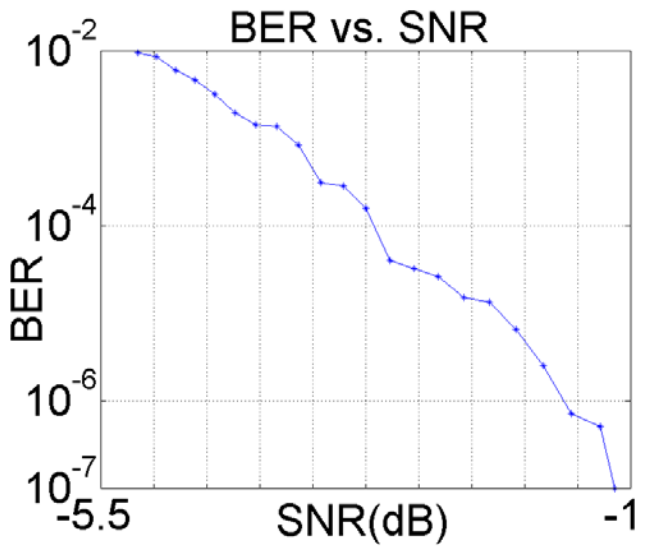


Figure 6-5: 7-bit NLL test results: the Bit Error Rate vs. the Signal-to-Noise Ratio plot.

algorithm development.

In this section, the mathematical formulation of the decoding problem is firstly defined; and then an exemplary FBA decoder for bitwise $(7, 4)$ Hamming code decoding is mapped onto AnLA array and simulated in Matlab. Finally, discussions on implementing other kinds of decoder on AnLA is given.

6.2.1 The Mathematical Development of ECC Decoding Problems

For the purpose of describing the ECC decoding on the AnLA array, we need to have a formal definition of the decoding problem. However, this is such a broad topic that we will not be able to cover completely. Therefore, here we will only give a truly brief collection of various definitions that are crucial for our demonstration. Readers interested in the mathematical aspects of the problem could refer to Appendix A, which provide a more detailed derivation.

We could define a ECC decoding task as an inference problem, as follows:

> At the transmitter, a codeword $t = \{t_1, t_2, ...t_N\}$ is selected from a linear (N, K) codeword set C. After it is transmitted over a noisy channel, the receiver would

receive a signal $y = \{y_1, y_2, ...y_N\}$. The decoding task is that for a given channel model P (y|t) and the received signal y, estimate the most probable transmitted signal t.

Depending on the nature of the codes, i.e., they could be generally decoded in two different ways:

1. The codeword decoding problem:

   Identify the most probable transmitted codeword t, given the received signal y. Mathematically,

   $$\max P\left(t|y\right), t \in C \tag{6.1}$$

2. The bitwise decoding problem:

   For each transmitted bit $t_n, n = 1, 2, ...N$, identify whether the bit was a "1" or a "0", given the received signal y. Mathematically,

   $$\max P\left(t_n|y\right), t_n \in \{0, 1\} \tag{6.2}$$

The codeword decoding problem is mathematically equivalent to solving a Maximum APosteriori (MAP) problem, where we seek to maximize the aposteriori probability distribution funciton $P\left(t|y\right)$. Moreover, based on the Bayes' Formula and some appropriate assumptions, as shown in Appendix A, we could obtain the relationship in equation (6.3).

$$P\left(t|y\right) \propto P\left(y|t\right) \tag{6.3}$$

Equation (6.3) could further simplify the codeword decoding problem (or the MAP problem) into a Maximum Likelihood (ML) problem, where we seek to maximize the likelihood function $P\left(y|t\right)$. Therefore, the codeword decoding problem could be described in mathematical form as shown in equation (6.4).

$$\max P\left(y|t\right), t \in C \tag{6.4}$$

.

Because the likelihood function could be derived apriori, exclusived based on the channel and signal property, the solution could be relatively easily obtained. This is certainly not the case in the MAP problem, in which aposteriori knowledge is required.

As to the bitwise decoding problem, the solution could be obtained by marginalizing the codeword probability distribution over all the other bits:

$$P\left(t_n|y\right) = \sum_{\{t_{n'}:n'\neq n\}} P\left(t|y\right) \tag{6.5}$$

Again exploiting the proportionality between the MAP probability $P\left(t|y\right)$ and the ML probability $P\left(y|t\right)$ in equation 6.3, we get:

$$P\left(t_n|y\right) \propto \sum_{\{t_{n'}:n'\neq n\}} P\left(y|t\right) \tag{6.6}$$

As a result, the decoding criteria could be specified, which is given in equation (6.7).

$$\frac{P\left(t_n=1|y\right)}{P\left(t_n=0|y\right)} = \frac{\sum_t P\left(y|t\right)\cdot I\left(t_n=1\right)}{\sum_t P\left(y|t\right)\cdot I\left(t_n=0\right)} = \begin{cases} \geq 1 \Rightarrow \left(t_n = \text{‘}1'\right) \\ < 1 \Rightarrow \left(t_n = \text{‘}0'\right) \end{cases} \tag{6.7}$$

6.2.2 Modeling the Decoder with the Message-passing Algorithm on a Trellis

Although we now have the mathematical representation of the two decoding problems. Directly calculating the target functions in both of the above two maximization problems would be computational prohibitive when the problems get to large scale. However, iterative message-passing algorithms exist, which could reduce the problems to more computationally feasible tasks. For example, the min-sum or max-product Viterbi algorithms are used for codeword decoding; while FBA is used for bitwise decoding problem.

As an demonstration of the modeling procedure, we will pick an examplary de-

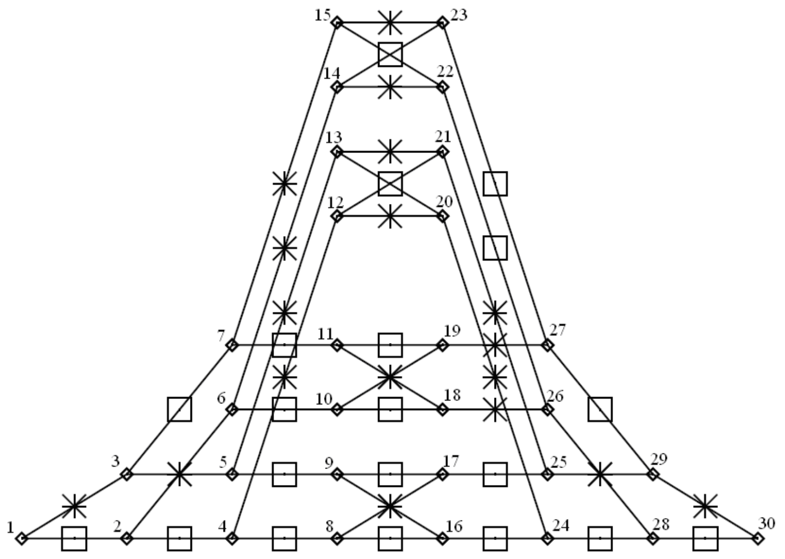


Figure 6-6: The trellis of the $(7, 4)$ Hamming code.

coding problem, *A* $(7, 4)$ *Hamming code decoded by the bitwise FBA*, and show key steps in the following development.

Firstly, the FBA could be formulated on a trellis for the $(7, 4)$ Hamming code. The trellis, as shown in Figure 6-6, is a graphical modeling tool and a variant of the Factor Graph. The message-passing algorithms mentioned above all works with the trellis representation [31, 41]. Because it could also be easily mapped onto the AnLA hardware, we choose it to be the bridge between the original mathematical problem to the final hardware realization.

The Forward-backward Algorithm could be derived based on the trellis. Conceptually, messages containing information about the received code signals are injected into the trellis at the nodes marked with $*$ and #, and then the messages propagate for two times independently, one from left to right (forward direction) and the other from right to left (backward direction). The bitwise decision could be retrieved by combining the forward messages with the backward messages at the bottom nodes.

More specifically, the seven vertical sectors of the trellis in Figure 6-6 correspond to the seven code bits. A edge marked with an $*$ (or a #) corresponds to a bit of 1 (or 0). Let i runs over nodes labeled from 1 through I, and $P(i)$ denotes the set of nodes that are parents of node i. The edge from node j to i in sector n is assigned of a weight equal to the bitwise likelihood function:

$$w_{ij} = P\left(y_n | t_n\right), \tag{6.8}$$

where t_n is the n^{th} transmitted bit; y_n is the received bit after going through the noisy channel. We define the forward messages α_i, propagating from node 1 back to node I, by:

$$\begin{aligned} &\alpha_1 = 1 \\ &\alpha_i = \sum_{j \in P(i)} w_{ij} \alpha_j, i = 2, 3, ..., I \end{aligned} \tag{6.9}$$

And the backward messages β_j, propagating from node I back to node 1, by:

$$\begin{aligned} &\beta_I = 1, \\ &\beta_j = \sum_{i:j \in P(i)} w_{ij} \beta_i, j = I-1, I-2, ..., 1 \end{aligned} \tag{6.10}$$

Finally, the merge of the forward and backward messages at each sector yields the bitwise code probability as follows:

$$r_n^{(t)} = \sum_{i,j:j \in P(i), t_{ij} = t} \alpha_j w_{ij} \beta_i, t = 0, 1 \tag{6.11}$$

And the posterior probability distribution is obtained after normalization:

$$P\left(t_n = t | y\right) = \frac{1}{Z} \cdot r_n^{(t)}, \tag{6.12}$$

where the normalization constant is: $Z = r_n^{(1)} + r_n^{(0)}$.

The AnLA array schematic for $(7, 4)$ Hamming code decoding is shown in Figure 6-7. As can be seen from the figure, the schematic is a straight-forward implementation of the FBA described above. The top left part of the AnLA cells performes forward message propagation while the bottom right part is in charge of backward message propagation. At the center, 7 UNEQ cells are used for combining the forward and backward messages. The decoded bits of the codeword could be collected from the outputs of these 7 UNEQ cells.

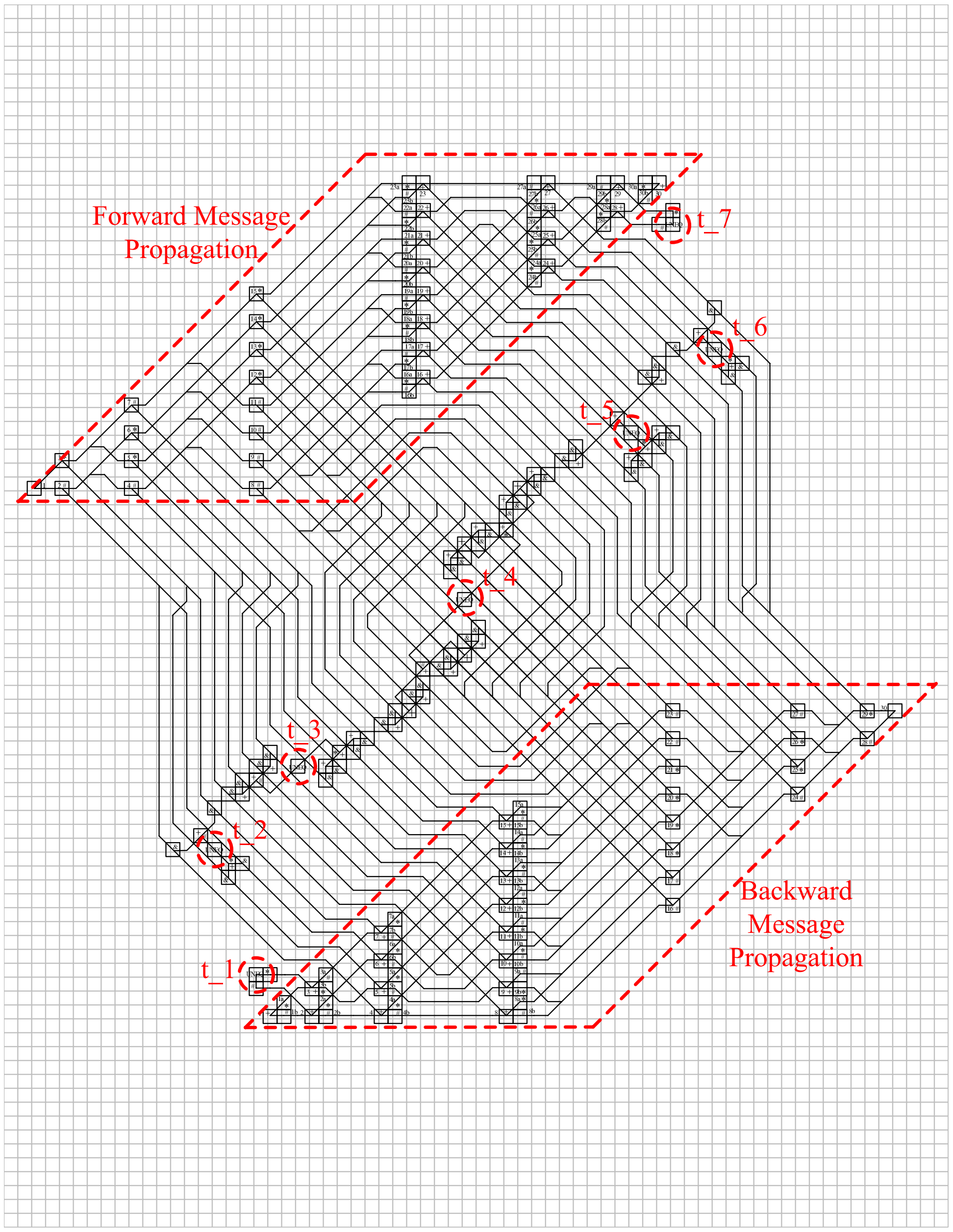


Figure 6-7: AnLA array schematic for $(7, 4)$ Hamming code decoding.

number	Likelihood		Posterior marginal		Output
n	$P(y_n \mid t_n = 1)$	$P(y_n \mid t_n = 0)$	$P(t_n = 1 \mid y)$	$P(t_n = 0 \mid y)$	digit
1	0.1	0.9	0.0615	0.9385	0
2	0.4	0.6	0.6738	0.3262	1
3	0.9	0.1	0.7460	0.2540	1
4	0.1	0.9	0.0615	0.9385	0
5	0.1	0.9	0.0615	0.9385	0
6	0.1	0.9	0.0615	0.9385	0
7	0.3	0.7	0.6594	0.3406	1

Table 6.1: An examplary $(7,4)$ Hamming code decoding process and result.

The intuitive explanation given here should be enough for us to understand the AnLA realization derived from the FBA that works on a trellis. Readers could get the complete information from Appendix A, which offers a formal definition of the trellis, the mathematical description of the FBA, as well as the mapping details from the trellis to the AnLA array.

6.2.3 Simulation Results of the AnLA Bitwise FBA Decoder

Based on the AnLA array schematic for $(7,4)$ Hamming code decoding, we did Matlab simulations to evaluate the performance of the decoder. Appendix A also provide complementary information on the simulation.

As an example, the following input received code is translated into bitwise likelihood probabilities and fed into the decoding array:

$$[P(y_1|t_1 = 1), P(y_2|t_2 = 1), ..., P(y_N|t_N = 1)] = [0.1, 0.4, 0.9, 0.1, 0.1, 0.1, 0.3] \tag{6.13}$$

$$[P(y_1|t_1 = 0), P(y_2|t_2 = 0), ..., P(y_N|t_N = 0)] = [0.9, 0.6, 0.1, 0.9, 0.9, 0.9, 0.7] \tag{6.14}$$

When the decoder finishes computation, the bitwise aposterior marginal probability can be read out of the array. After normalization and comparing of the output probability values, the decoded codeword is obtained, as shown in Table 6.1. The decoded result is in accordance with standard decoding algorithms.

As a summary, the AnLA array is always able to produce decoded result after 56 clock cycles. The correctness of our decoder is tested under a number of testing examples. The results matched correctly with nominal results derived from the standard FBA implementations.

6.2.4 More Discussion

1. Implementing Min-sum Viterbi Algorithm on AnLA array

To implement this algorithm on an AnLA array, more hardware for making decision and routing is needed. That is because at every step of the whole path through the trellis, the decision of which candidate path has the minimum log likelihood sum or maximum likelihood product must be made. Therefore, some sort of real-time decision and real-time reconfiguration has to be made. The AnLA array with self-modification capability would suffice to implement this algorithm.

2. Asynchronous AnLA array

In the present implementation of the forward-backward algorithm for bitwise ECC decoding, a lot of array cell resource is used as wiring for the message to propagate. Because the array is timed synchronously and every cell in the array updates its state synchronously, the synchronous wiring causes quite a large delay for the states to be propagated from a source to a destination. This indicates a waste of computation resource and hence room for improvement. The transition to asynchronous AnLA can be a potentially more efficient realization of the whole scheme. In the asynchronous version of AnLA cells, the cell will be in the "unready" state if the cell state is in an unusual state: $P_Z(1) = 0, P_Z(0) = 0$. Except that, the cell's normal "ready" state would still be represented by complementary analog value: $P_Z(1) = \alpha, P_Z(0) = 1-\alpha$. The cell in unready state would behave like a pump, waiting for its parents to become "ready" and then extracting its parents' states for its computation. The parents which provide their state for computation will be reset to "unready" state and in turn pumping their parents' information. In this way, an asynchronous updating rule is established and the whole network functions almost the same as its synchronous counterpart.

The revision from synchronous AnLA array to asynchronous might demand some extra digital control logic and a different scheme of initialization for the array to work properly. However, the gain would be large because the asynchronous updating can not only reduce delay, but also saves a lot of energy.

In conclusion, we first investigated a wide range of message-passing algorithms for ECC decoding. The corresponding message-passing formulations of the min-sum/max-product Viterbi Algorithm for MAP codeword decoding and the Forward-backward Algorithm for bitwise decoding are developed. To demonstrate that the forward-backward algorithm can be implemented on an AnLA array, the schematic of a $(7, 4)$ Hamming code decoder is designed and simulated in Matlab. Some further discussion concerning potential improvements of the AnLA array is given at the end of the discussion.

6.3 Image Processing

In this section, we are going to show a different exploitation of the analog states in the AnLA array, for the application of grey-scale image processing. The normalized grey level of each pixel of the image is represented by the analog state $P_Z(1)$ of the corresponding cell in the AnLA array. With each cell representing a grey-level pixel, the AnLA array becomes a programmable image processor. Three different imaging processing examples are presented here.

6.3.1 Image Segmentation

Segmentation is achieved by the soft EQ operation on each cell. The soft EQ operation on each pixel in this application can be viewed as a computation for depicting the *similarity* between the pixel itself and the average of its four neighbors. In our particular setting, each cell's inputs X and Y are selected as in equation (6.15), in

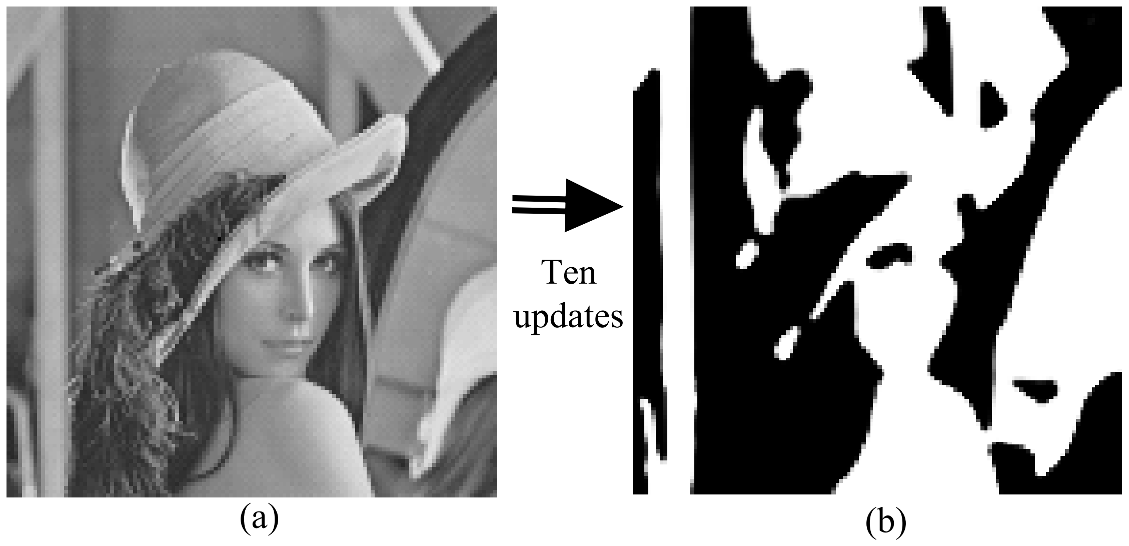


Figure 6-8: Segmentation effect after AnLA processing: (a) Original image; (b) Segmented image after 10 updates.

which A, B, C, and D are rectangular neighbors of the center cell Z.

$$\begin{aligned} P_X(1) &= P_{Z\_(t-1)}(1) \\ P_Y(1) &= \left[P_A(1) + P_B(1) + P_C(1) + P_D(1)\right]/4 \end{aligned} \tag{6.15}$$

In each time step, every cell does soft EQ operation and updates its state. MATLAB simulation shows an excellent segmentation / low-pass filtering effect of the "Lena" test image after 10 time steps (Figure 6-8).

Another way to understand this processing technique is that soft EQ tends to have a low-pass property on its two input signals, which would smooth out the picture and aggregate pixel intensity in a more compact manner, leading to the segmentation effect.

6.3.2 Image Edge Enhancement

A second and similar application is image edge enhancement / high-pass filtering effect achieved by soft UNEQ operation. The soft UNEQ operation takes in the same input configurations as soft EQ, i.e., equation (**??**). But UNEQ computation on each pixel depicts the *difference* between the pixel itself and the average of its four neighbors.

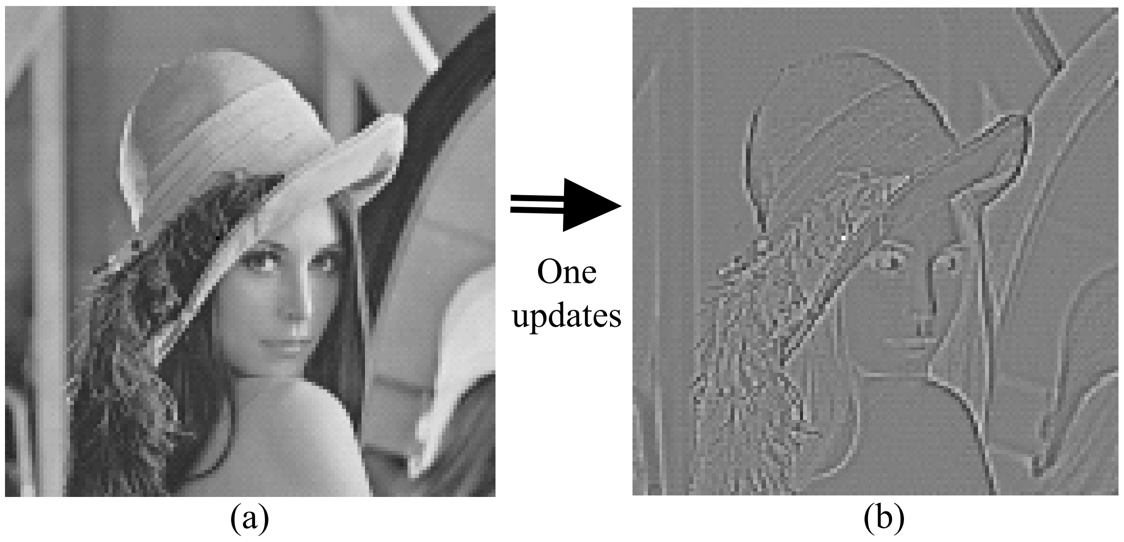


Figure 6-9: Edge enhancement effect after AnLA processing: (a) Original image; (b) Enhanced image after only 1 updates.

More importantly, in contrast to the iterative updates needed in image segmentation, the edge enhancement effect appears right after only ONE array update, owing to the complete parallel computation on the AnLA array. MATLAB simulation shows a good edge enhancement effect in Figure 6-9, which is fulfilled in only one time step.

6.3.3 Motion Detection

The motion detection application is an extension of the image edge enhancement technique. Since motion detection is usually achieved by comparing consecutive video frames and highlighting the difference, we could apply soft UNEQ computation on the two video frames to disclose the difference.

Therefore, the configuration of the AnLA array becomes: each AnLA cell corresponds to the two pixels from the two video frames of the same coordinate; the input X and Y are the pixel values from the two pixels, as shown in equation (6.16).

$$\begin{aligned} P_X(1) &= P_{Z\_frame1}(1) \\ P_Y(1) &= P_{Z\_frame2}(1) \end{aligned} \tag{6.16}$$

After one array update, the analog states stored in AnLA would be the resultant

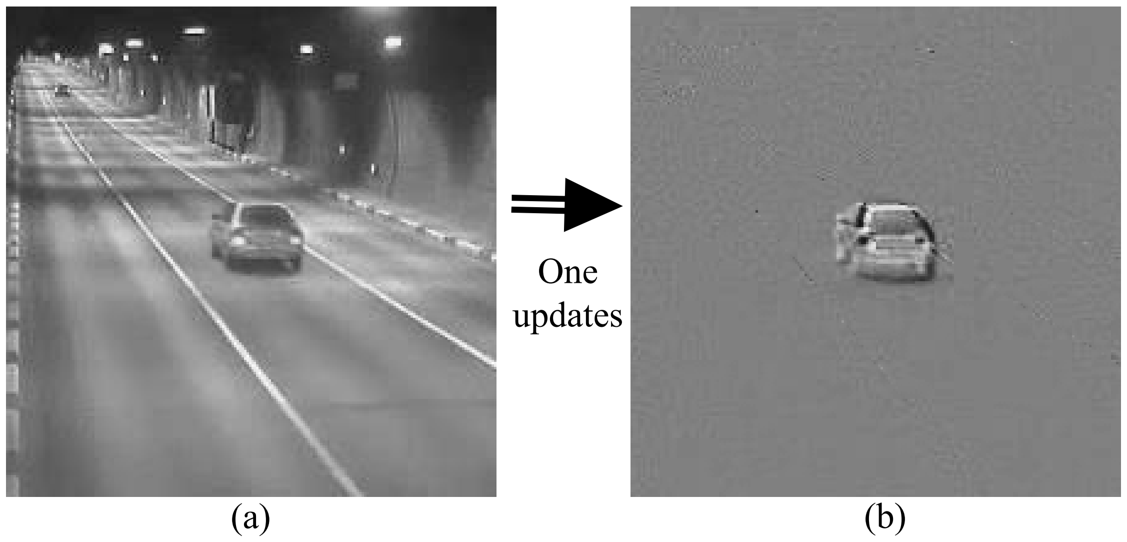


Figure 6-10: Motion detection effect after AnLA processing: (a) One frame from a traffic video; (b) Motion detection effect.

motion detection profile. We used a traffic video stream as a testbed for our motion detection technique. Figure 6-10 (a) shows the motion detection result of the traffic video (Figure 6-10 (b)).

The image processing applications implemented in AnLA could be made very low-power and high-speed, because not only it exploits the analog computation, but also it fully enjoys the distributed processing capability offered by AnLA. Such kind of image processor could be useful in many real-time image front-ends that require low-power dissipation. For example, we could integrate the AnLA image processor into the image sensor array chip of the wireless capsule Gastrointestinal (GI) endoscopy [35] to facilitate image pre-processing and image compression before streaming the recorded video to the wireless telemetry link, thus reducing the transmission workload.

6.4 More Potential Applications

The AnLA circuits presented here promise great potential in many research fields. Firstly, this architecture offers a new approach to a better neuromorphic design.

Secondly, more complex image processing algorithms can be developed and used in different biomedical applications, given this highly programmable hardware. Thirdly, the AnLA architecture suggests a new way to realize Software Defined Radio (SDR). Instead of first digitizing the RF signal and then processing the signal digitally, AnLA circuits can directly obtain the baseband signal through analog computation, only digitizing at the output. Finally, AnLA can implement decoders for other types of ECC codes decodable in similar message-passing algorithms, such as turbo codes and low-density parity-check codes. Overall, an AnLA receiver is suitable for low-power wireless applications, and in general, the AnLA architecture is promising as a versatile platform for fast algorithm/application development.

Chapter 7

Conclusion

In this work, we have successfully demonstrated the potential of the Logic Automata model by implementing the Asynchronous Logic Automata and the Analog Logic Automata with transistor-level circuits.

The ALA cell library and the “pick-and-place” design flow is developed in a 90nm CMOS process. The cell library provides a universal computation architecture in which the parallelism is extremely fine-grained, down to the bit level. The bitwise pipelined structure, together with the asynchronous communication interface not only offers a high computation throughput, but promises the maximum design scalability because there are no global design constraints such as the clock and its distribution networks. In our ALA design work flow, each autonomous cell within an ALA design could be generated by selecting corresponding cell type and interconnection directions from the cell library and the interconnection mask layer. After cells are placed on a grid, they align themselves automatically and a working ALA circuitry ensues. The work flow maps directly the description of computer programs into circuit realizations, which could dramatically simplify the IC development. Example ALA circuits are assembled with the ALA design flow and tested. These ALA circuits can generally operate at a throughput of around $1GHz$ and compute energy efficiently.

The next steps for the ALA effort include both designing more efficient ALA cell library and the Reconfigurable ALA (RALA) development. Firstly, it is possible to improve the C-element design by switching to other logic style implementations. The current C-element implementation is based on the dynamic logic structure with a weak feedback keeper at the output to statically hold the circuit state. But a C-element is also possible to be designed with a R-S latch and a few logic, which do not have much short current in state transitions. This could potentially lead to less energy consumption per computation. Secondly, a different way to reduce the hardware overhead of implementing the handshake protocol is to use a simpler communication interface. We can continue to explore the self-timed circuitry as the asynchronous interface between ALA cells, which is more hardware efficient than the PCFB-based handshaking logic used in the current ALA cell library. Thirdly, the RALA model, as an in-band reconfigurable, univeral computation system, has aroused much interest as the true alternative to the von Neumann architecture computers. To develop transistor-level realizations for RALA, reprogrammable capability is needed. As we now have a good understanding of the asynchronous finite state machine that is governing the ALA cell operation, we could try to manipulate and disassemble the asynchronous Finite State Machine (aFSM) into parts so that the parts could be re-assembled as working aFSM's for RALA cells with varying behaviors in a programmable fashion.

As another variant of the Logic Automata family, the AnLA model is demonstrated by a prototyping chip in AMI $0.5\mu m$ CMOS technology. The AnLA computing structure relaxes digital processing into its analog counterpart where the Analog Logic principles is the primary computing mechanism. The analog computations are efficient and are made both distributed and reconfigurable in the Logic Automata framework. Therefore, the reconfigurable AnLA implementing message-passing algorithms have the potential to solve many kinds of statistical inference and signal processing problems with much more efficiency than their digital counterparts. In this work, we have already illustrated the unique advantage of AnLA in pseudo-random signal synchronization (the NLL loop), ECC decoding and statistical image processing. But many more applications could be found on this versatile platform.

The AnLA circuits are currently programmable, but at a cost of many-bit configuration storage overhead. This can be improved by cutting the cell function set to the extent that enough functionality is maintained while the configuration bit storage is acceptable. A second further development would be to make AnLA circuits asynchronous. As explained in the ALA model, asynchrony is essential to ensure absolute extensibility for the model. The analog state storage could be controlled by additional logic implementing a handshake protocol to become asynchronous state storage and transfer. Application-wise, the AnLA architecture could be potentially useful in many fields that enjoy analog computation or statistical signal processing. For example, the NLL circuitry as a receiver for the pseudorandom signal could be embedded into the front end of a wide-band communication radio, with the programmability of the AnLA NLL receiver indicating a software defined RF radio. Similarly, the programmable AnLA ECC decoder can be augmented to implement more complex decoding systems and the image processor is suitable for performing image pre-processing in some low-power imaging systems.

Overall, this work serves as an initial investigation on the Logic Automata model, from the hardware perspective. It could be a starting point for more future hardware development to uncover the huge computing power of the Logic Automata.

Appendix A

Detailed Modeling of Error Correcting Code Decoding on Analog Logic Automata

This chapter gives mathematical interpretation of the ECC decoding problems. It also describes the mapping from the message-passing algorithms that solve decoding problems, to the corresponding AnLA realization.

The two categories of decoding problems - the codeword decoding problems and the bitwise decoding problems - are firstly mathematically formulated into a maximization problem. Message-passing algorithms running on a trellis [31, 41] are obtained afterwards. And then an exemplary Forward-backward Algorithm (FBA) decoder for bitwise $(7, 4)$ Hamming code decoding is demonstrated, with the mapping procedure described in detail based on this example.

A.1 Definitions and Assumptions for the ECC Decoding Problem

A linear (N, K) code is most commonly represented in terms of its generator matrix, G, and its parity-check matrix, H. The examplary $(7, 4)$ Hamming code that we will

$$\boldsymbol{G}=\begin{bmatrix}1&0&0&0&1&0&1\\0&1&0&0&1&1&0\\0&0&1&0&1&1&1\\0&0&0&1&0&1&1\end{bmatrix} \qquad \boldsymbol{H}=\begin{bmatrix}1&0&1\\1&1&0\\1&1&1\\0&1&1\\1&0&0\\0&1&0\\0&0&1\end{bmatrix}$$

(a) (b)

Figure A-1: Matrix representation of a $(7, 4)$ Hamming code: (a)the generator matrix G; (b) the parity-check matrix H.

be using throughout our derivation could be describe as shown in the following Figure A-1:

We then give a definition of the linear ECC decoding task as an inference problem [41]:

> At the transmitter, a codeword $t = \{t_1, t_2, ...t_N\}$ is selected from a linear (N, K) codeword set C. After it is transmitted over a noisy channel, the receiver would receive a signal $y = \{y_1, y_2, ...y_N\}$. The decoding task is that for a given channel model P (y|t) and the received signal y, estimate the most probable transmitted signal t.

Secondly, we introduce the two different ways of decoding:

1. The codeword decoding problem:

   Identify the most probable transmitted codeword t, given the received signal y. In other words:

$$\max P\left(t|y\right), t \in C \tag{A.1}$$

2. The bitwise decoding problem:

   For each transmitted bit $t_n, n = 1, 2, ...N$, identify whether the bit was a "1" or a "0", given the received signal y.

In other words:

$$\max P\left(t_n|y\right), t_n \in \{0,1\} \tag{A.2}$$

Very often in practice, we have the following assumptions:

- The channel model is a memoryless white Gaussian channel.

  Assuming the transmitted signal bit t_n took value of $+x$ for a digital bit of “1”; and $-x$ for a digital bit of “0”. The received signal bit y_n is a corrupted version of the transmitted signal t_n. If the additive white Gaussian corruption has a standard deviation of σ, the conditional distribution of y_n can be written as:

$$P\left(y_n|t_n=1\right) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{\left(y_n - x\right)^2}{2\sigma^2}\right) \tag{A.3}$$

$$P\left(y_n|t_n=0\right) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{\left(y_n + x\right)^2}{2\sigma^2}\right) \tag{A.4}$$

  In the context of decoding, most of the time we are interested in the relative value of the probability terms because we can normalize probability distribution anyway. So here we are only concerned with the ratio of the two likelihood functions, which is sufficient in providing complete information of the channel model, as in equation A.5.

$$\frac{P\left(y_n|t_n=1\right)}{P\left(y_n|t_n=0\right)} = \exp\left(\frac{2x \cdot y_n}{\sigma^2}\right) \tag{A.5}$$

  Because the channel noise characteristics σ and transmitted symbol waveform x are known before-hand, we are guaranteed to obtain the likelihood function every time we receive a signal y_n from the channel, by doing a “dot-product” computation on the transmitted symbol, x, and the received symbol, y_n, as shown in equation A.5.

- Every codeword has equal possibility of being transmitted.

  This assumption leads to the simplification that the transmitted codeword distribution $P\left(t\right)$ (“the prior”) is uniform over the whole codeword set C.

A.2 Mathematical Formulation of the Codeword Decoding and Bitwise Decoding Problems

With the above preparation, we are now ready to discuss separately about the two categories of problems, following distinct but related formulations.

A.2.1 The Codeword Decoding Problem

This problem is also called the Maximum APosteriori (MAP) decoding. Because it is seeking to maximize the posteriori probability P (t|y). We will show that the MAP problem could turn into Maximum Likelihood (ML) problem under the assumptions listed previously.

According to Bayes' Formula, we could link MAP function with ML function in the following equation:

$$P\left(t|y\right) = \frac{P\left(y|t\right)P\left(t\right)}{P\left(y\right)} \tag{A.6}$$

In equation (A.6), the prior, $P\left(t\right)$, is uniform under our asumption; the normalization factor in the denominator is also constant, in that it is the marginalization of the numerator over the whole codeword set C:

$$P\left(y\right) = \sum_{t\in C} P\left(y,t\right) = \sum_{t\in C} P\left(y|t\right)P\left(t\right) \tag{A.7}$$

Therefore, we could ignore the prior and the normalization factor, simplifying the MAP problem into a ML decoding problem:

$$P\left(t|y\right) \propto P\left(y|t\right) \tag{A.8}$$

The codeword decoding problem now becomes:

$$\max P\left(y|t\right), t \in C \tag{A.9}$$

.

A.2.2 The Bitwise Decoding Problem

The solution for bitwise decoding problem is obtained by marginalizing the codeword probability distribution over the other bits:

$$P\left(t_n|y\right) = \sum_{\{t_{n'}:n'\neq n\}} P\left(t|y\right) \propto \sum_{\{t_{n'}:n'\neq n\}} P\left(y|t\right) \tag{A.10}$$

To be more specific, the probability distribution function for bit t_n is:

$$P\left(t_n = 1|y\right) = \sum_t P\left(t|y\right) \cdot I\left(t_n = 1\right) \tag{A.11}$$

$$P\left(t_n = 0|y\right) = \sum_t P\left(t|y\right) \cdot I\left(t_n = 0\right) \tag{A.12}$$

In which the function $I\left(\bullet\right)$ is the indication function.

Consequently, the bitwise decoding becomes equivalent to the following comparison:

$$\frac{P\left(t_n = 1|y\right)}{P\left(t_n = 0|y\right)} = \frac{\sum_t P\left(y|t\right) \cdot I\left(t_n = 1\right)}{\sum_t P\left(y|t\right) \cdot I\left(t_n = 0\right)} = \begin{cases} \geq 1 \Rightarrow \ (t_n = \text{‘}1'\text{)} \\ < 1 \Rightarrow \ (t_n = \text{‘}0'\text{)} \end{cases} \tag{A.13}$$

A.3 The Message-passing Algorithms for ECC Decoding

Up to now, we already have the mathematical representation of the two decoding problems. But directly calculating the target functions in both of the above two maximization problems would be computationally prohibitive when the problems get to large scale. Actually, MAP codeword decoding for a general linear code is known to be NP-complete, and marginalizing bitwise probability takes exponential time.

It is exactly because of the need to reduce computation efforts needed, that message-passing algorithms come into play. These iterative algorithms could reduce problems to more computational feasible tasks. For example, the min-sum or max-product Viterbi algorithms are used for codeword decoding. And FBA is used for bitwise decoding problem. This section will discuss the algorithms working under the trellis representation.

A.3.1 The Trellis

In order to describe the message-passing algorithms, we need a graphical representation of the problem. A useful representation method, among others, is called trellis [31, 41]. It is a variant of the Factor Graph and many ECC decoding algorithms could be generalized to operate on a trellis.

- Definition of Trellis

  A trellis is a graph consisting of nodes (also known as states or vertices) and edges, in which the nodes are grouped into vertical slices called times, and the times are ordered such that each edge connects a node in one time to a node in a neighboring time. Every edge is labeled with a symbol. The leftmost and rightmost states contain only one node. Apart from these two extreme nodes, all nodes in the trellis have at least one edge connecting leftwards and at least one connecting rightwards.

A trellis is called a linear trellis if the code it defines is a linear code. For the purpose of discussion here, we will only consider linear binary trellises. And we list some basic conventions and properties of the trellis here:

1. The leftmost time is numbered time 0 and the rightmost time N.

2. The leftmost state is numbered state 0 and the rightmost state I, where I is the total number of states (vertices) in the trellis.

3. For a binary trellis, each symbol on a edge is representing either bit 1 or bit 0.

4. The width of the trellis at a given time is the number of nodes in that time.

5. For any linear code the minimal trellis is the one that has the smallest number of nodes. In a minimal trellis, each node has at most two edges entering it and at most two edges leaving it. All nodes in a time have the same left degree as

each other and they have the same right degree as each other. The width is always a power of two.

6. A minimal trellis for a linear (N, K) code cannot have a width greater than 2K since every node has at least one valid codeword through it, and there are only 2K codewords. Furthermore, the minimal trellis's width is everywhere less than 2(N-K).

7. K is the number of times a binary branch point is encountered as the trellis is traversed from left to right or from right to left.

A (N, K) linear code has block length of N and could be modeled by a trellis with $(N + 1)$ times: A codeword is obtained by taking a path that crosses the trellis from left to right and reading out the symbols on the edges that are traversed. Each valid path through the trellis defines a codeword. The n^th bit of the codeword is emitted as we move from time $(n - 1)$ to time n.

Following the above rules, we could construct the trellis in Figure 6-6, which corresponds to the $(7, 4)$ Hamming code as defined in matrix form previously (Figure A-1).

A.3.2 Decoding Algorithms on Trellises

We now introduce the algorithms on trellises for ECC decoding.

1. The min-sum / max-product algorithm for codeword decoding problem

The MAP codeword decoding problem can be solved using the min-sum algorithm. Each edge of "time n" on the trellis is associated with the bitwise log likelihood cost $-\log\left(P\left(y_n|t_n\right)\right)$, where t_n is the transmitted bit associated with that edge, and y_n is the received symbol. Since each codeword of the code corresponds to a path across the trellis, the cost of a journey is the sum of the costs of its constituent steps, i.e., the bitwise log likelihood. By adding the bitwise log likelihood together, we get the log likelihood of a codeword. Therefore, the path with minimum cost corresponds to the codeword with maximum log likelihood, i.e., the most probable transmitted codeword.

The max-product algorithm is just an alternative representation of the min-sum algorithm, in which each edge is directly assigned the bitwise likelihood $P(y_n|t_n)$, and the path cost is obtained by multiplying together the costs of edges that are traversed by that path. The total cost is exactly the codeword likelihood and the path that maximizes the cost is the resultant decoded code in MAP sense. These algorithms are also known as the Viterbi algorithm [76].

2. The forward-backward algorithm for bitwise decoding problem

The bitwise decoding problem can be solved by forward-backward algorithm, which is an application of sum-product algorithm. To solve the bitwise decoding problem, we assign the bitwise likelihood function $P(y_n|t_n)$ to each edge on the trellis just as what we did in max-product Viterbi algorithm. Then the messages passed through the trellis define "the probability of the data up to the current point". Unlike min-sum or max-product algorithm, we do not discard any path traversing the trellis, but add the product of the message accumulatively in each step. The messages propagate in forward and backward direction independently and the final result is derived from the combined result of forward and backward message. The detailed description of the algorithm is give below.

Let i runs over nodes/states, $i = 0$ be the label for the start state, $P(i)$ denotes the set of states that are parents of state i, and w_{ij} be the likelihood associated with the edge from node j to node i. We define the forward-pass messages α_i by

$$\alpha_0 = 1 \tag{A.14}$$

$$\alpha_i = \sum_{j \in P(i)} w_{ij}\alpha_j \tag{A.15}$$

These messages are called forward-pass messages and can be computed sequentially from left to right. The message α_I computed at the end node of the trellis is proportional to the marginal probability of the data.

Similarly, we define a second set of backward-pass messages β_I (starts from node I).

$$\beta_I = 1 \tag{A.16}$$

$$\beta_j = \sum_{i:j \in P(i)} w_{ij} \beta_i \tag{A.17}$$

These messages are called backward-pass messages and can be computed sequentially in a backward pass from right to left.

Finally, the merge of forward and backward messages yields the bitwise code probability. The probability of the n^th bit being a 1 or 0 is obtained by doing two summations over products of forward and backward messages. Let i runs over nodes at time n and j runs over nodes at time $(n-1)$ (j is parent of i), and let t_{ij} be the value of t_n associated with the trellis edge from node j to node i. For $t = 0$ and $t = 1$, we compute:

$$r_n^{(t)} = \sum_{i,j:j \in P(i), t_{ij}=t} \alpha_j w_{ij} \beta_i \tag{A.18}$$

After computing $r_n^{(1)}$ and $r_n^{(0)}$, the posterior probability distribution is obtained after normalization:

$$P\left(t_n = t | y\right) = \frac{1}{Z} \cdot r_n^{(t)} \tag{A.19}$$

Where the normalization constant $Z = r_n^{(1)} + r_n^{(0)}$. As a sanity check, Z should be equal to the final forward message α_I computed earlier.

A.4 Implementation of Forward-backward Algorithm on AnLA

The algorithm can be implemented on an AnLA array with both rectangular and diagonal neighbor connections. As a proof of concept, a schematic for decoding the $(7, 4)$ Hamming Code introduced previously is developed and simulated in Matlab. The schematic is shown in Figure 6-7.

The up left part of the schematic works as the forward-message passing (α message), while the bottom right part is doing backward-message passing (β message). After the forward and backward messages finish propagating, the resultant messages α_i and β_i, $i \in \{1, 2, ..., I\}$, are then merged in the center part of the schematic to give the final decoded bits.

Implementation details concerning the notations used in Figure 6-7 are given below:

The AnLA cell can perform many different operations according to configuration. Below is the detailed description of how to implement the desired operation on the AnLA array for the Forward-backward Algorithm.

- Multiplication of two input probabilities — Multiply Function

  Cell icon: &

  Gate Function: Soft AND

  Input X: Multiplicand 1, $P_X(1) = P_1$

  Input Y: Multiplicand 2, $P_Y(1) = P_2$

  Output: $P_{out} = P_Z(1) = P_X(1) \cdot P_Y(1) = P_1 \cdot P_2$

- Multiplication with bitwise likelihood — Multiply $*$ Function

  Cell icon: $*$

  Gate Function: Soft AND

  Input X: Bitwise Likelihood from EXT, $P_X(1) = P(y_n|t_n = 1)$

  Input Y: Input Probability, $P_Y(1) = P_{in}$

  Output: $P_{out} = P_Z(1) = P_X(1) \cdot P_Y(1) = P_{in} \cdot P(y_n|t_n = 1)$

- Multiplication with bitwise likelihood — Multiply # Function

  Cell icon: #

  Gate Function: Soft AND

  Input X: Bitwise Likelihood from EXT, $P_X(1) = P(y_n|t_n = 0)$

  Input Y: Input Probability, $P_Y(1) = P_{in}$

Output: $P_{out} = P_Z(1) = P_X(1) \cdot P_Y(1) = P_{in} \cdot P(y_n|t_n = 0)$

- Addition of two input probabilities — Add Function

  Cell icon: +

  Gate Function: Soft X

  Input X: Addend 1, Addend 2, $P_X(1) = (P_1 + P_2)/2$

  Input Y: Not care

  Output: $P_{out} = P_Z(1) = P_X(1) = (P_1 + P_2)/2$

- Propagation of probability — Wire Function

  Cell icon: \ or /

  Gate Function: Soft X

  Input X: Input Probability, $P_X(1) = P_{in}$

  Input Y: Not care

  Output: $P_{out} = P_Z(1) = P_X(1) = P_{in}$

- Comparison of two probabilities — Bitwise Decision Function

  Cell icon: UNEQ

  Gate Function: Soft UNEQ

  Input X: Probability of "bit 1" (output of cell "*"), $P_X(1) = P(t_n = 1|y)$

  Input Y: Probability of "bit 0" (output of cell "#"), $P_Y(1) = P(t_n = 0|y)$

  Output:

$$P_Z(1) = P_X(1) \cdot P_Y(0)$$
$$P_Z(0) = P_X(0) \cdot P_Y(1)$$

$$P_Z(1) > P_Z(0) \Leftrightarrow P_X(1) > P_Y(1) \Leftrightarrow t_n = 1$$
$$P_Z(1) < P_Z(0) \Leftrightarrow P_X(1) < P_Y(1) \Leftrightarrow t_n = 0$$

With the above gate function, it is sufficient to perform forward-backward algorithm. One thing worth noting is that the “Add Function” would automatically average the two input probability, which gives an extra gain of 1/2 in message propagation, as compared with the original algorithm. However, since we are only concerned with the final probability ratio, and the additional gain factors are applied equally to both probability of “bit 1” and “bit 0”, this effect does not affect the correct operation of the algorithm.

A.5 Matlab Simulation Details

The schematic is input into Matlab using programmed command. The script file “layout2netlist.m” generates netlist of the decoding array according to the schematic drawn previously. The netlist is subsequently read into simulation file “netlist2sim.m” and the program simulate the whole array according to the configuration specified in the netlist. Another script file “netlist2layout.m” is also developed for generating the schematic view in Matlab without actually simulating the schematic. The designed schematic is able to give decoding results after 56 clock cycle. Figure A-2 shows the Matlab generated schematic and the array states after completing the decoding computation of a test input.

The description of the graphical representation in Matlab is give below:

- The grayscale indicates the state of an AnLA cell. As the intensity changes from black to white, the value of the soft state changes from 0 to 1.

- The connection of each cell is indicated by the colored lines and triangles. The blue lines/triangles are associated with X inputs and the yellow ones are associated with Y inputs. The lines indicate the input orientation (N, E, W, S, NW, NE, SW, SE) while the existence of the triangles indicates either X input from EXT (external input) or Y input from Z (input from the cell’s last state).

- Most of the unlabeled cells are performing wire function. Other cells have some kind of labeling: the Multiply Function cell is labeled with a letter “m”, the

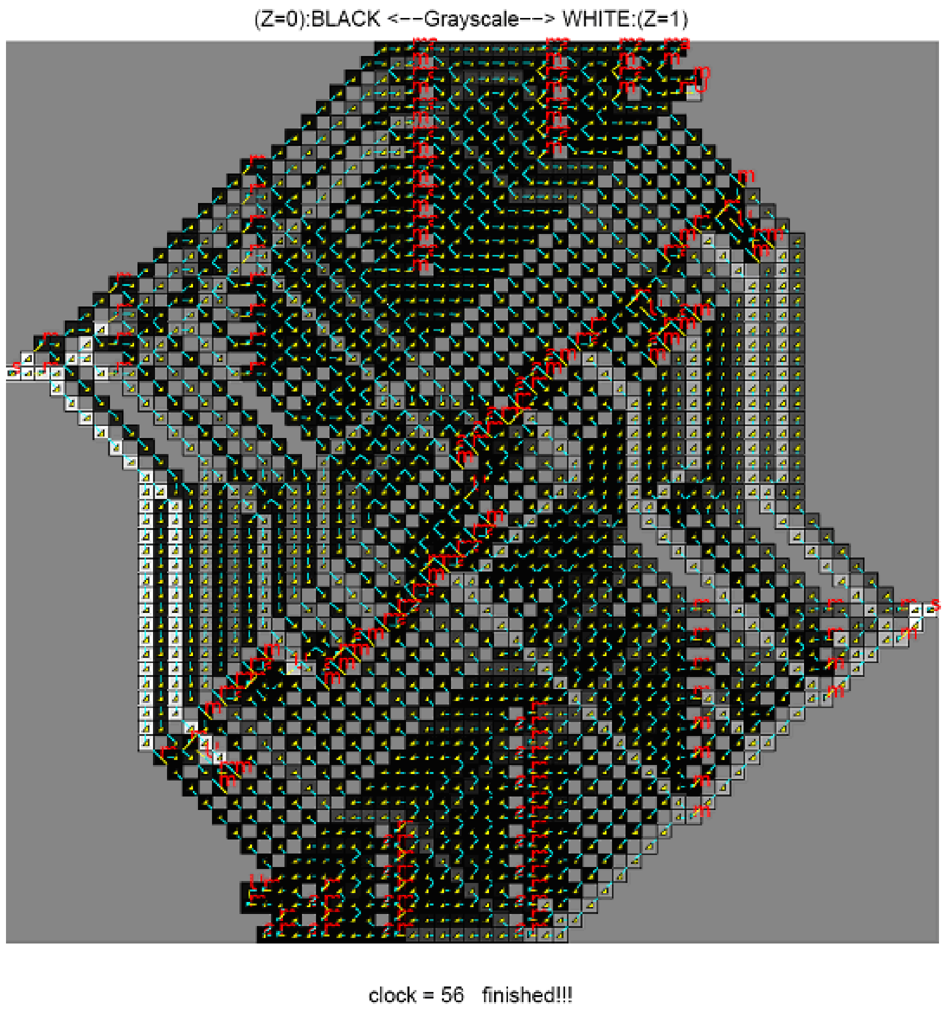


Figure A-2: The AnLA array states after completing the decoding computation for (7, 4) Hamming code decoding.

Add Function cell is labeled with a letter “a”, and the Bit Decision Function cell is labeled with a letter “U”.

Bibliography

[1] A.P. , S. Sheng, and R.W. Brodersen. Low-power cmos digital design. *Solid-State Circuits, IEEE Journal of*, 27(4):473–484, Apr 1992.

[2] R.E. Banks. *Information Processing and Transmission in Cellular Automata.* PhD thesis, Massachusetts Institute of Technology, 1971.

[3] J. Bohorquez, W. Sanchez, L. Turicchia, and R. Sarpeshkar. An integrated-circuit switched-capacitor model and implementation of the heart. *Proceedings of the First International Symposium on Applied Sciences on Biomedical and Communication Technologies*, pages 1–5, Oct 2008.

[4] W.F. Brinkman. The transistor: 50 glorious years and where we are going. pages 22–26, 425, Feb 1997.

[5] A. Burks. *Essays on Cellular Automata.* University of Illinois Press, 1970.

[6] K. Chen, F. Green, and N. Gershenfeld. Asynchronous logic automata asic design. to be submitted (2009).

[7] K. Chen, J. Leu, and N. Gershenfeld. Analog logic automata. *Biomedical Circuits and Systems Conference, 2008. BioCAS 2008. IEEE*, pages 189–192, Nov. 2008.

[8] D. Dalrymple. Asynchronous logic automata. Master's thesis, Massachusetts Institute of Technology, Jun 2008.

[9] D. Dalrymple, E. Demaine, and N. Gershenfeld. Reconfigurable asynchronous logic automata. to be submitted (2009).

[10] D.A. Dalrymple, N. Gershenfeld, and K. Chen. Asynchronous logic automata. *Proceedings of the AUTOMATA 2008 (14th International Workshop on Cellular Automata)*, pages 313–322, Jun 2008.

[11] A. Davis and S.M. Nowick. An introduction to asynchronous circuit design. *Technical Report UUCS-97-013, Computer Science Department, University of Utah*, Sep 1997.

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

[13] D. Fang. Width-adaptive and non-uniform access asynchronous register files. Master's thesis, Cornell University, Jan 2004.

[14] D. Fang and R. Manohar. Non-uniform access asynchronous register files. pages 78–85, April 2004.

[15] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient belief propagation for early vision. In *In CVPR*, pages 261–268, 2004.

[16] William T. Freeman, Egon C. Pasztor, and Owen T. Carmichael Y. Learning low-level vision. *International Journal of Computer Vision*, 40:2000, 2000.

[17] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4:247–253, 1996.

[18] G. Geannopoulos and X. Dai. An adaptive digital deskewing circuit for clock distribution networks. pages 400–401, Feb 1998.

[19] N. Gershenfeld. Programming bits and atoms. to be published.

[20] N. Gershenfeld. *The Nature of Mathematical Modeling*. Cambridge University Press, 1999.

[21] B. Gilbert. A precise four-quadrant multiplier with subnanosecond response. *Solid-State Circuits, IEEE Journal of*, 3(4):365–373, Dec 1968.

[22] B. Gilbert. Translinear circuits: a proposed classification. *Electronics Letters*, 11(1):14–16, 9 1975.

[23] E. Goetting, D. Schultz, D. Parlour, S. Frake, R. Carpenter, C. Abellera, B. Leone, D. Marquez, M. Palczewski, E. Wolsheimer, M. Hart, K. Look, M. Voogel, G. West, V. Tong, A. Chang, D. Chung, W. Hsieh, L. Farrell, and W. Carter. A sea-of-gates fpga. pages 110–111, 346, Feb 1995.

[24] S. Greenwald, L. Molinero, and N. Gershenfeld. Numerical methods with asynchronous logic automata. to be submitted (2009).

[25] J. Hagenauer and M. Winklhofer. The analog decoder. *Proc. IEEE Int. Symp. on Information Theory*, page 145, Aug 1998.

[26] T.S. Hall, C.M. Twigg, J.D. Gray, P. Hasler, and D.V. Anderson. Large-scale field-programmable analog arrays for analog signal processing. *Circuits and Systems I: Regular Papers, IEEE Transactions on [Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on]*, 52(11):2298–2307, Nov 2005.

[27] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[28] D. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley Professional, second edition, Apr 1998.

[29] P. Kogge and H. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22:783–791, 1973.

[30] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, 47(2):498–519, Feb 2001.

[31] F.R. Kschischang and V. Sorokine. On the trellis structure of block codes. *Information Theory, IEEE Transactions on*, 41(6):1924–1937, Nov 1995.

[32] H. T. Kung and C. E. Leiserson. *Algorithms for VLSI processor arrays; in C. Mead, L. Conway: Introduction to VLSI Systems.* Addison-Wesley, 1979.

[33] S. Kung. Vlsi array processors. *ASSP Magazine, IEEE*, 2(3):4–22, Jul 1985.

[34] P. Lalwaney, L. Zenou, A. Ganz, and I. Koren. Optical interconnects for multiprocessors cost performance trade-offs. pages 278–285, Oct 1992.

[35] Meng-Chun Lin, Lan-Rong Dung, and Ping-Kuo Weng. An ultra-low-power image compressor for capsule endoscope. *BioMedical Engineering OnLine*, 5(1):14, 2006.

[36] H.-A. Loeliger. An introduction to factor graphs. *Signal Processing Magazine, IEEE*, 21(1):28–41, Jan. 2004.

[37] H.-A. Loeliger, F. Lustenberger, M. Helfenstein, and F. Tarkoy. Probability propagation and decoding in analog vlsi. *Information Theory, IEEE Transactions on*, 47(2):837–843, Feb 2001.

[38] F. Lustenberger. *On the Design of Analog VLSI Iterative Decoders.* PhD thesis, Swiss Federal Institute of Technology, Nov 2000.

[39] F. Mace, F.-X. Standaert, and J.-J. Quisquater. Asic implementations of the block cipher sea for constrained applications. *Proceedings of RFID Security 2007*, 2007.

[40] F. Mace, F.-X. Standaert, and J.-J. Quisquater. Fpga implementation(s) of a scalable encryption algorithm. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(2):212–216, Feb. 2008.

[41] David MacKay. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003.

[42] R. Manohar. Reconfigurable asynchronous logic. pages 13–20, Sept. 2006.

[43] N. Margolus. An embedded dram architecture for large-scale spatial-lattice computations. pages 149–160, 2000.

[44] A.J. Martin. *Formal progress transformations for VLSI circuit synthesis in Formal Development of Programs and Proofs.* Addison-Wesley, 1989.

[45] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *AUSCRYPT '90: Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278, Cambridge, MA, USA, 1990. MIT Press.

[46] C. Matthew. Universality in elementary cellular automata. *Complex Systems*, 15:1–40, 2004.

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

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

[49] S. Mohammadi, S. Furber, and J. Garside. Designing robust asynchronous circuit components. *Circuits, Devices and Systems, IEE Proceedings -*, 150(3):161–166, June 2003.

[50] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[51] G.E. Moore. No exponential is forever: but "forever" can be delayed! [semiconductor industry]. pages 20–23 vol.1, 2003.

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

[53] C.J. Myers. *Asynchronous Circuit Design.* Wiley-IEEE, 2003.

[54] O. Petlin and S. Furber. Designing c-elements for testability. *Technical Report UMCS-95-10-2, University of Manchester*, 1995.

[55] C. A. Petri. Nets, time and space. *Theor. Comput. Sci.*, 153(1-2):3–48, 1996.

[56] R. Ronen, A. Mendelson, K. Lai, Shih-Lien Lu, F. Pollack, and J.P. Shen. Coming challenges in microarchitecture and architecture. *Proceedings of the IEEE*, 89(3):325–340, Mar 2001.

[57] P. Rothemund. Folding dna to create nanoscale shapes and patterns. *Nature*, 440:297–302, Mar 2006.

[58] M.S. Shakiba, D.A. Johns, and K.W. Martin. Bicmos circuits for analog viterbi decoders. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 45(12):1527–1537, Dec 1998.

[59] M. Shams, J.C. Ebergen, and M.I. Elmasry. Modeling and comparing cmos implementations of the c-element. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 6(4):563–567, Dec 1998.

[60] O. Shental, N. Shental, S. Shamai (Shitz), I. Kanter, A.J. Weiss, and Y. Weiss. Discrete-input two-dimensional gaussian channels with memory: Estimation and information rates via graphical models and statistical mechanics. *Information Theory, IEEE Transactions on*, 54(4):1500–1513, April 2008.

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

[62] A. Smith. *Cellular Automata Theory*. PhD thesis, Stanford University, 1970.

[63] M. Smith, Y. Bar-Yam, Y. Rabin, N. Margolus, T. Toffoli, and C.H. Bennett. Cellular automaton simulation of polymers. *Mat. Res. Soc. Symp. Proc.*, 248:483–488, 1992.

[64] J. Sparsø and S. Furber. *Principles of Asynchronous Circuit Design - A Systems Perspective*. Kluwer Academic Publishers, dec 2001.

[65] F.-X. Standaert, G. Piret, N. Gershenfeld, and J.-J. Quisquater. Sea: A scalable encryption algorithm for small embedded applications. In *Smart Card Research and Applications, Proceedings of CARDIS 2006, LNCS*, pages 222–236. Springer-Verlag, 2006.

[66] X. Sun. Analogic for code estimation and detection. Master's thesis, Massachusetts Institute of Technology, Sep 2005.

[67] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, 1989.

[68] S. Tam, J. Leung, R. Limaye, S. Choy, S. Vora, and M. Adachi. Clock generation and distribution of a dual-core xeon processor with 16mb l3 cache. pages 1512–1521, Feb. 2006.

[69] K. Tanaka. Statistical-mechanical approach to image processing. *Journal of Physics A: Mathematical and General*, 35(37):R81–R150, 2002.

[70] K. Tanaka, J. Inoue, and D. M. Titterington. Probabilistic image processing by means of bethe approximation for q-ising model. *Journal of Physics A: Mathematical and General*, 36:11023–11036, 2003.

[71] T. Toffoli and N. Margolus. *Cellular Automata Machines: A New Environment for Modeling*. The MIT Press, 1987.

[72] K. van Berkel. Beware the isochronic fork. *Integr. VLSI J.*, 13(2):103–128, 1992.

[73] J. Van Campenhout, P. R. A. Binetti, P. R. Romeo, P. Regreny, C. Seassal, X. J. M. Leijtens, T. de Vries, Y. S. Oei, R. P. J. van Veldhoven, R. Notzel, L. Di Cioccio, J.-M. Fedeli, M. K. Smit, D. Van Thourhout, and R. Baets. Low-footprint optical interconnect on an soi chip through heterogeneous integration of inp-based microdisk lasers and microdetectors. *Photonics Technology Letters, IEEE*, 21(8):522–524, April15, 2009.

[74] B. Vigoda. *Analog Logic: Continuous-Time Analog Circuits for Statistical Signal Processing*. PhD thesis, Massachusetts Institute of Technology, Jun 2003.

[75] B. Vigoda, J. Dauwels, M. Frey, N. Gershenfeld, T. Koch, H.-A. Loeliger, and P. Merkli. Synchronization of pseudorandom signals by forward-only message passing with application to electronic circuits. *Information Theory, IEEE Transactions on*, 52(8):3843–3852, Aug 2006.

[76] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269, Apr 1967.

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

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

[79] K. Wee, L. Turicchia, and R. Sarpeshkar. An analog vlsi vocal tract. *IEEE Transactions on Biomedical Circuits and Systems, in press*, 2008.

[80] E. Yahya and M. Renaudin. Qdi latches characteristics and asynchronous linear-pipeline performance analysis. *Research Report, TIMA-RR–06/-01–FR, Caltech*, 2006.

[81] Lie-Liang Yang and L. Hanzo. Iterative soft sequential estimation assisted acquisition of m-sequences. *Electronics Letters*, 38(24):1550–1551, Nov 2002.

[82] Lie-Liang Yang and L. Hanzo. Acquisition of m-sequences using recursive soft sequential estimation. *Communications, IEEE Transactions on*, 52(2):199–204, Feb. 2004.

[83] J.S. Yedidia, W.T. Freeman, and Y. Weiss. Constructing free-energy approximations and generalized belief propagation algorithms. *Information Theory, IEEE Transactions on*, 51(7):2282–2312, July 2005.