Matrix Multiplication with Asynchronous Logic Automata

by
Scott Wilkins Greenwald

B.A. Mathematics and German, Northwestern University, 2005
M.S. Scientific Computing, Freie Universität Berlin, 2008

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 2010

author: [illegible]
Program in Media Arts and Sciences
September 2010

certified by: [illegible]
Neil Gershenfeld
Professor of Media Arts and Sciences
Center for Bits and Atoms, MIT

accepted by: [illegible]
Pattie Maes
Associate Academic Head
Program in Media Arts and Sciences

Matrix Multiplication with Asynchronous Logic Automata

by Scott Wilkins Greenwald

B.A. Northwestern University, 2005
M.S. Freie Universität Berlin, 2008

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences
at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 2010

Abstract

A longstanding trend in supercomputing is that as supercomputers scale, they become more difficult to program in a way that fully utilizes their parallel processing capabilities. At the same time they become more power-hungry – today's largest supercomputers each consume as much power as a town of 5000 inhabitants in the United States. In this thesis I investigate an alternative type of architecture, Asynchronous Logic Automata, which I conclude has the potential to be easy to program in a parametric way and execute very dense, high-throughput computation at a lesser energy cost than that of today's supercomputers. This architecture aligns physics and computation in a way that makes it inherently scalable, unlike existing architectures. An ALA circuit is a network of 1-bit processors that perform operations asynchronously and communicate only with their nearest neighbors over wires that hold one bit at a time. In the embodiment explored here, ALA circuits form a 2D grid of 1-bit processors. ALA is both a model for computation and a hardware architecture. The program is a picture which specifies what operation each cell does, and which neighbors it communicates with. This program-picture is also a hardware design – there is a one-to-one mapping of logical cells to hardware blocks that can be arranged on a grid and execute the computation. On the hardware side, it can be seen as the fine-grained limit of several hardware paradigms which exploit parallelism, data locality and application-specific customization to achieve performance. In this thesis I use matrix multiplication as a case study to investigate how numerical computation can be performed in this substrate, and how the potential benefits play out in terms of hardware performance estimates. First we take a brief tour of supercomputing today, and see how ALA is related to a variety of progenitors. Next ALA computation and circuit metrics are introduced – characterizing runtime and number of operations performed. The specification part of the case study begins with numerical primitives, introduces a language called *Snap* for design for in ALA, and expresses matrix multiplication using the two together. Hardware performance estimates are given for a known CMOS embodiment by translating circuit metrics from simulation into physical units. The theory section reveals in full detail the algorithms used to compute and optimize circuit characteristics based on directed acyclic graphs (DAG's). Finally it is shown how the *Snap* procedure of assembling larger modules out of modules employs theory to hierarchically maintain throughput optimality.

Thesis supervisor:
Neil Gershenfeld
Professor of Media Arts and Sciences
Center for Bits and Atoms, MIT

Matrix Multiplication with Asynchronous Logic Automata

by Scott Wilkins Greenwald

Thesis reader: [illegible]

Alan Edelman
Professor of Mathematics
Massachusetts Institute of Technology

Matrix Multiplication with Asynchronous Logic Automata

by Scott Wilkins Greenwald

Thesis reader: [illegible]

Jack Dongarra
Professor of EECS
University of Tennessee, Knoxville

Acknowledgements

This work was supported by MIT's Center for Bits and Atoms, the U.S. Army Research Office under grant numbers W911NF-08-1-0254 and W911NF-09-1-0542, and the Mind-Machine Project.

I would like to thank my advisor Neil Gershenfeld for giving me and my collaborators Forrest Green and Peter Schmidt-Nielsen the opportunity to work on this interesting project together at this great institution. Thanks to Neil also for gathering support for the project – both financially and by getting others interested with his enthusiasm – as well as for the time he spent advising me on this thesis and all of us on this project. I would also like to thank my readers Alan Edelman and Jack Dongarra for their support and the time they spent giving input on the content of the thesis. Thanks to Jonathan Bachrach, Erik Demaine, Forrest Green, Bernhard Haeupler and Peter Schmidt-Nielsen for valuable discussions and related work done together on this project. Thanks also to David Dalrymple, Guy Fedorkow, Armando Solar-Lezama, and Jean Yang for valuable comments and discussions had together in the course of this research. Thanks to the Mind-Machine Project team for helping to identify the important questions. Also thanks to the members of my group for their constant support – Amy, Ara, Manu, Kenny, Max, Jonathan, Nadya, Forrest, Peter – also John and Tom in the shop, and Joe and Nicole for their facilitation.

On the personal side, I would like to thank my mother Judy, father Ken and grandmother Patricia for giving me both the upbringing and means to pursue the education that allowed me to get to this point. Thanks to Constanza for being tele-presently there for me on a daily basis through many months in front of the computer. Thanks to M^4 my family in Boston – Michael, Mumtaz, Malika, and Meira for their hospitality in having me over for dinner regularly and providing valuable emotional support. Thanks to my roommate Damien for encouraging me and for always listening to both rants and raves concerning research as well as life in general, and roommate Per for teaching me how to accomplish goals and not be deterred by menial tasks or obstacles.

Thanks go out to.... everyone back in Fort Collins – my family circle there Mom, Dad, Granny, Don, as well as friends through the years – Dave, Seth, Maria, Ki. Holdin' it down in Boston, Marcus and Mike S. Freunde in Berlin, danke dass ihr immer fuer mich da seid, Lina, Martin, Davor, Torsten, Hone, Nate, Jerry, Marco R. In der Schweiz, Marco T, always keeping me honest. New York crew – Nate, Jamie, Miguel, also Christian. At the Lab – Richard, Jamie, Marcelo, Haseeb, Ryan O'Toole, Forrest, Max, Jonathan for always being up for some fun, and Newton for being a friend and mentor. Jonathan B, thanks for your friendship beyond the work context. My uncle Chuck for being my ally in fun and teaching me about the wind, and Candy for great holidays and hosting me in Madison. The Jam/Ski jammers – thanks for your wisdom and giving me something to look forward to every year – Mcgill, Major, Cary, Joey, Jimmy O, Gino, and the rest. Also Olli E for good times in the Alps, sorry I couldn't make it this year – I had a thesis to write! Grazie alla famiglia Carrillo – Rosanna, Antonio, Giovanni, Carmen per il loro affetto e la loro ospitalità, and Sharon for lessons on cynicism. Camilla for always being a friend.

Contents

1 Introduction: Scaling Computation 8

2 Supercomputing Today, Lessons of History, and ALA 10

2.1 Software Today: Complex Programs for Complex Machines 10

2.2 Hardware Today: Power Matters 11

2.3 History's Successes and Limitations 12

2.4 ALA aligns Physics and Computation 13

3 What is Computation in ALA? 15

3.1 An ALA embodiment: 2D Grid of 1-bit processors 15

3.2 The Picture is the Program, the Circuit is the Picture 16

3.3 Performance of Circuits 17

3.3.1 Realization 17

3.3.2 Observables 18

3.4 Properties and Results 18

4 Building an ALA Matrix Multiplier 19

4.1 The Target Systolic Array / Dataflow Scheme 19

4.2 Primitive Operations for Numbers 21

4.2.1 Number Representation 21

4.2.2 Addition 21

4.2.3 Multiplication 21

4.2.4 Selection and Duplication 24

4.3 Design Language: "Snap" 24

4.3.1 Primitives for spatial layout 25

4.3.2 Primitive for interconnect: Smart Glue 26

4.3.3 Modules made of Modules: the hierarchical building block 26

4.4 Matrix Multiplier in "Snap" 27

5 Results: Simulated Hardware Performance **29**

5.1 HPC Matrix Multiplication in ALA Compared 29

5.2 Simulating ALA 32

5.3 Benchmarking ALA 33

5.3.1 Definition of Metrics 33

5.3.2 Data on ALA Matrix Multiplier 35

6 Graph Formalism and Algorithms for ALA **37**

6.1 Formalism to Describe Computation * 37

6.1.1 Net, State, Update 37

6.1.2 Computation 38

6.2 Realization and Simulation * 38

6.2.1 Dependency Graph 39

6.2.2 Realization 39

6.3 Metrics * 40

6.4 Calculation of Throughput 41

6.5 Optimization of Throughput: Algorithmic * 44

6.6 Optimization of Throughput: Examples By Hand 44

6.6.1 Adder 44

6.6.2 Multiplier 44

6.6.3 Matrix Multiplier 45

7 A Guide to "Snap": Hierarchical Construction with Performance Guarantees **48**

7.1 Hierarchy and the Abstraction of Modules 48

7.2 The Induction Step: Making a Module out of Modules and Preserving Performance 49

7.3 DAG-flow: The Anatomy of a Sufficient Framework 50

7.3.1 Input Syntax and Assembly Procedure 50

7.3.2 Construction Primitives and Module Objects: Inheritance and Derivation of Attributes 51

8 Conclusion **52**

A Formal Results **54**

A.1 Stability under asynchronous updates 54

B Benchmarking Code **56**

C Example Layout Code **60**

C.1 Periodic Sequence generation 60

Bibliography **62**

List of Figures

3.1 Example of an update 15

3.2 Example computation: one-bit addition 17

4.1 Matrix Product Example 20

4.2 Matrix Product Tile Specification 20

4.3 Data Flow Tile 21

4.4 Full Adder and Serial Ripple Carry Adder 22

4.5 Pen-and-Paper Binary Multiplication 22

4.6 Flow Chart Diagram of ALA Multiplication 22

4.7 ALA Integer Multiplication Circuit 23

4.8 Selection and Duplication Modules 24

4.9 ABC Module example 25

4.10 CBA Module Example 26

4.11 ALA Tile 27

5.1 Update scheme used in *librala*. Credit: Peter Schmidt-Nielsen 33

6.1 Low-throughput Example 39

6.2 High-throughput Example 39

6.3 Throughput and Latency Example 40

6.4 Finite Dependency Graphs 42

6.5 12-gate adder with throughput-limiting cycle highlighted. 44

6.6 4-bit addition, before and after computation, showing max-throughput 45

6.7 ALA High-Throughput Multiplication 46

6.8 Pipelined 2×2 matrix multiply in ALA 47

7.1 Merging Complete Graphs with Glue 49

Chapter 1

Introduction: Scaling Computation

Supercomputing is an important tool for the progress of science and technology. It is used in science to gain insight into a wide variety of physical phenomena: human biology, earth science, physics, and chemistry. It is used in engineering for the development of new technologies – from biomedical applications to energy generation, technology for information and transportation. Supercomputers are needed when problems are both important and difficult approach – they help to predict, describe, or understand aspects of these problems. These machines are required to process highly complex inputs and produce highly complex outputs, and this is a form of complexity we embrace. It is neither avoidable, nor would we want to avoid it. The sheer size and quantity of details in the systems we wish to study calls for a maximal ability to process large quantities of information. It is this desire to *scale* that drives the development of computing technology forward – that is, the will to process ever-more complex inputs to yield correspondingly more complex outputs. Performance is usually equated with speed and throughput, but successful scaling involves maintaining a combination of these with ease of design, ease of programming, power efficiency, general-purpose applicability, and affordability as the number of processing elements in the system increases.

Looking at today's supercomputers, the dominant scaling difficulties are ease of design, ease of programming and power efficiency. One might suspect that affordability would be an issue, but sufficient funding for supercomputers has never been a problem in the countries that support them the most – presumably because they have such a significant impact the benefits are recognized to far outweigh the costs. Today, much more than ever before, the difficulties with scaling suggest that we cannot expect to maintain success using the same strategies we have used thus far. The difficulty of writing programs that utilize the capabilities of the machines is ever greater, and power consumption is too great for them to be made or used in large numbers. The complex structure of our hardware appears to preclude the possibility of a simple, expressive language for programs that run fast and in a power-efficient way. Experience with existing hardware paradigms, in light of both their strengths and their weaknesses, has shown that successful scaling requires us to exploit the principles of parallelism, data locality, and application-specific customization.

The model of Asynchronous Logic Automata aligns physics and computation by apportioning space so that one unit of space performs one operation of logic and one unit of transport in one unit of time. Each interface between neighboring units holds one bit of state. In this model adding more computation, i.e. scaling, just means adding more units, and not changing the entire object in order to adapt to the new size. It also means that computation is distributed in space, just as interactions between atoms are; there are no privileged points as one often finds in existing architectures, and only a single characteristic length scale – the unit of space.

Many topologies and logical rule sets exist which share these characteristics in 1D, 2D, and 3D. In the embodiment explored here, ALA circuits form a 2D grid of 1-bit processors that each perform one operation and communicate with their nearest neighbors, with space for one bit between neighboring processors. The processors execute asynchronously, and each performs an operation as soon as its inputs are present and its outputs are clear. This is both a paradigm for computation and a hardware architecture. The program is a picture – a configuration which specifies what operation each cell does, and which neighbors it communicates with. This program-picture is also a hardware design – there is a one-to-one mapping of logical cells to hardware blocks that can be arranged in a grid and execute the computation. Hardware embodiments are studied in Green [2010], Chen [2008], Chen et al. [2010] and a reconfigurable version is presented in Gershenfeld et al. [2010]. Upon initial inspection, we will see that by aligning physics and computation, ALA has the potential to exploit parallelism, data locality, and application-specific customization while also maintaining speed, ease of design and programming, power efficiency, and general-purpose applicability.

There are two types of applications on the horizon for ALA to which the analysis here applies. The first is a non-reconfigurable embodiment which could be employed to make "HPC-On-A-Chip" – special-purpose, high-performance hardware devices assembled from ALA. This type of design will be explored quantitatively in this thesis. Another context to which the analysis presented here applies and is useful is that of reconfigurable embodiments. Initial efforts at reconfigurable hardware exhibited prohibitive overhead in time and power, but with some clever-but-not-miraculous inventiveness we expect that the overhead can be brought down. Achieving the level of performance where the advantage of reconfigurability justifies the remaining overhead in time and power is an engineering problem that we surmise can be solved.

In this thesis, I use matrix multiplication as a case study to investigate how numerical computation can be performed in this substrate, and how the potential benefits play out in terms of hardware performance estimates. First, in Chapter 2 we take a brief tour of supercomputing today, and see how ALA is related to a variety of its progenitors. Next in Chapter 3 ALA computation and circuit metrics are introduced – allowing us to talk about a computation's runtime and number of operations. Chapter 4, the specification part of the case study, begins with numerical primitives, introduces a language called *Snap*[1] for design for in ALA, and expresses matrix multiplication using the two together. Hardware performance estimates are given in Chapter 5 for a known embodiment by translating time units and token counts from ALA simulation into physical units. This comparison is able to be performed without simulating full circuits at the transistor level. Chapter 6 comprises the theory portion of the thesis, detailing the algorithms used to compute and optimize circuit characteristics based on directed acyclic graphs (DAG's). The final Chapter 7 is a "Guide to *Snap*," which most importantly shows how the procedure of assembling larger modules out of modules employs the theory to hierarchically maintain throughput optimality.

[1] *codeveloped by Jonathan Bachrach with Other Lab*

Chapter 2

Supercomputing Today, Lessons of History, and ALA

In this chapter I paint a picture of the software and hardware landscape of today in which ALA was conceived. There is a wide variety of hardware for computation – and each kind leverages a different advantage which justifies it in a different setting. ALA occurs as an intersection point when the bit-level limit of many different concepts is taken – a point where physics and computation align – and therefore appears to be an interesting object of study. In the first section I review evidence that programming today is difficult and surmise that the complexity of programs is rooted in the complexity of hardware. Next I talk about the cellular automata of both Roger Banks [1970] and Stephen Wolfram [2002] as differing attempts at simple hardware. I contrast these with ALA, showing the differences in where the simplicity and complexity reside.

2.1 Software Today: Complex Programs for Complex Machines

As the world's supercomputers have grown in size, the difficulty of programming them has grown as well – that is, programming paradigms used to program the machines don't scale in a way that holds the complexity of programs (lines of code, ease of modification) constant. Perhaps even more importantly, once a program is prepared for some number of cores, it is a substantial task to modify it to work with some other number of cores on some other slightly different architecture. This is whether or not the new program is "more complex." Many resources have been invested in alleviating this problem in the past decade. One major effort has been DARPA's High Productivity Computing Systems project, running from 2002 to 2010 and involving IBM, Cray, and Sun. Each of these companies developed a new language in connection with its participation. This program has undoubtedly yielded important new paradigms and approaches for programming today's supercomputers. However, each of these languages is very large in size, making learning to program in any one of them a major investment of effort on the part of an individual programmer. In all likelihood this is an indication that programming today's supercomputers is fundamentally a complex task.

How did this come to be? Must writing programs be so difficult? When the Turing/von Neumann architecture was developed in the 1940's, the abstraction of sequential execution of instructions stored in memory was what allowed computer science to be born. Since then, most computer science has been an extension of it in spirit. When parallelism was introduced, sequential languages were given special parallel constructs, but remained otherwise unchanged. The giant languages that are most adapted to programming today's heterogeneous architectures reflect the amalgamation of this ecology of species of hardware which has resulted from the incremental process of scaling; fixing problems one by one as they occur – caches to remedy slow access to RAM (generating data locality), multiple CPU's to gain speed when speeding up the clock was no longer possible (parallelism), addition of GPU's for certain types of computation (optimization through customization). Each such introduction of new organs in the body of hardware has given birth to new and corresponding programming constructs which add to the complexity of choreographing computation. In July 2010, David Patterson published an article describing some of the "Trouble with Multicore," focusing on language aspects of the difficulties in parallel computing today[Patterson, 2010].

One might suppose that this complexity in the environment makes complex hardware and complex programs somehow unavoidable. Or that choreographing large physical systems is fundamentally hard. Certainly there is one aspect of "complexity" for which this holds true – the number of processing elements must go up in correspondence to the number of significant degrees of freedom in the data. On the other hand, I claim that another type of complexity – manifested in heterogenaity of hardware structure – is indeed avoidable. Fundamentally, there is evidence that simple programs can behave in a complex way and perform complex operations, if there is a sufficient number of degrees of freedom in the processing substrate. If we say the substrate is a piece of graph paper, sufficient degrees of freedom can be obtained by making the graph paper (but not the boxes) larger. In constrast to this conception of structural simplicity with many degrees of freedom, the supercomputers of today are complex in structure, making them difficult to program.

Viewed from this perspective, the concept of ALA is to reintroduce both simple programs and simple machines. The idea of using simple machines goes back to a computing model presented by Roger Banks in his 1970 thesis *Cellular Automata*[1970]. In his CA, a 2D grid of identical cells is synchronously updated using an update rule that considers only the near neighborhood of a cell in the previous state of a system to determine its new state. Banks shows that a logic function can be implemented in this model using a roughly 10×10 patch of the grid. In this way a heterogeneous network of logic functions can be constructed to perform a desired computation. A different form of automata computing has been proposed by Wolfram, wherein instead of generating an elaborate array of interconnected logic gates as an initial condition and projecting logic onto it, we take binary states as a form of native representation, and study what simple programs on simple substrates can do. The practical difficulty with either variety of cellular automata as a model for hardware is their combination of low information density and the requirement for strict global synchrony. ALA takes the Banks approach, maintaining the concept of choreographing logic to be performed on bits in an anisotropic grid, as opposed to the Wolfram approach which seeks emulated characteristics of physical systems in the spatial structure of an isotropic computational grid. Now, addressing the "simplicity" of this model: as such, programs constructed of logic functions are not necessarily "simple" programs – they must be expressed in some simple way in order to earn the title of simplicity. We'll attempt to do this in ALA, along with relaxing the requirement for synchrony. What I refer to as the simplicity of programs in ALA does not manifest itself as literal uniformity – we use a language which leverages hierarchy and modularity in (simple) program specifications to specify very non-uniform programs. Recall that an ALA program is a picture which specifies what operation each gate does and which neighbors it communicates with – this picture may appear irregular and complex in spite of the simplicity of the program that specified it.

2.2 Hardware Today: Power Matters

What was once an after-thought for hardware designers – how much power a piece of hardware consumes to perform its logical function – has now become the primary basis upon which candidate technological innovations in hardware are to be judged viable or not in high-performance computing. No longer is it only a question of speed – a great amount of attention is given to the speed/power tradeoff represented by supercomputer performance. The supercomputer that currently tops the TOP500, Jaguar, consumes 6.95 megawatts of power performing the Linpack benchmark. This equals the average per capita amount of power consumed by 4,900 people in the United States [1]. Responses to alarming figures such as this are manifested in numerous ways. One example is the introduction of the Green500 in 2005 (chun Feng and Cameron), which poses an alternative measure of success in supercomputing – power-efficient supercomputing – and thus challenges the exclusive authority of the speed-only TOP500 ranking. Another show of regard for power consumption in computing took place in 2007 with the forming of a global consortium, the Green Grid [gre,

[1]Based on total total national energy consumption. According to the US Energy Information Administration, the United States consumed 3.873 trillion kilowatt hours in 2008, which makes per capita energy consumption 1.42 kilowatts.

2010], focused on a variety of efforts to bring down power consumption in computing. This consortium notably involves the worlds largest computing hardware manufacturers – Intel, IBM, and many more.

2.3 History's Successes and Limitations

Now let's take a tour of today's technology and some of its history in order to extract the most important lessons learned – in terms of what scales and what does not. The combination of goals that we'll keep in mind are speed, energy efficiency, ease of design and programming (get the program, perturb the program), and general applicability. Ease of programming consists both of the ease of writing a program from scratch, as well as the ease of modifying a program to work on a system with different parameters (processors, number of cores, etc.). The primary principles I'll be referring to are parallelism, data locality, and optimization through customization. Parallelism is associated with increased speed, but often worse energy efficiency, ease of programming, and general applicability. Leveraging data locality is good for speed and saves power by eliminating unnecessary data transport, but strategies can be difficult to program and/or not generally applicable. Optimization through customization is good for speed and power, but is difficult to program, and strategies don't carry over between specialized applications. Considering these principles one-by-one, we'll see where they've been successful, but how each technology employing them has failed on some other account to scale gracefully.

Parallelism

Since the mid-1960's there has been an uninterrupted trend towards multi-processor parallelism in general-purpose computing, corresponding to increasing speed in performance. Originating in high performance computing, in the mid-1960's single-CPU machines such as the IBM Stretch were outperformed by Cray computers which introduced parallel processing in the form of peripheral CPU's feeding a single main CPU. Even as much attention was given to increasing processor clock speeds until a wall of practicality was hit in 2004, parallelism grew in the same way, leading up to today's massively parallel machines. As of June 2010, top500.org listed seven machines each with over 100,000 cores [Meuer et al., 2010]. While these enormous machines perform impressive computations, they also suffer from poor programmability and extreme levels of power consumption. In personal computing, the trend towards parallelism emerged somewhat later, it once again corresponds to sustained increases in performance. The first consumer 64-bit processor was the IBM PowerPC 970 announced in late 2002 [2007]. The first consumer dual-core processors hit the market in 2005 with AMD's release of the dual core Opteron [2010]. As of 2010, commodity PC's with up to eight cores (dual quad-core) are common in the market place. One problem with this form of growth in personal computing is that each discrete size and number of processors, when assembled into a PC, requires an extensive process of adaptation and optimization before it can derive any benefit from increased parallelism. Summarizing, parallelism in CPU's has been a success in speed, affordability, and general-purpose applicability, but now suffers from difficulties with power consumption and ease of design and programming.

The past several years have seen a rise in a significant new instance of successful multiprocessor parallelism in personal computing – the GPU. These appeared first in 1999 with the release of nVidia's GeForce 256, and today come in flavors which package up to several hundred cores in a single unit. For example, the nVidia Fermi weighs in at 512 stream-processing cores[2010]. GPU's have excellent performance in applications which are amenable to SIMD representation – trading off general-purpose applicability in favor of optimization through customization for performance. Efforts to scale GPU's have been relatively successful so far – however currently the model also requires much effort to go from one size to the next, each jump requiring its own redesign phases and set of creative solutions for problems not previously encountered. Similar to CPU's, the track record of GPU's shows success in speed and affordability, but differ in the broadness of their applicability – GPU's being much more restrictive.

Data Locality

The principle of exploiting data locality is avoiding unnecessary transport of data. In the traditional von Neumann model, nearly every bit the CPU processes has to travel some intermediate distance from memory to processor which is "always equal". It is neither closer if it is used more, nor further if it is used less. By introducing forms of locality, energy can be saved and speed gained.

GPU's and CPU's both use multiple levels of caches to gain speed and save energy by exploiting data locality. Optimizing cache coherency for CPU's is an art that is both difficult and very architecture-dependent, and the same can be said for GPU's. That is, data locality is introduced for speed at the cost of making design and programming more difficult.

The systolic array is one member of the family of architectures that exploits data locality natively. Systolic arrays enforce the use of only local communication, and as such avoid problems scaling problems associated with introducing long wires. For certain types of problems, most notably adaptive filtering, beamforming, and radar arrays – see for example Haykin et al., 1992 the economics have so far worked out in a way that made systolic arrays a viable investment in developing custom hardware – winning by optimization through customization. The principle of systolic system relies on a synchronized "pulse" or clock, and one problem with scaling them is the power cost of distributing clock signals. In any case Systolic arrays do a great job of taking advantage of parallelism and minimizing the use of long wires, but apply to a very particular set of algorithms which are amenable to implementation this way (including the matrix multiplier demonstrated in this thesis), losing on general-purpose applicability.

Optimization through Customization

Now let us turn our attention to special-purpose computing. In high-performance special applications, such as image and sound processing, custom-designed integrated circuits use parallelism to speed up specific mathematical operations, such as the discrete Fourier transform and bit sequence convolutions, by maximally parallelizing operations – this usually involves introducing redundancy, which costs power but boosts performance. These specialized designs perform well – the only problem with them is that they are optimized for one particular set of parameters to such a great degree that they generally won't work at all if naively modified. Not only are small changes in parameters non-trivial – large changes in parameters require completely different strategies since various scale-dependent elements cease to work – wires get too long, parasitic capacitances take over, and so on.

FPGA's have been used since the early 1990's in some applications to derive many of the benefits of custom hardware at a vastly reduced development cost: FPGA's require only software to be designed and not physical hardware. This is a testament to the fact that cost and ease of design are being optimized. As a rule of thumb, to perform a given task, an FPGA implementation uses about 10 times more energy than a full-custom ASIC, and about 10 times less than a general-purpose processor would require. They abstract a network of processors that perform 4-bit operations by using look-up tables to simulate interconnect. FPGA's can indeed be configured to operate in a very parallel way, but they suffer both from problems of design and programmability in terms of parametric modification. Currently it is hard to make a bigger FPGA, and it's hard to program two FPGA's to work together.

2.4 ALA aligns Physics and Computation

When we consider the "limit" of many of these areas of general- and special-purpose computing, ALA can be both derived from that limit, as well as side-step some of the difficulties associated with scaling which we have seen are present in each one. Recall that this limit is interesting because occurs at a point where concepts of space, logic, time, and state are equated - in one unit of space it performs one unit of logic and one unit of transport in one unit of time, and the interface between neighboring units of space hold one bit of state. Aligning these concepts means aligning physics and computation. This is the limit of multi-core parallelism because it uses anywhere from hundreds to billions of processors in parallel to perform a computation, but doesn't suffer from difficulties in scaling because it respects data locality. In a similar way it is the limit of FPGA's and GPU's, but scales better because the only characteristic length scale introduced is the size of a single cell. If configured in a uniform or regular way, it looks much like a systolic array, but is asynchronous so it doesn't suffer from difficulties with the power cost of operating on a global clock. It is asynchronous and distributed like the Internet, but has the ability to be pipelined better because of the substrate's regular structure. ALA is also a cellular automaton in the sense of R. Banks [1970]and J. Conway [1970], but one with a large rule set and a large number of states, some of which we separate into the static state of our logic gates. It has the addition property of independence of update order at one granularity, which allows us to evaluate updates in a distributed, local, and asynchronous fashion.

Each of these progenitors brings with it wisdom from which lessons can be learned in the development of ALA. We want designs to be scalable so as to avoid difficulties associated with characteristic length scales which are appropriate for some applications but not for others. We want programs to be parametric so they don't have to be rewritten to fit a similar application with only slightly different parameters. We want designs to be hierarchical and modular so that they remain manageable even when their complexity far exceeds that which can be perceived at once by a programmer. Based on the idea of asynchronous processors that communicate only locally, we believe that these goals can be achieved.

Chapter 3

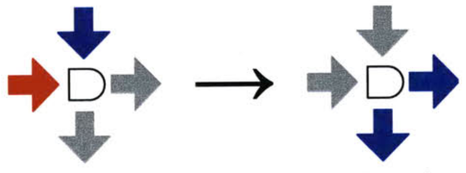
What is Computation in ALA?

Inspired by the idea that physical systems are composed of many units of space which each contain a certain amount of information, interact locally, and have the property that these local interactions causally induce changes which propagate throughout the system, we translate this idea into an asynchronous computing abstraction. We postulate a computing substrate in which logical nodes are arranged in a grid and single tokens can occupy the edges between the nodes. Edges are directional and nodes configured to have either one or two input edges, and between one and four output edges (distinct edges of opposite orientation are allowed). Whenever a node has a token on every one of its inputs and no token on any of its outputs, it performs an operation which pulls tokens from its inputs, performs a logical operation on them, and then returns its result as tokens on its outputs.

We pose the question: could this substrate be useful for performing numerical and mathematical computations? Answering this question requires answering a number of subsidiary questions: (i) can we program it? (ii) can we implement it and if so how does it compare to existing technologies and architectures? As a first step towards answering these questions, we'll need to first describe the model in detail, then define what a program is, and then examine on what basis we might compare this platform to other platforms.

3.1 An ALA embodiment: 2D Grid of 1-bit processors

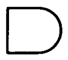
The version of asynchronous logic automata discussed here uses a square grid of logic gates with nearest-neighbor (N,S,E,W) connections. Tokens used for input/output are passed between the gates, and operations performed asynchronously – each gate fires as soon as both (i) a token is present on each of its inputs and (ii) all of its outputs are clear of tokens The gates are AND, OR, NAND, XOR, COPY, DELETE, and CROSSOVER. An AND gate with both inputs pointed in the same direction is referred to as a WIRE gate, and an XOR with both inputs in the same direction is a NOT. The gates COPY and DELETE are asymmetric, using one input as data and the other as a control, and can produce or destroy tokens, respectively. The input/output behavior of each gate is tabulated below.



We denote 0 tokens with blue arrows, and 1 tokens with red.

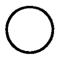
Figure 3.1: Example of an update

AND Gate

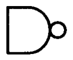
Glyph			
Inputs	2		
Outputs	1,2,3,4		
Behavior	in1	in2	out
	0	0	0
	0	1	0
	1	0	0
	1	1	1

WIRE Gate

AND gate with equal inputs

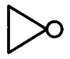
Glyph		
Inputs	1	
Outputs	1,2,3,4	
Behavior	in	out
	0	0
	1	1

NAND Gate

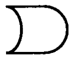
Glyph			
Inputs	2		
Outputs	1,2,3,4		
Behavior	in1	in2	out
	0	0	1
	0	1	1
	1	0	1
	1	1	0

NOT Gate

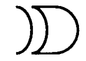
NAND gate with equal inputs

Glyph		
Inputs	1	
Outputs	1,2,3,4	
Behavior	in	out
	0	1
	1	0

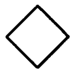
OR Gate

Glyph			
Inputs	2		
Outputs	1,2,3,4		
Behavior	in1	in2	out
	0	0	0
	0	1	1
	1	0	1
	1	1	1

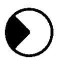
XOR Gate

Glyph			
Inputs	2		
Outputs	1,2,3,4		
Behavior	in1	in2	out
	0	0	0
	0	1	1
	1	0	1
	1	1	0

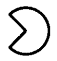
CROSSOVER Gate

Glyph				
Inputs	2			
Outputs	2			
Behavior	in1	in2	out1	out2
	0	x	0	x
	x	0	x	0
	1	x	1	x
	x	1	x	1
	0	0	0	0
	0	1	0	1
	1	0	1	0
	1	1	1	1

COPY Gate

Glyph				
Inputs	2			
Outputs	1,2,3,4			
Behavior	in	in_c	out	in'
	0	0	0	x
	0	1	0	0
	1	0	1	x
	1	1	1	1

DELETE Gate

Glyph			
Inputs	2		
Outputs	1,2,3,4		
Behavior	in	in_c	out
	0	0	0
	0	1	x
	1	0	1
	1	1	x

3.2 The Picture is the Program, the Circuit is the Picture

Because the ALA architecture is distributed and asynchronous, the concepts *program* and *computation* need to be defined. In ALA "the picture is the program," in the sense that viewing a representation of the state of a circuit is sufficient information to execute it. A circuit can be viewed as executing a *program*, which takes a set of inputs and computes a desired set of outputs, if we select input sites to feed in data as sequences of tokens and output sites to monitor sequences of tokens emitted. Figure 3.2 shows an example layout and computation.

One very important property of computations in ALA is that they are deterministic – causality is enforced locally everywhere, so that the result is guaranteed to be correct and consistent.

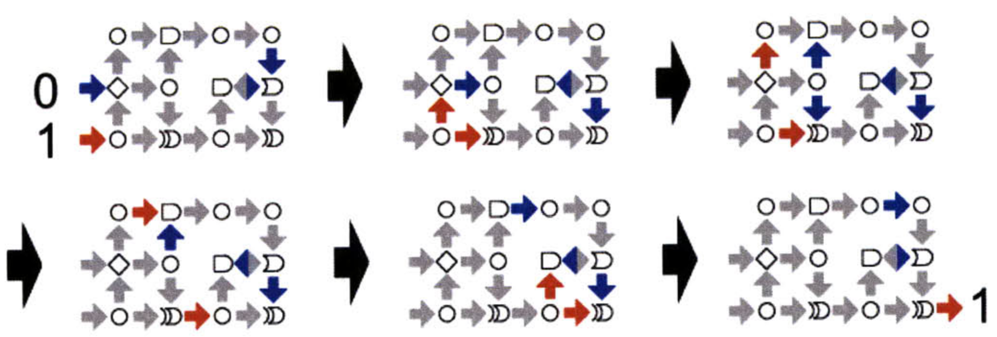


Figure 3.2: Example computation: one-bit addition

3.3 Performance of Circuits

Now that we know what is meant by computation, we can ask some important questions about the "performance" of such circuits. Four very important measures for program/circuit execution are latency, throughput, energy, and power. That is, how long does it take to get an answer, how frequently can answers be had, how much energy to does it take to get an answer, and how much power does the circuit consume if it is continuously producing answers? In today's hardware these questions are general answered in terms of global clock cycles and the energy that is consumed. Since ALA circuits are asynchronous, we need to characterize circuits in a slightly different way which will still have the effect of telling us when to expect answers and at what cost.

It turns out that by considering the net of dependencies defined by circuit layouts, we can formally describe gate firing times in terms of both (i) token arrival times from the outside world and (ii) the times required by tokens to propagate and gates to perform their operations in some hardware or software embodiment. An instance of distributed evaluation of this sort, in which each firing of each gate is indexed and assigned a time, is called a *realization*. Based on the concept of realization, we can translate and define the observables latency, throughput, energy, and power for these circuits.

3.3.1 Realization

In real applications, the asynchronicity of these circuits allows them to both react to external inputs whenever they become available, and also only to perform operations when they are necessary – when the circuit is static, it doesn't do anything. Done right in hardware, this means the circuit consumes no power to do nothing. This is not generally the case for today's hardware, except in specific circumstances when special measures are taken. When we design these circuits it isn't feasible to simulate every possible timing of token arrival at input gates. We need a robust way of characterizing circuit performance irrespective of the behavior of the outside world in any particular instance. For this purpose, we use define a model for simulating circuit evaluation in discrete time – an entire circuit is considered to have some initial state in which positions of all tokens are known. Then, all cells which are ready to fire in that state are at once updated, producing a second discrete-time state. We can proceed in this way in "bursts" performing "burst-updates" in which all cells that are ready to fire do fire from one step to the next. As such, we can describe realizations of a circuit in discrete time, assigning an integer time to each firing of each gate as the circuit executes. From here on we will refer to burst updates as units of time. In Chapter 6 it will be shown rigorously that these pseudo-synchronous updates are indeed very accurate at characterizing the behavior of our circuits.

In addition to burst updating, another assumption that is useful when characterizing circuits is the notion of *streaming*. That is, when inputs are assumed to be available whenever inputs are ready to accept them, and circuit outputs are always clear to drain output tokens the circuit produces. When inputs are streamed to circuits which begin in some initial state, evaluation at first occurs in an irregular way, producing outputs at some variable rate. After some amount of time they settle down into an equilibrium streaming state. I consider these phases of evaluation – the *initial* and *equilibrium* modes – separately and in relation to one another when defining observables.

3.3.2 Observables

Using the burst-update model to discretize the times at which events occur, it is possible to define our observables in a straight-forward way. More formal definitions will be given in Chapter 6.

Definition (Throughput). The *throughput* of a gate in a circuit is the average number of times a cell fires per unit time in the equilibrium mode of evaluation

Definition (Latency). The *latency* between an input and an output gate is the delay in units of time between when the input gates fires and when the output gate fires a corresponding output token. The notion of correspondence is defined by tracing the logical path of a token through the circuit. When streaming in equilibrium, the latency between an input and an output gate may follow some regular pattern. What is most often of interest is the *average latency*, which I'll usually refer to as *latency.*

Definition (Energy). The amount of *energy* required for a computation is the number of cell firings within the circuit (not including the motion of tokens along data lines outside the circuit in question) which are logically required to produce the output bits from the input bits. Note that this number doesn't depend on the discrete-time model, only on the logical dependencies within the circuit.

Definition (Power). The *power* consumed by a circuit is the number of cell updates per unit time in the equilibrium mode of evaluation. This number depends explicitly on the assumption of the discrete-time model.

3.4 Properties and Results

ALA has several important theoretical and practical properties. Because of cellular modularity, power required for a computation can be computed by counting tokens fired. Execution time can be efficiently computed by making assumptions about time required to update cells, and the result is quantifiably robust. Simulation can be done fast in parallel, and when cells are assumed to all take unit time to update, this fast simulation also reflects expected time of firing events in a hardware implementation. If the dataflow graph of a computation contains no cycles, throughput can be optimized in an ALA circuit such after in initial latent period, results are produced as fast as output cells can fire. There is a constructive procedure that allows for this optimization to be maintained through the process of creating modules from submodules and treating them as "black boxes." Each instance of optimal module assembly requires only solving a problem that concerns only the modules own submodules at a single level of hierarchy, and not a global problem involving all fine circuit details. Because of the blackbox property and existence of the algorithmic optimization procedure, layout in ALA lends itself well to abstraction in a programming language.

Chapter 4

Building an ALA Matrix Multiplier

Serving the goal of evaluating ALA as a strategy for designing hardware, I have selected matrix multiplication as an end goal to construct and profile in this thesis. Since establishing an intuitive and powerful interface with programmers is of critical importance for viable computing substrates, my collaborators and I have made a substantial effort to develop and language and implementation that automate aspects of circuit layout that are irrelevant to function, and to specify function as succinctly as possible. In the first section I'll introduce the "source" specification – a systolic array for matrix multiplication – which we'll aim to automatically lay out by the end of the chapter. In the next section, the primitive operations addition, multiplication, selection, and duplication are constructed "manually" by mapping familiar representations into ALA. In the following section the primary language constructs for the language *Snap* are introduced. In the final section I illustrate the use of *Snap* first at a very low level to construct a circuit for subtraction, and then finish up at a high level with matrix multiplication.

4.1 The Target Systolic Array / Dataflow Scheme

The type of specification we'll use for matrix multiplication draws from the theory and practice of both systolic arrays and dataflow programming. The major distinction is that it applies the principles on a single-bit level, when they normally apply to blocks that process words and perform multiple operations. The one-bit ALA substrate is universal, so it's possible to implement any systolic array (or any irregular structure), although the asynchronicity property means that scheduling is not explicitly necessary. When dataflow programming is considered at the one-bit level, there is a one-to-one mapping between dataflow nodes and hardware nodes, subject to the constraint the graph be embeddable in the plane. Recall that we have both crossovers and the ability to buffer additionally in a way that affects geometry but not logic, making this constraint somewhat benign.

To implement matrix multiplication in ALA we construct an $n \times n$ array of tiles that perform similar functions. Let's suppose we want to compute the product C of a matrix A with a matrix B. We represent the matrices as column vectors, so that the elements of columns occur in sequence along parallel channels – the desired configuration is show in Figure4.1.

In order to produce this flow of data, each tile performs five functions:

(i) select an element w from an incoming vector of B and reads it into a register.

(ii) direct the incoming column vector of B unmodified to its neighboring tile to the south

(iii) repeatedly multiply the selected element w successively with elements v of a column vector of A that arrive in sequence.

$$\begin{array}{ccc} A & B & C \\ \begin{pmatrix} a & c \\ b & d \end{pmatrix} & \begin{pmatrix} k & m \\ l & n \end{pmatrix} = & \begin{pmatrix} ak+cl & am+cn \\ bk+dl & bm+dn \end{pmatrix} \end{array}$$

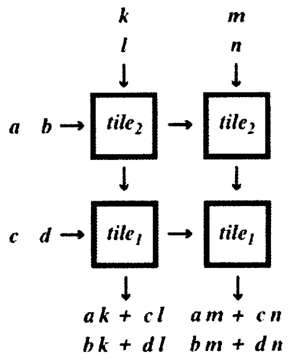


Figure 4.1: Matrix Product Example

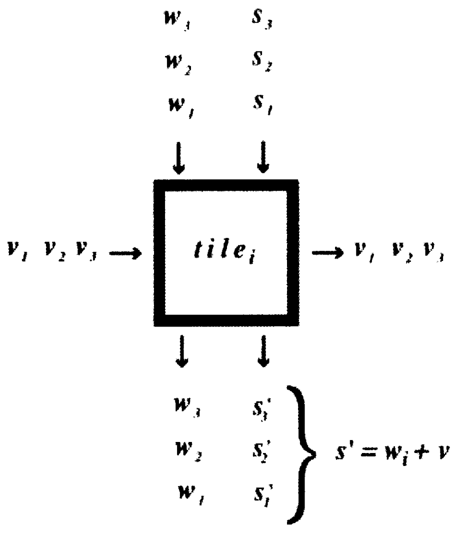


Figure 4.2: Matrix Product Tile Specification

(iv) directs the incoming column vector of A unmodified to its neighboring tile to the east

(v) add the products obtained (step iii) successively to the elements s of a third vector and passes the sums s' to its neighboring tile to the south. These sums are the elements of a column of C which are accumulated as they work their way from north to south.

This is illustrated in Figure 4.2.

Now we need to synthesize a dataflow diagram of the internals of the tile in terms of functional blocks. This is pictured in Figure 4.3.

This scheme is similar to one that appears in Fig 4b, p. 250 of Moraga [1984] – the version that appears there differs from this one in that it does not include the select/duplicate operation – the element of w selected by the select/duplicate in my model is assumed to have been preloaded and the issue of performing the preloading is not addressed. Note that there are a total of four levels of representation involved: (1) gates, (2) primitives (addition, etc), (3) systolic array tiles, (4) assembly of tiles. The plan for this chapter will be to acquire the means to turn this functional block / dataflow description for a matrix multiply into a parametric ALA circuit.

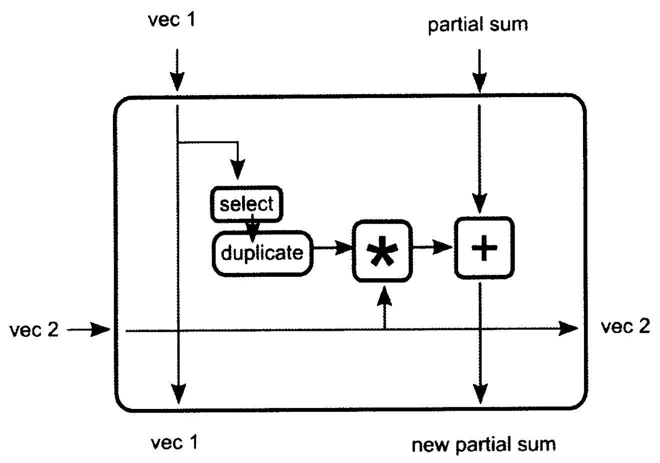


Figure 4.3: Data Flow Tile

4.2 Primitive Operations for Numbers

In order to do hierarchical construction of circuits that perform numerical operations, we need a level of primitives above individual logic gates upon which to build higher level operations. In the case of matrix multiplication, the necessary intermediate primitives are addition, multiplication, selection, and duplication, as depicted in Figure 4.3.

4.2.1 Number Representation

Many number representations can be implemented in ALA. Numbers can be represented on parallel channels, in sequence, or anywhere in between (e.g. sequences of several bits on several channels). Representations can be fixed-length or variable length, and can range from very short to very long. Integer, fixed point, or floating point numbers could be implemented. Non-binary representations (still using 0- and 1-tokens) are thinkable as well - a unary representation could represent numbers as the lengths of 0-token sequences separated by 1-tokens, or decimal representations could represent digits individually or in groups as sequences of binary bits of fixed length. Native arithmetic operations could then work within these representations. Each possible number representation represents a certain trade-off between power, speed, and latency that varies by algorithm and application. With proper theory developed, it is conceivable that the right representation could be dialed in very precisely on a per-application basis.

In this thesis I use unsigned integers represented as fixed-length bit strings in sequence. This is likely the simplest representation to work with, hence a wise choice for an initial exploratory effort.

4.2.2 Addition

The kernel of the one-bit serial adder is a full adder, the input/output behavior of which is shown in the table of Figure 4.4. In order to make a streaming asynchronous adder, we implement the full adder in its familiar form in ALA, with the modification that the carry output is wired back into the carry input, and a 0 token initialized on this wire. This is shown in Figure 4.4.

4.2.3 Multiplication

In this section we aim to illustrate correspondences between heuristic and implementation in ALA in the example of integer multiplication. We look at the algorithm in three different representations. Firstly we

$\{A, B, C_{in}\}$	$\{S, C_{out}\}$
$\{0,0,0\}$	$\{0,0\}$
$\{1,0,0\}$	$\{1,0\}$
$\{0,1,0\}$	$\{1,0\}$
$\{1,1,0\}$	$\{0,1\}$
$\{0,0,1\}$	$\{1,0\}$
$\{1,0,1\}$	$\{0,1\}$
$\{0,1,1\}$	$\{0,1\}$
$\{1,1,1\}$	$\{1,1\}$

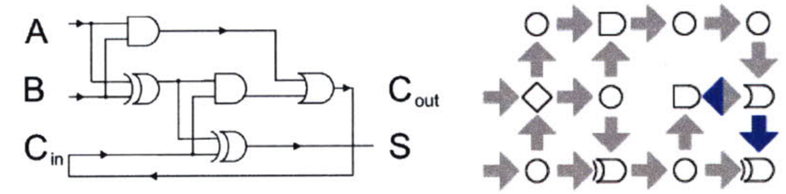


Figure 4.4: Full Adder and Serial Ripple Carry Adder

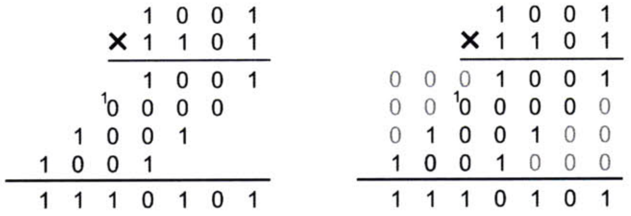


Figure 4.5: Pen-and-Paper Binary Multiplication

look at the familiar pen-and-paper heuristic. We then unpack this as a flow chart of states that is amenable to implementation in ALA. Finally we see the ALA circuit which exhibits both the spatial structure of the heuristic and a spatial structure according to the stages in the flow chart representation.

To begin, Figure 4.5 illustrates the binary version of the familiar technique for performing multiplication with pen and paper. Partial products are obtained by multiplying individual digits of the lower multiplicand **B** by the entire string **A**. In the right part of the diagram, 0's are added to explicitly show the complete representation of the summands that are added to produce the product. We will use these in order to represent the operation in ALA (note that other representations within ALA are possible).

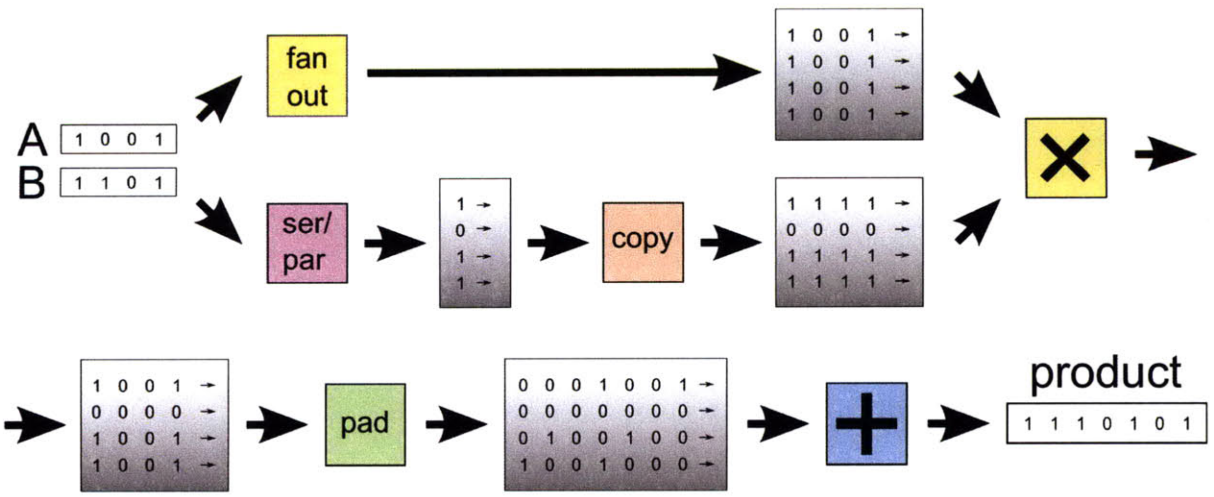


Figure 4.6: Flow Chart Diagram of ALA Multiplication

The input labeled **A** in Figures 4.6 and 4.7 is analogous to the upper number in pen-and-paper multiplication, **B** is the lower, and we'll say that these have n and m bits, respectively. In the example, $n = m = 4$. In Figure 4.6 we assign a color to each *stage* of the computation, and different shades in the gray scale gradient to different *channels* which compute in parallel in the course of the computation. The contents of the gray boxes indicate the bits that pass through the corresponding point in the circuit in the course of the example computation $1001 * 1101$.

The same color and shading schemes are used in Figure 4.7 depicting the ALA circuit. The two schemes are conflated and imposed on the ALA circuit structure to indicate coincident spatial distribution of channels and stages. We now look at the ALA implementation of each of the stages in detail.

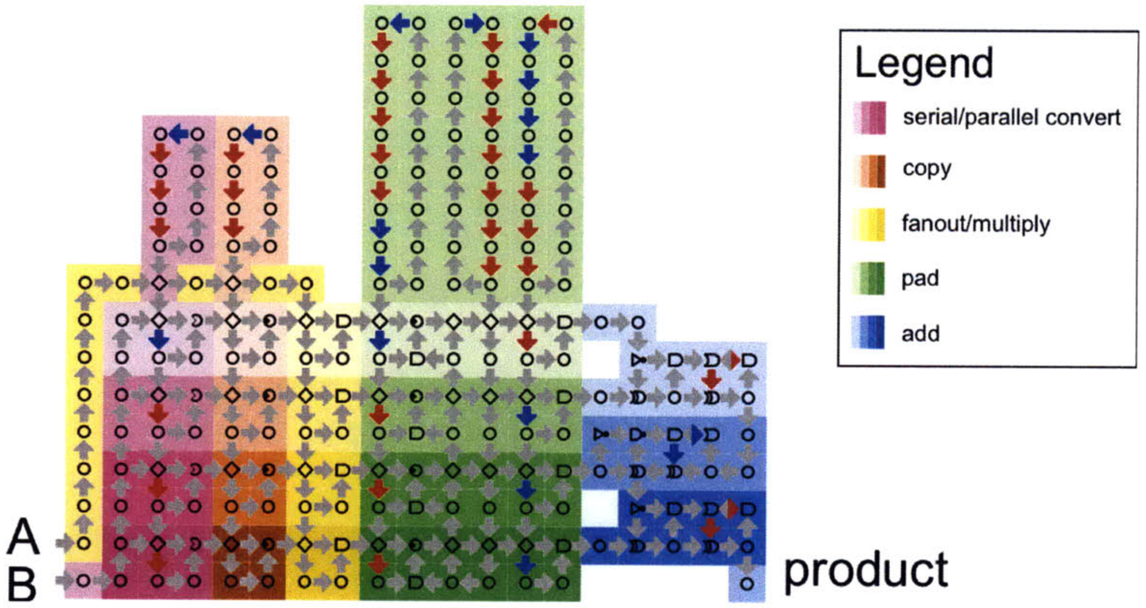
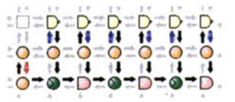


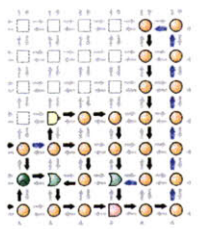
Figure 4.7: ALA Integer Multiplication Circuit

In the *serial/parallel convert* stage, input **B** is first fanned out into n copies. Using a delete gate for each copy and a global control sequence $[1] + [0] * (n - 1)$ (in Python notation) that resides in a bit loop, only a single bit from each copy passes to the duplicate stage. The pictured initialization bits create the "phase" offset between channels that results in each copy having a different bit selected. As pictured, the uppermost channel selects the first (least-significant) bit of input **B**, the next channel the next bit, and so on. In the *copy* stage, each channel creates m copies of its bit. This is achieve by giving each copy gate the sequence $[1] * (m - 1) + [0]$, clearing the bit on the m-th gate update.

Then in the *fanout/multiply* stage these sequences of m 0's and m 1's are bit-wise AND'ed with copies of **A** to compute the summands for addition. Note that the fanout of **A** happens as it propagates from top to bottom, with each stage of order two fanout occurring adjacent to its corresponding AND multiplication.

In the *pad* phase, the channels are padded with $n-1$ extra bits, distributed variously between front-padding and back-padding. The uppermost channel is back-padded with $n - 1$ bits, the next gets one bit front-padded and $n - 2$ bits back-padded, and so on. This achieves the offset between summands that is familiar from pen-and-paper multiplication. The ALA implementation uses three logic gates. The COPY gate is responsible for extending each m-bit product to an $m + n$ bit sequence, with the product appearing at the correct substring location in each channel. The phase offset is sufficient for getting the front-padding right. The AND gate below each COPY along with the bit loop that feeds it provides the stop condition to terminate the sequence after $m+n$ bits. Finally, the rightmost AND gate sets to 0 the bits outside the m-bit substring that corresponds to the product from the *multiply* stage (these are initially copies of the least and most significant bits of the m-bit products). This leaves us with summands of the format introduced in the right side of Figure 4.5.


(a) Sequence Generator


(b) Register Module

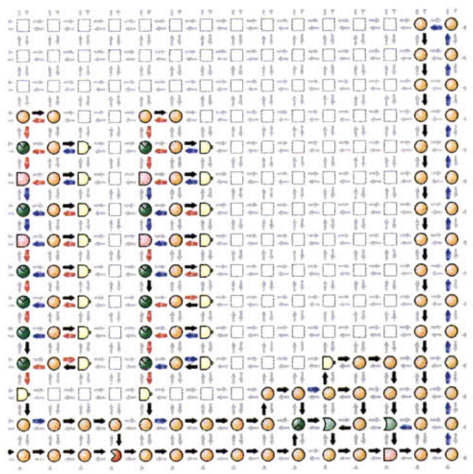

(c) Select/Duplicate

Figure 4.8: Selection and Duplication Modules

Finally, in the *sum* stage, the n padded summands of length $n + m$ are summed, cascading from top to bottom. The uppermost adder sums the two uppermost partial products, the next sums that sum with the next partial product, and so on progressing downwards until the final sum emerges from the lower-most channel.

4.2.4 Selection and Duplication

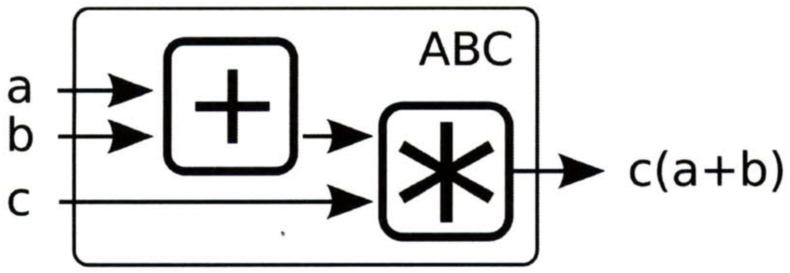
The core concepts of selection and duplication are controlled delete and controlled copy, respectively. For selection the principle is straightforward: there is an incoming sequence of data bits, and there is a control sequence in one-to-one correspondence with the data bits, with the value 0 for data bits that are to be "selected" or allowed to pass (not deleted), and 1 for those that are not selected (deleted). The simplest way to produce a periodic sequence is with a bit loop – as sequence lengths grow, linear growth of bit loop size is undesirable, so a logarithmic-sized representation is used – a very small class of sequences is sufficient for the purposes of selection and duplication. The language *Snap* will be introduced in the next section, and the parametric code that generates these "compressed bit loops," can be found in Appendix C.1. The scheme is similar to that presented in Green, 2010.

For duplication of a single bit, the story is similar to that of selection – there is a controlled copy and zeros on the control sequence let bits pass, while ones make multiply copies. In this way indexing the zeros in a control sequence establishes the correspondence to data bits, and any strings of 1's makes copies of the bit that its trailing zero indexes. If, however, we wish to make multiple copies of a sequence of bits which is given as data, we need a way to read the data bits into some kind of register, emit copies of the data some desired number of times, and then clear the register by reading in new data. A design for such a circuit is show in Figure 4.8c. (figures: Sequence Generator for seven 0's terminated in a 1, Duplication with clearing, Select/Duplicate for 4 x 4 matrix, 16-bit words)

For the purposes of $m \times n$ matrix multiplication, we need to be able to select one word out of each n in a sequence and make m copies of it – i.e., we need a selection module followed by a duplication module. In the figure we show such a module for a 4 x 4 matrix multiply with 16-bit words.

4.3 Design Language: "Snap"

In this section we'll see how to put the pieces together. The three primary functionalities we need are relative spatial alignment, interconnect, and the ability to define new modules as a collections of interconnected



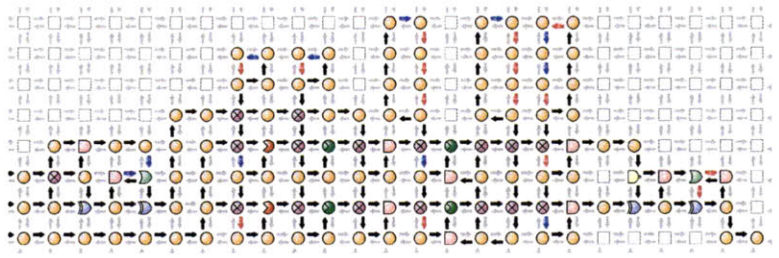


Figure 4.9: ABC Module example

submodules. The basic facilities for each of these will be addressed in the subsections that follow. The language *Snap* was developed in collaboration with Jonathan Bachrach.

In DAG-flow we specify a circuit construction procedure that alternates between vertical and horizontal concatenations. We begin by stipulating that individual modules have their inputs on the left and outputs on the right – arbitrarily choosing an axis. We introduce two primitive operations – (i) producing module columns by vertical concatenation of modules that operate in parallel, and (ii) horizontal concatenation of sequences of columns that are connected unidirectionally. Without loss of generality, we assert that these connections be directed from left to right, and specify them as permutations that map outputs of one column to inputs of the next. These are not strictly permutations since we allow fanout of single outputs to multiple inputs.

Once we have a specification of such a circuit, we begin layout with a pass of assembling columns in which we equalize width. Then, beginning by fixing the left-most column at the x=0, we add columns one-by-one, connecting them with smart glue, which takes as input potential values associated with the outputs and inputs in question, and buffering properly. This procedure could also be performed in parallel in a logarithmic number of sequential steps – pairing and gluing neighboring columns preserves the ability to optimally buffer connections between the larger modules thus assembled. In order to augment the class of circuits that can be defined using DAG-flow, we also introduce a rotation operator, which allows us to assemble sequences of columns, then rotation them and assemble columns with these as modules.

4.3.1 Primitives for spatial layout

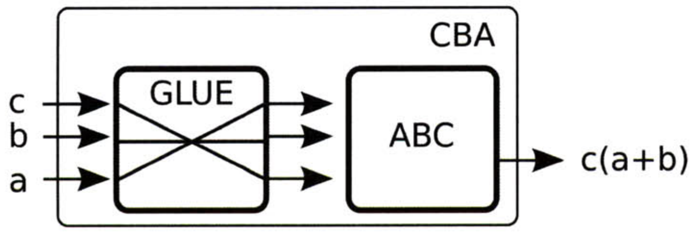
Here I'll briefly go over two indispensable programming constructs for two-dimensional layout: horizontal concatenation (*hcat*) and vertical concatenation (*vcat*). The idea is straightforward – a list of blocks/modules is aligned horizontally or stacked vertically. The constructs can be nested to create hierarchy.

Consider this example. Given the blocks we have from the previous section, suppose we want to make a module that computes $c\ (a+b)$. One way of specifying this in data flow would be as in Figure 4.9.

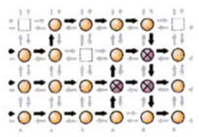
In *Snap* we specify this as

```
ABC = hcat[vcat[wire,adder],multiplier]
```

and get the circuit also shown in Figure 4.9.



(a) Module with Glue



(b) Glue Module

Figure 4.10: CBA Module Example

4.3.2 Primitive for interconnect: Smart Glue

Notice that in the *Snap* construction of the c(a+b) example that we end up with a circuit that requires the inputs to be spatially ordered from top to bottom (a, b, c). Suppose, however, that they aren't in that order – instead they're in the order (c, b, a). We have two options – either we tear our module apart (although in this simple example we could use some symmetry to accomplish this by just mirroring the layout vertically – but for the sake of generality we suppose this isn't so), or we produce an adapter, which we refer to as *smart glue* or just *glue*, to spatially permute the input channels. Now what we want is this, shown in Figure 4.10a.

By assuming that inputs come in on the left and outputs go out on the right, we can specify the connections in a glue module by a sequence of ordered pairs. The inputs and outputs are indexed according to their spatial ordering from bottom to top, so in this case the glue is specified as `glue[{1,3},{2,2},{3,1}]` and the module CBA generated by the syntax

```
CBA = hcat[vcat[wire,wire,wire],glue[{1,3},{2,2},{3,1}],ABC]
```

This glue module is shown in Figure 4.10b.

4.3.3 Modules made of Modules: the hierarchical building block

I tacitly presented the previous examples without highlighting several critical *Snap* behaviors which made them possible.

The first of these was the use of identifiers in hierarchy – I defined `ABC` in the first example, and used it as a submodule when defining `CBA`in the second example. That is, making modules-out-of-modules is a critical capability that allows circuits to be hierarchically constructed. The reason this is so important in this and many other systems that use hierarchy is that it allows us to, roughly speaking, define things of dimension n using only $log(n)$ expressions or lines of code.

The second capability can be called *block padding* – this means that when things are aligned horizontally or stacked vertically, the input and output channels of submodules are padded with wires so as to reach the edges of the bounding box of the new module. This way we avoid the hard problem of routing i/o through an arbitrary maze of structure. Instead we guarantee that at each level generating interconnect is only a problem of connecting points on the edges of adjacent rectangles. Observe, for example, that the single-gate "wire" in the specification of Figure 4.9 was padding into a sequence of wire gates in Figure 4.9. Hence the glue module in the second example only needed to solve the problem of connecting an aligned vertical set of points on the left with another such set on the right, as is visible in Figure 4.10b.

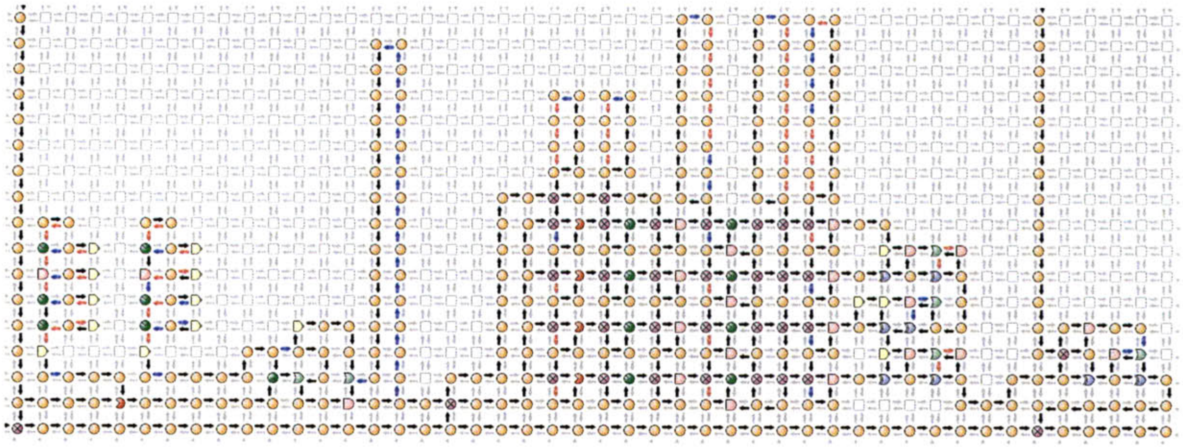


Figure 4.11: ALA Tile

4.4 Matrix Multiplier in "Snap"

Tile Specification Now we finally have all the tools to put together the tile described by Figure 4.3. The first thing we need to do is convert it to a format where dependencies flow from top-to-bottom and left-to-right. For now, the reason for this is that the language implements it this way – in Chapters 6 and 7 we will see more about why this is so. Once we have the figure in this format, the code can be read directly from it. In the orientation the figure is drawn, each vertical column corresponds to a line of code. A list of modules in the code (left-to-right) corresponds to stacking from bottom-to-top in the diagram. Lines alternate between specifying modules and specifying connectivity of neighboring columns.

```
tile[selectCopy_] :=
  matCons[
   {{cross, wire},
    straight,
    {wire, selectCopy},
    {{1, 1}, {1, 2}, {2, 3}},
    {wire, multiplier},
    straight,
    {hcatMat[{cross, wire}], vGlue[straight], adder}}
  ];
```

The selectCopy module is taken as an argument to the tile constructor because it is initialized in a way that will vary from bottom to top in the array – recall the subscript i that appeared in Figure 4.2 indicating which element is to be selected from the vector coming in from the north.

Note that an orientation-faithful spatial mapping between the code and Figure **??** is established by rotating the code 90 degrees counter-clockwise.

The resultant tile is pictured in Figure 4.11.

Array Construction Now, in order to construct the final array, we need to generate the n distinct selectCopy modules, assemble them into columns, and then concatenate the columns horizontally.

```
matMul[dim_] :=
  Module[{scs, htiles},
   scs = Table[selectCopy[dim, i], {i, dim}]; (* select/copies *)
   column = Table[tile[sc], {sc, scs}];
   matCons[rep[column, dim]];
```

The `rep` command makes `dim` copies of its argument.

Chapter 5

Results: Simulated Hardware Performance

The High Performance Computing Challenge is designed to measure up candidate HPC technologies' performance in terms of time, area, cost, and power. One important benchmark is matrix multiplication, and with this in mind I've chosen matrix multiplication as a benchmark task for ALA. Results suggest that the massive parallelism of ALA makes for very high throughputs: a present design performs 42 tera-ops with a 1.1 m^2 chip assembly running at 488 kW. Straight-forward improvements of this design point to a next generation 7.4 cm square of ALA chip assembly that performs the same 42 tera-ops at 20 kW. These estimates are made possible by the fact that the modularity of ALA cells allows us to obtain transistor-level power and time estimates without simulating the full system at transistor level - counting tokens is sufficient for computing power, and accumulating cell update times is sufficient for computing speed and latency. Only individual cells (roughly 30 transistors in design-in-progress at time of writing) need to be simulated using a physics engine.

5.1 HPC Matrix Multiplication in ALA Compared

Matrix Multiplication: Basis for Comparison As promised, I will now go through a comparison of a 1024 $\times$ 1024 matrix multiplication performed in ALA with one performed on a supercomputer. We will compare the power consumption and throughput of Jaguar performing the Linpack benchmark for LU decomposition (linear equation solving) on double-precision operands with an ALA circuit performing matrix multiplication with 64-bit integer operands. There are two notes on this comparison. Firstly, comparing throughput (FLOPS) of matrix multiply with that of LU decomposition is justified by a comparison of asymptotics: LU decomposition and matrix multiplication have the same asymptotic complexity $O(n^{2.376})$, based on the Coppersmith-Winograd Algorithm. Second, the operands used in the Linpack benchmark are double precision floating point, whereas those for the ALA matrix multiplier are 64-bit integers. Because data transport in and out of the circuit is one of the largest costs associated with numerical operations, the best side-by-side comparison between integer and floating point operations in terms of power and throughput is obtained by using numbers of equal size.

Simulation The measurements presented in the Table 5.1 and the remainder of this section were computed with an efficient parallel circuit simulator written by Forrest Green and Peter Schmidt-Nielsen. Some details about the implementation will be presented in the next section.

Throughput The matrix multiplication is implemented in ALA as a grid of similar tiles. Each tile processes one vector through one vertical channel and performs a multiply and an add. Producing bits at 1.27

machine name	description	data type	efficiency (Op/J)
Jaguar (top500)	linpack	experiment	2.5E8 (FLOP)
Juelich (green500)	linpack	experiment	7.7E8 (FLOP)
ALA	present	simulation	5E7 (Int Op)
ALA	no wires	estimate	2E8 (Int Op)
ALA	32 nm, optimized	estimate	1.2E9 (Int Op)

(a) Energy Efficiency in Number of Ops per Joule

name/description	process	description	data type	area (μm^2)	linear (μm)
FPU [Kim et al., 2007]	32 nm	–	experiment	8.5E4	2.92E2
ALA MatMul Tile	90 nm	present	simulation	1.16E6	1.08E3
ALA MatMul Tile	90 nm	packed	estimate	3.15E5	5.61E2
ALA MatMul Tile	90 nm	packed, no wires	estimate	8.18E4	2.86E2
ALA MatMul Tile	32 nm	packed, no wires	estimate	1.03E4	1.01E2
ALA MatMul Tile	32 nm	packed, no wires, optimized	estimate	5.17E3	7.19E1
ALA 1024 × 1024	90 nm	present	simulation	1.22E12	1.11E6
ALA 1024 × 1024	32 nm	packed, no wires, optimized	estimate	5.46E9	7.39E4

(b) Area

Description	Throughput	Equilibrium Latency
ALA MatMul Channel	1.27 Gbps	2.6 μs
ALA 1024 × 1024 MatMul	1.3 Tbps	.21 ms

(c) Throughput and Latency

name/description	process	revision	data type	area (μm^2)	linear (μm)	update energy (pJ)
ALA Cell [1]	90 nm	present	simulation	1.21E2	1.10E1	0.17
ALA Cell	32 nm	process switch	estimate	1.53E1	3.91E0	0.057
ALA Cell	32 nm	optimized	estimate	7.7E0	2.77E0	0.028

(d) ALA Cell Information

Table 5.1: Numbers for Comparison

Ghz and performing 2 Ops every 64 hardware cycles implies performing 2.1×10^6 Ops at a rate of 1.98×10^7 Hz, i.e. 42 tera-ops per second! ($4.16 \times 10^{13} OPS$). Comparing this with numbers from Jaguar, with 224,162 cores (about 37,360 machines, 6-core, [some cores are CELL processors right? check this]) performing 1.76 Petaflops sustained (1.76×10^{15}), this is 47 Gigaflops per machine, which would make throughput of the ALA chip assembly equivalent to that of 885 machines. The data rate out is then 1.3 Tbps – 1024 channels each producing bits at 1.27 Gbps.

Power and Energy Consumption Let us begin with power. The number for "ALA current" presented in Table 5.1a was computed as follows. Updating a single cell consumes 0.17 pJ of energy averaging over all cell types. The tile from which the matrix multiply is generated occupies a rectangle of size 125×77, of which (only) 2700, or 28% of the rectangle's area is occupied. This is an artifact of the way that the circuit layout is done at present (more in Chapter 7), and leaves much room for improvement using throughput-maintaining packing algorithms, which I and my collaborators have not seriously explored as of the time of writing. The power consumed by this tile corresponds to 1078 cell firings per unit of time. A tiling 1024×1024 of these therefore fires 1.13×10^9 cells per unit of time, or 1.92×10^{-4} J per unit time. A hardware cell update does 2 units of time per cycle when the circuit being executed is buffered for maximum throughput, and operates at the minimum frequency taken over all cell types, which is 1.27 Ghz. Therefore the 1024×1024 array operates at 488 kW. Note that the equation of 2 units of time with one hardware-cell cycle is derived from a hardware dependency graph similar but not identical to that presented in Section 6.2.1. It is valid in the case of max-throughput circuits- details are presented in Green [2010].

Now let us consider updated power and speed metrics for a future CMOS ALA design based on straightforward improvements of the existing design. First of all we know that it is possible to replace buffering wire cells with "physical" wire cells which do not buffer and connect points neighboring cells physically rather than logically. These physical wires dissipate very little power. As the system scales, there is some maximum number of gates n that can be travelled before refresh is necessary. That is, for every desired "unbuffered" physical connection that exceeds n grid units, one or more extra logical buffers must be inserted to maintain proper operation. If this constant n equals, say, 100, then the power consumed by wire gates in a given circuit gets much smaller. In the present example, wires account for 70% of cells in the circuit – so the "no wires" version of the cell would consume marginally above 30% of the present power estimate. Next we assume that roughly a factor of two can be gained from expert optimization. Lastly, by migrating from 90 to 32 nm CMOS process, we presume that a roughly linear factor of three is gained. Although leakage can account for a larger percent of power dissipation in a finer process, the fact that the circuit at hand is optimized for throughput implies that the activity factor will remain very high and thus minimize losses from leakage. Taking these improvements into account, the new 4.16 tera-op, 1024×1024 matrix multiplier consumes 20.3 kW. In this configuration we get 1.2 billion ops per Joule. Comparing this with Jaguar operating on double-precision floats- Jaguar gets a "mere" 250 million floating point ops per Joule, giving the ALA chip a factor of 4.8 better power efficiency. Comparing to the Green500 leader, ALA does a factor of 1.5 better.

Area So how much die area does a such a chip assembly require – that does the work of 885 multi-core servers at 1/5 of the energy cost (plus cooling)? The power-hungry current version uses 1.12 m^2 of die. With all optimizations taken into account, the assembly instead occupies a 7.4 cm square. Recall that it consumes 20.3 kW of power, and delivers 42 tera-ops. Table 5.1b shows estimates of area and linear dimension of the chip assemblies for performing 1024×1024 integer (64-bit) matrix multiply.

Latency Equilibrium latency is for a 64-bit integer $n \times n$ matrix multiplication is 0.21 ms for the present hardware design. This is the time required to read in the data, perform the operation, and read out the product. Inputs can then be streamed in at 1.27 Ghz, so that the time increment on a given channel to read inputs is 2.6 μs – at 64×1024 bits per channel per operand. As of writing I have not simulated a 64-bit 1024×1024 matrix multiplication, but the data for smaller multiplications indicates that latencies for matrix multiplication scale as $O(n)$. Recall that we measure two kinds of latency: the length of the initial phase, when there are irregularities in the rate at which bits are produced, and the time it takes in equlibrium between when bits for an operand (in this case a matrix) start being consumed on the first input channel, until when the last corresponding output bit is produced on the last channel. From the tiling geometry and the max-throughput property, we expect there to be an exact formula for equilibrium matrix operation latency. The data indicates confirms this – equilibrium latency is for a 64-bit integer $n \times n$ matrix multiplication is $152 + 480\,n$ units of time, which equates to 0.21 ms for current hardware.

Cost Based on the following calculation, I conclude that the bare die hardware for the 42 tera-op matrix multiplier could be produced for about \$5500 per unit in bulk. A survey of die sizes and costs of Intel chips can be used as to estimate cost per area of producing CMOS chips in 32 nm process. The Intel Core i3, i5, and i7 are common desktop processors on the market at the time of writing. Dual core releases each have an $81mm^2$ CPU in 32 nm process and 114 mm^2 of graphics and integrated memory controller hardware in 45 nm process. Depending on frequency and cache configuration, these sell in bulk for between \$100 and \$300 per unit[2]. Because processors are graded and priced upon testing, it is conceivable that the cheapest processors could be sold at less than the average cost to produce, so let's take the average price, \$200. This takes into account the actual yield of the manufacturing process. Now accounting for the fact that (1) packaging and graphics hardware are likely to account for at least half of the cost, and (2) chips are not sold at cost, it is fairly conservative to say that the 32 nm die actually costs at most \$1 per mm^2. At this cost, the 42 tera-op, 7.4 × 7.4 cm ALA chip assembly costs about \$5500. There are, of course, costs associated with assembling bite-sized dies into bigger dies and packaging the bigger dies – but short of accounting for these, this figure does give a well-grounded ballpark estimate for the cost of materials necessary to manufacture the bare hardware being proposed here.

Comments In this section we have seen that fast, high-throughput matrix multiplication can be performed in hardware using ALA. Further, comparisons to today's consumer Intel chips indicate that the cost of this hardware is low considering the sheer quantity of computation it performs. The power density is high as I've presented it here, but an important note is imposing a speed constraint on any cell in the circuit adjusts the speed of the entire circuit. That is, if the application calls for consuming less power, or heat dissipation is difficult – limiting the rate at which data is flowed into the system can be used to adjust power consumption arbitrarily over many orders of magnitude. In the low-leakage limit, the energy consumed depends only upon the computation, not on how fast it is performed. Hence if there is any upper limit on power or heat dissipation imposed by the hardware or, say, power available, the computation can be performed more slowly on the same piece of hardware.

It is worth to note that it is surprising for an ALA chip to be at all competitive with a commodity AMD/Intel chip or supercomputer composed of them. It is a design composed of blocks of 30 transistors, which we compare with highly optimized designs of order 10^9 transistors, and come out a factor of 4 low on power consumption (Table 5.1a).

5.2 Simulating ALA

In order to benchmark ALA, we need to be able to burst update circuits (see Section 3.3.1), i.e. successively perform updates all-at-once updates of gates that are ready to fire. We also need to be able to count how many gates fire in a given interval, and accumulate data on the firing times of specific gates (inputs and outputs). Forrest Green, Peter Schmidt-Nielsen and I developed a software toolchain that performs these functions. The simulator back end that performs these functions, *librala,* is accessed through a front end written in Mathematica which orchestrats circuit assembly and execution in the back end.

The low-level representation for ALA circuits uses a set of bit layers to represent gate types, input directions, and token states in a rectangular grid of dimension according to the width and height of the circuit in ALA gates. This large rectangle is broken up into a grid of smaller rectangles called *patches* which have a width and height equal to 64. This way each component bit plane of a patch is represented as 64 successive words in memory. Then, using processor instructions for bit-wise boolean operations on words (such as bit-xor

[2]http://ark.intel.com/

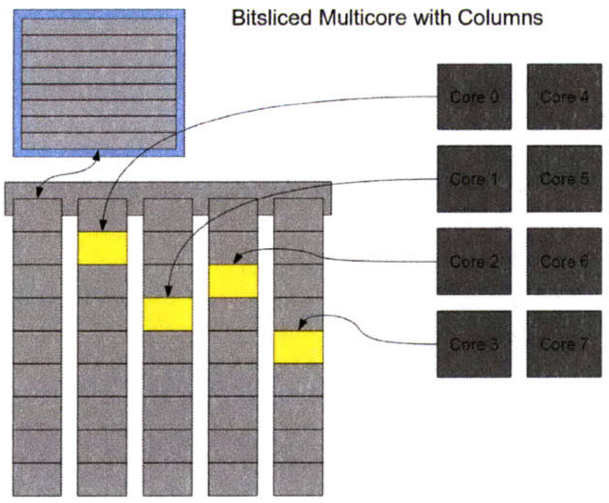


Figure 5.1: Update scheme used in *librala*. Credit: Peter Schmidt-Nielsen

between two input words), the simulator scans through each patch sequentially and updates the state of cells in it that are ready to fire. In order to prevent multiple propagations of a particular token in a single pass of scanning, two copies of the entire circuit are kept in memory, and updates are from one are written to the other, alternating back and forth. These update-scans can be parallelized by starting multiple threads at different points in the circuit. Since the logic of asynchronous updates enforces correctness, this procedure is guaranteed to produce a correct result as long as multiple threads don't try to update the same cells at the same time. On 8 cores, *librala* is able to update 1.3 Billion cells per second. Figure

5.3 Benchmarking ALA

In this section we'll see what the estimates of the Section 5.1 are based upon as well as much more detailed information about the execution of the ALA matrix multiplier circuit. In this section circuit metrics are defined in terms of cell updates and counting units of time in burst updates. It possible to derive transistor-level power estimates without simulating the full system at transistor level because assemblies of cells concatenate discretely - one cell updating in sequence with another cell takes twice the time and twice the power. Counting tokens is sufficient for computing power, and accumulating cell update times is sufficient for computing speed and latency. Only individual cells (roughly 30 transistors in design-in-progress at time of writing) need to be simulated using a physics engine in order to obtain atomic constants for individual cell types.

5.3.1 Definition of Metrics

In the case of matrix multiply, we have a multiple-in multiple-out circuit, so the circuit metrics selected reflect this.

Initial Phase Length

Initial phase length is computed by executing the circuit sufficiently long that equilibrium max-throughput state is achieved. Analytically this may be related to the maximum cycle length (more in Chapter 6) but we don't present any formal/rigorous results – data is experimental. There are two ways of measuring the initial phase length – one is to index it by bits – how many bits emerge before all subsequent bits a produced every second unit of time – and another is to index it by units of time. Because the ceasing of irregularities

label	description
ch	channels
bpW	bits per word
iniPh	initial phase length
eChT	channel throughput
eT	total throughput
iniBL	initial bit latency
iniWL	initial word latency
iniOpL	initial op latency
eChL	equil channel latency
eBL	equil bit latency
eWL	equil word latency
eOpL	equil op latency
enrB	energy per bit
enrW	energy per word
enrOp	energy per operation
eP	power in equlibrium

Table 5.2: Data Labels

in the consumption of bits at the inputs and such irregularities in the production of bits at the outputs does not generally coincide, we take the maximum of the two. It is conceivable that one or the other might be more important in some application, but for the purposes of this document we declare the maximum to be a useful figure for profiling the behavior of the circuits.

Explication of computation of metrics, listed in Table 5.2.

Latencies

There are a variety of latencies that can be measured – I'll run through the definition of the ones that I chose.

First let us consider latencies that apply to the initial phase of execution. One natural first question to ask is how many cycles pass between the time when the first bit is consumed on any input channel to when a significant output bit is produced on any output channel (iniBL). Next, since the production of bits can be quite irregular in the initial phase, one might ask the question of when the first word operation is complete – since in the case of operations on numbers we usually have a word length with which we are concerned (iniWL). In the case of a matrix multiplication, completion of the first operation might be defined by the last bit of the first matrix multiplication completed to be produced on any channel (iniOpL).

In the equilibrium phase of execution, we can ask similar questions – knowing the time index as well as the bit number corresponding to bits in and bits out, what is the time between when the earliest k-th bit goes in and when the latest k-th bit comes out (eBL)? What about the time between when the earliest k-th word has its first bit enter until the time when it has its last bit emerge (eWL)? Or the same question for the entire matrix – the time from first-bit-consumed until last-bit-emerges (eOpL)? Now, recall the fact that a matrix multiplier is a multiple-in-multiple-out circuit. We would like to characterize the relationship between the channels. One might expect the relationship to be very regular since the algorithm implements a systolic array – this is indeed the case, and the most pertinent question to ask is, in equilibrium, what is the latency between neighboring channels – that is, the time between when the k-th bit emerges on a channel and when the k-th bit emerges on neighboring channels (eChL).

Energy

We define the energy required to perform an operation as the total number of cells that fire between the initial state and when the final bit of the result emerges from the last channel.

Power

The power consumed by a circuit is the average number of gates that fire per cycle in equilibrium.

5.3.2 Data on ALA Matrix Multiplier

Data presented in Table 5.3.

bpw	dim	cbpo	iniPh	iniPhB	iniBL	iniWL	iniOpL	eBL	eWL	eOpL	eChL	enrOp	eP
16	2	32	536	246	300	330	362	358	388	420	43	96286	1418
16	4	64	796	278	497	569	665	666	696	792	185	734181	5604
16	8	128	1412	342	985	1057	1521	1282	1312	1536	469	5242624	22328
16	16	256	2644	592	1961	2033	3009	2514	2544	3024	1037	36193850	88525
16	32	512	5108	1104	3913	3985	5985	4978	5008	6000	2173	212958229	278293
32	2	64	895	432	451	513	577	528	590	654	30	310714	2376
32	4	128	1212	496	713	809	1001	970	1032	1224	210	2436715	9372
32	8	256	2096	624	1285	1437	2317	1854	1916	2364	570	17603578	37571
32	16	512	3864	1136	2541	2693	4597	3622	3684	4644	1290	117202637	136125
64	2	128	1634	816	746	872	1000	858	984	1112	1	1118608	4332
64	4	256	2140	944	1200	1326	1710	1562	1688	2072	251	8862036	17189
64	8	512	3436	1200	2108	2234	3881	2970	3096	3992	751	66604974	68640
64	16	1024	6252	2550	3924	4050	7721	5786	5912	7832	1751	413765260	217395

Table 5.3: Matrix Multiplier Data

Chapter 6

Graph Formalism and Algorithms for ALA[1]

In previous sections, the optimization of throughput was presented without comment – it was stated at the end of chapter 3 that throughput could be optimized so that bits on output channels are produced every second unit of time, and the circuits profiled in Chapter 5 were indeed optimized as such. In that chapter a number of observables were tabulated, but these have not yet been defined at a level of formalism that one would know precisely how to measure them from the content of this document [bad sentence]. In this chapter we build up a formalism within which we can both rigorously define the observables as well as understand how to optimize an important one of these, the throughput of a circuit, when sufficient conditions are fulfilled.

In the first section we will go over a formal definition of computation, then in the following section realization. In Section 6.3 we will see formal definitions of latency, throughput, energy, and power.

6.1 Formalism to Describe Computation [2]*

In order to be able to formulate rigorous descriptions of circuit behavior as well as algorithms for simulation and optimization, we need to formally describe an ALA circuit. This includes what happens when cells fire, as well as what a computation is.

6.1.1 Net, State, Update

We begin with a directed graph $N = (V, E)$. We call the nodes $v \in V$ gates/transitions/cells. With every v we associate a boolean function

$$f_v : \{0,1\}^{\text{in}(v)} \longrightarrow \{0,1\}^{\text{out}(v)}$$

that, given a full assignment on its incoming edges defines a new assignment on its outgoing edges. Edges function as 1-bit memory cells that can hold one value $\{0,1\}$ or be empty, which we denote by x. The state of an ALA circuit can thus be described by

$$S \in \{0,1,x\}^E.$$

[1]In this chapter Sections 6.1 through 6.5 were co-authored by Bernhard Haeupler with MIT CSAIL.
[2]coauthored by Bernhard Haeupler

During a computation the state of an ALA circuit is transformed by sequences of updates. Gates that have a value on all inputs and all outputs empty can update by deleting their inputs (assigning them to x), and assigning their outputs the values $f_v(i)$ Gates that fulfill

$$S_{\text{in}(v)} \in \{0,1\}^{\text{in}(v)} \ , \ S_{\text{out}(v)} = \{x\}^{\text{out}(v)} \tag{6.1}$$

are called *ready*, and we let $\mathcal{R}(S) = \{e \in E \mid e \text{ ready}\}$. We define an *update* (or *firing*)

$$\begin{aligned} &S \xrightarrow{v} S', \\ &S'_{\text{out}(v)} = f_v(S_{\text{in}(v)}) \ , \ S_{\text{in}(v)} = \{x\}^{\text{in}(v)}, \\ &S_{\overline{\Gamma(v)}} = S'_{\overline{\Gamma(v)}}, \end{aligned} \tag{6.2}$$

where $\Gamma(v) = \{(v_1, v_2) \in E \mid v_1 = v \text{ or } v_2 = v\}$ is the neighborhood function of a node. An example of an update is shown in figure 3.1.

An update $\xrightarrow{R}$, with $R \subset \mathcal{R}(S)$, can update more than one cell, and when we update all of $\mathcal{R}(S)$ we write $S \xrightarrow{\beta} S'$ (a *burst* update).

6.1.2 Computation

An input gate is a gate with no incoming edges with which we associate a sequence $\{0,1\}^*$ and an update

$$S \xrightarrow{v} S' \ , \ S_{\text{out}(v)} = \{x\}^{\text{out}(v)} \ , \ S'_{\text{out}(v)} = \{y\}^{\text{out}(v)}$$

where y is the next value in the sequence. We define an output gate as a gate with one incoming and no outgoing edges that records a sequence.

As such, we can view an ALA circuit as a function which transforms a set of input sequences into a set of output sequences:

$$C_A \left(\{0,1\}^*\right)^{\text{input gates}} \longrightarrow \left(\{0,1\}^*\right)^{\text{output gates}} .$$

A computation repeatedly performs updates if possible.

Theorem 1 (Determinism). *Every computation leads to the same output, i.e., the computation is deterministic.*

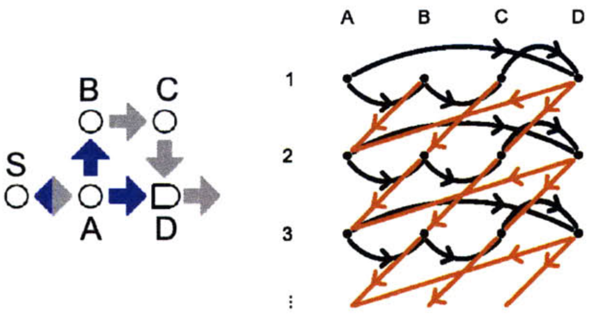
Figure 3.2 shows an example of a simple computation – a one-bit addition. In the figure we move from one state to the next by performing burst updates, i.e. by updating all gates are ready to fire with valid inputs and empty outputs. This semblance of synchronous updates is not to be taken too literally – in the abstraction of realization (see Section 6.2) each gate operates independently and asynchronously.

Now that we have an idea of what is meant by a computation in an ALA circuit, the question is, if this is really asynchronous, how can we characterize it? If we built a giant sea of asynchronous processors, could we characterize its behavior? Could we predict it?

6.2 Realization and Simulation [3]*

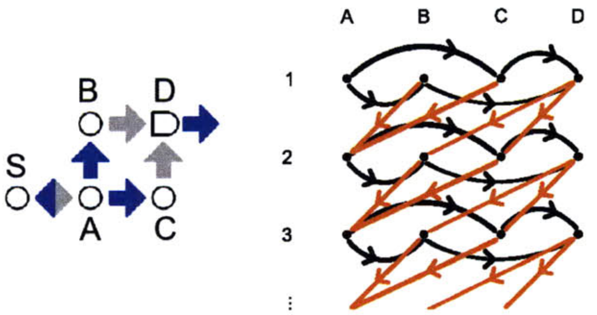
We begin by exploring how evaluation might occur in an ideal hardware circuit where each gate updates as soon as it is ready, and the operation of computing and passing tokens takes a total of precisely one unit of time. A complete characterization would assign each firing of a gate to some global time, so that we could in general say when the n-th gate fires for the k-th time.

[3]coauthored by Bernhard Haeupler



I	S	A	B	C	D
1	S_0				
2		A_0			
3	S_1		B_0		
4				C_0	
5					D_0
6		A_1			
7	S_2		B_1		

Figure 6.1: Low-throughput Example



I	S	A	B	C	D
1	S_0				
2		A_0			
3	S_1		B_0	C_0	
4		A_1			D_0
5	S_2		B_1	C_1	
6		A_2			D_1
7	S_3		B_2	C_2	

Figure 6.2: High-throughput Example

6.2.1 Dependency Graph

In order to construct such a characterization, we introduce a directed, infinite *dependency graph* $P_N = (V_P, E_P)$ to characterize the data token dependencies associated with each firing of each gate. Beginning with an ALA circuit $N = (V, E)$ with initial state S^0, we make a table of nodes in which columns correspond to nodes $v \in V$, and rows corresponding to firings. That is, for each node $v_0 \in V$ we have an infinite column of nodes $\{v_0^1, v_0^2, \dots\} \subset V_p$ in the infinite dependency graph representing each time v_0 fires in some realization/computation. Now we assign edges in E_p according to the structure of N. An edge $(v_i^k, v_j^l) \in V_P$ corresponds to the statements (i) $(v_i, v_j) \in E$ the edge set of the net N (ii) v_i must fire a k-th time before v_j can fire an l-th time.

Formally,

Definition 1. *For an ALA circuit $N = (V, E)$ with initial state S^0 we define the directed dependency graph $P_N = (V_P, E_P)$ by setting $V_P = \mathbb{N}^V = \{v_1^1, v_2^1, \dots, v_n^1, v_1^2, \dots, v_n^2, v_1^3, \dots\}$ and*

$$(v_i^t, v_j^{t'}) \in E_P \qquad \Longleftrightarrow \begin{cases} e = (v_i, v_j) \in E, & t' = t, \qquad S_e^0 = \emptyset \\ & \vee \\ e' = (v_j, v_i) \in E, & t' = t, \qquad S_{e'}^0 = \{T\} \\ & \vee \\ e = (v_i, v_j) \in E, & t' = t + 1, \quad S_e^0 = \{T\} \\ & \vee \\ e' = (v_j, v_i) \in E, & t' = t + 1, \quad S_{e'}^0 = \emptyset \end{cases} .$$

6.2.2 Realization

There are two ways we'll treat realization – one is in the general setting where gate update times may vary – by type, over time, randomly, etc., and the other is the special case where all gates take exactly one unit

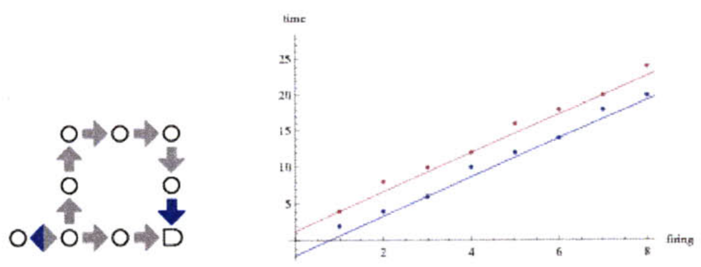


Figure 6.3: Throughput and Latency Example

of time to fire. By restricting the general case to the special case, it follows that performing burst updates properly reflects the discrete times at which ideal hardware updates occur. This establishes that we know how to simulate in a way that faithfully reflects "real" execution – an important link in the ALA circuit design tool chain.

Using the dependency graph V_p, we can construct "realizations" by successively removing (analog to "firing") nodes with no incoming edges. We can assign a global time of evaluation to nodes by the following procedure. We begin by assigning $t(v_i^0) = 0$ for each node v_i which is ready in state S^0. Now, whenever we remove a node v_r (which always has only outgoing edges), we consider each node v_d which has an incoming edge from v_r. We let $t(v_d) = max(t(v_d), t(v_r) + \Delta t)$, where $\Delta t = 1$ in the case at hand. When the node v_d is removed by this procedure, the value $t(v_d)$ corresponds to the longest path (or in general: path of maximum weight) to v_d from any node v_i^0 which is ready in S^0. But this is also precisely the evaluation time in the parallel asynchronous model, if we assume that we can begin evaluation of all ready nodes simultaneously at $t = 0$. Even without this assumption, as long as we can characterize the time when asynchronous evaluation begins we can add weighted edges to V_p to capture this behavior. This result is stated more formally in Appendix A.

6.3 Metrics [4]*

We are interested in characterizing computations by their throughput, latency, and energy consumption. Roughly speaking, that is, how fast outputs are produced, how much time it takes between when a given input goes in and when the output corresponding to that input comes out, and how many updates are performed to produce a bit of output. Note that taken strictly the definitions given below apply only to circuits without copy and delete gates. Modified versions which accommodate token creation/destruction are conceptually very similar, but require more detail.

Definition (Throughput). The *throughput* $T(v)$ of a vertex in an ALA circuit N with initial state S^0 is defined

$$T(v) = \lim_{k\to\infty} \frac{k-1}{t(v^k) - t(v^1)}.$$

That is to say, the throughput of a gate is the asymptotic number of firings per unit time, which is must lie in the interval $[0, 1/2]$. For example, in a sequence of wire gates fed by a stream of tokens from an input gate, each gate fires once for every two units of time – making the average firings per unit time equal to $1/2$.

Definition (Latency). We can define **path latency** (of a path) as the asymptotic average time a token spends traversing the path. A useful working definition for **gate min latency** between an input x and an output y is the minimum path latency taken over all $x - y$ paths.

[4]co-authored by Bernhard Haeupler

Multiple path latencies between a single set of endpoints (x, y) indicate (by relative offset) which emerging bits at y are logically dependent upon bits entering x. If any (x, y) path intersects a feedback loop, then there is an infinite sequence of emerging tokens that are logically dependent upon any given entering bit. Observe that for systems with maximal throughput, the gate min latency is equal to the path length.

In Fig 6.3, the lower set of points represents the firing times of the gate at the lower left-hand corner of the square in the configuration, and the upper set of points is the firing times of the gate at the lower right-hand corner. The slope of the lines is inverse throughput T^{-1}, and the vertical space between the lines is the latency between the said gates.

Definition (Energy). Each gate in a circuit fires once for each output token produced. Therefore we define circuit **energy** as

$$E = |V| .$$

If a circuit has multiple output gates $\{outs\}$, we say that **energy per bit B** equals

$$B = \frac{E}{|\{outs\}|} .$$

6.4 Calculation of Throughput[5]

We now observe that throughput and latency are defined asymptotically, and therefore direct numerical evaluation would take infinitely long. Therefore we wish to develop a framework for direct and finite computation of these quantities. We do this by projecting the infinite DAG into a finite cyclic dependency graph. We can then formulate throughput as a property of this finite graph which lends itself to efficient computation.

The finite dependency graph N' is generated by projecting the nodes of the infinite graph N_P back into the nodes N. Those edges which in N_p connect nodes of different firing indices are called $1 - edges$, and those that connect node within a single time index are $0 - edges$. See Figure 6.4 which illustrates the finite dependency graphs for the examples in Figures 6.1and 6.2.

Now, for any cycle C in N', we can define the *value* of the cycle $val(C)$ as the quotient

$$val(C) = \frac{\#\text{ 1-edges}}{\text{cycle length}} .$$

Observe that for any directed cycle with $val(C) = v$, traversing the cycle in reverse order gives $val(-C) = 1 - v$.

We claim that the throughput $T(N)$ is given by

$$T(N) = \min_{C \in N'} val(C) .$$

Note that the maximum throughput of a net is $T = 1/2$. This follows from the facts that any cycle with $val(C) = v > 1/2$, we have $val(-C) = 1 - v < 1/2$ giving us $T(N) < 1/2$. Maximum throughput is achieved when every cycle has value $1/2$.

Observe that N' can be easily generated from N by considering all edges with state x to be 0-edges, and all edges with tokens to be 1-edges, and inserting edges in the opposite direction of the opposite type – 1-edges opposing 0-edges and 0-edges opposing 1-edges. This is illustrated in the figures.

[5]co-authored by Bernhard Haeupler

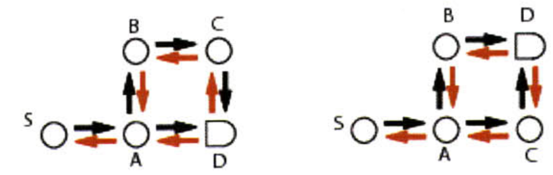


Figure 6.4: Finite Dependency Graphs

The dependency DAG is a helpful tool to derive theoretical bounds on the computation time and behavior. In the following we want to develop algorithms that determine and optimize the throughput of a network. For algorithms the infinite size becomes a problem. Since the dependency DAG has a very regular structure. Only subsequent levels are interconnected and the connections are the same on every level. We create an infinite dependency DAG that exactly captures this structure of the dependency DAG but has size $O(n)$. Essentially we map each vertex v^t back to a vertex v. We distinguish two types of edges, those who go from vertices v^t to a vertex w^t and those which go "forward in time" e.g. to w^{t+1}. We call these edges 0-edges and 1-edges respectively. The following theorem shows that at least for the matter of paths there is a nice mapping between paths in the infinite and the finite DAG.

Remark:
The finite dependency graph can be easily obtained directly from the original network N. For this take each edges e of N and reverse them iff $S_e^0 = \{T\}$. These are the 0-edges. The 1-edges are exactly the reverse of all 0-edges.

Definition 2. *For every $i \in \mathbb{N}$ and every directed path $P' = v_1, v_2, \dots, v_k$ in the finite dependency graph P'_N we define a directed path $P = v_1^{i_1}, v_2^{i_2}, \dots, v_k^{i_k}$ in P_N by setting $i_1 = i$ and $i_{j+1} = i_j + k$ if (v_j, v_{j+1}) is a k-edge in P'_N. Note that (as long as we don't have parallel edges) this mapping gives indeed a path in P_N and is furthermore bijective. We define the value of path P as $\frac{i_k - i_1}{\text{length of } P}$. For a path P' this translates into a value of the sum of the time values of the edges in P' divided by the length of the path.*

Let $\mathcal{C}(v)$ be the set of cycles through v in P'_N and $\mathcal{C}'$ be the set of all node disjoint cycles in P'_N. Note that $\mathcal{C}'$ is finite while $\mathcal{C}(v)$ is not.

Theorem 3. *(If N is connected then,)*

$$T(v) = \inf_{C \in \mathcal{C}(v)} \operatorname{val}(C) = \min_{C \in \mathcal{C}'} \operatorname{val}(C)\,.$$

Proof.
From the convergence in the definition of T we get that

$$\begin{aligned} T(v) &= \lim_{i \to \infty} \frac{i}{\max\limits_{P \text{ path to } v^i \text{ in } P_N} \operatorname{len}(P)} \\ &= \lim \inf_{P \text{ path to } v^i \text{ in } P_N} \operatorname{val}(P). \end{aligned}$$

This path consists according to Corollary 7 of a prefix of at most n updates that do not involve v. With $i \to \infty$ the length of this prefix does not change the asymptotics and we can instead assume a path from A_1 to A_i in P_N. Each such path corresponds to a cycle in P'_N containing v and the value of these cycles is preserved. This proofs the first equation. For the second equation we start by showing that that if a non node disjoint cycle C of value p exists than there is a node disjoint cycle C' of value at most p. For this take such a cycle C and decompose it into a collection $C_1, C_2, \dots$ of cycles from $\mathcal{C}'$ with length $l_1, l_2, \dots$ and value $\frac{i_1}{l_1}, \frac{i_2}{l_2}, \dots$. We have $\operatorname{val}(C) = \frac{\sum_k i_k}{\sum_k l_k} \geq \min_k \frac{i_k}{l_k} = \min_k \operatorname{val}(k)$. This proofs $\inf\limits_{C \in \mathcal{C}(v)} \operatorname{val}(C) \geq \min\limits_{C \in \mathcal{C}'} \operatorname{val}(C)$. To proof equality we give a series of cycles $C_t \in \mathcal{C}(v)$ whose values converge against the value of the minimum value cycle $C \in \mathcal{C}'$. Since N is connected there is a path P from v to a node in C. The cycle C_t we choose consists of P, t copies of C and finally P reverse. These cycles are in $\mathcal{C}(v)$ and their value for increasing t is dominated by the length and value of the t C-loops. This gives $\lim_{t \to \infty} \text{val}(C_t) = \text{val}(C)$ and thus $\inf_{C \in \mathcal{C}(v)} \text{val}(C) \leq \min_{C \in \mathcal{C}'} \text{val}(C)$. $\square$

The following dynamic program efficiently computes the quantity $T(v)$ given P'_N with integral length / delays δ. Let E' be the edge set of P'_N:

Initialization

$\forall e = (v, w) \in E' : f(v, w, \delta(w)) = 0$

$\forall e = (v, w) \in E' : f(w, v, \delta(w)) = 1$

Recursion

$f(w, v, l) = \min_{(z,v) \text{ is a } k\text{-edge in } E'} f(w, z, l - \delta(v)) + k$

Output

$\min_{v,l} \frac{f(v,v,l)}{l}$

Running time

$O(nm \sum_v \delta(v))$ since for every possible lengths from 1 to $\sum_v \delta(v)$ from each of the n nodes every of the m edge gets used twice.

The problem of calculating the minimum value cycle in a directed graph with arbitrary length and k-edges with arbitrary k is known as the minimum cost ratio cycle problem (MCRP). It has been intensely studied and several efficient algorithms are known. The best asymptotic running time is $O(mn \log n)$ (or $O(mn + n^2 \log n)$ using Fibonacci-Heaps). The best theoretical and practical algorithm due to [Young, Tarjan, and Orlin, 1991], which in practice runs in nearly linear time. The simplest algorithm which is also very fast in practice is due to [Howard, 1960]. For experimental evaluations and implementations see: [Dasdan, 2004], [Dasdan et al., 1999].

Perturbation of Throughput

Corollary 4. *Let $T(v)$ updates per round be the throughput of a gate v in the β-update model. In the timed model the throughput of v is at least $T(v)/t$ if every gate is guaranteed to update within t time after being ready. Earlier updates can only increase the throughput.*

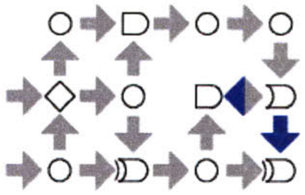


Figure 6.5: 12-gate adder with throughput-limiting cycle highlighted.

6.5 Optimization of Throughput: Algorithmic [6]*

We want to compute an optimal buffer strategy for a given ALA circuit. This means we want to assign every edge a length (i.e. number of buffers) that optimizes throughput. For this we give every edge a variable length $l_e \geq 1$ and find a feasible point with respect to the constraint that no cycle in the network with new length has a value smaller than the desired throughput λ. This is equivalent to looking for negative length cycles when one transforms an k-edge with length l to a length $k - \lambda l$ edge. If λ is not known one can do a binary search on it until an optimal feasible point is found. Since the lengths in the network are linear functions in the l_e variables these are linear constraints. Even so, the number of cycles and hence the number of constraints can be exponential and a feasible point (if one exists) can be found (up to an ϵ-slack) in polynomial time (e.g by using the ellipsoid algorithm). As a separation oracle for the ellipsoid algorithm, Bellman-Ford (or the faster algorithm by Tarjan) can be used – finding a negative cycle if one exists. By solving the linear program with the same feasible set and the objective min $\sum_e l_e$ we can find a solution that achieves optimal throughput, and in addition minimizes the number of buffers needed (caveat: this is not completely true since buffering solutions can be fractional).

6.6 Optimization of Throughput: Examples By Hand

6.6.1 Adder

The adder which was presented in chapter 4 consumes power $P = 12$, and has throughput $T = 3/8$. The cycle that limits the throughput of this adder is highlighted in Figure .

With a slightly different adder design, this figure shows a circuit before and after performing a 4-bit addition at maximum throughput.

By using 14 gates we can achieve maximal throughput $T = 1/2$, shown in Figure 6.6. Recall that this corresponds to the property that any connected cycle of nodes in the finite priority graph is balanced between 0- and 1-edges (not to be confused with edges occupied by 0- and 1- tokens).

6.6.2 Multiplier

In this section I present one way to construct a maximum-throughput multiplier. Two primary changes are made from the design presented in Figure , the first involving the serial/parallel stage, and the second involving the add stage.

In order to optimize the serial/parallel stage, the first step is related to the contrast shown between Figures 6.1 and 6.2. In this examples it is made apparent that in cases where shortest geometric paths are taken between fan-out and fan-in points, and there are an equal number of tokens in each branch, cycles are

[6]coauthored by Bernhard Haeupler

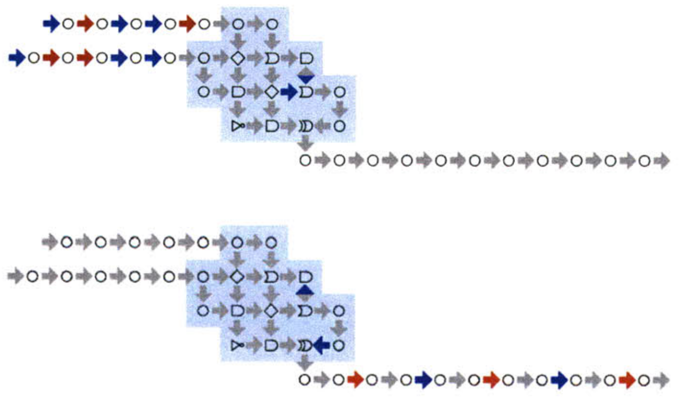


Figure 6.6: 4-bit addition, before and after computation, showing max-throughput

"automatically" buffered correctly. This is a sufficient condition, but not a necessary one for optimally buffering cycles. The first step that was taken to optimize the serial/parallel conversion module/stage in the multiplier was to conform to this principle and layout the primary stage axis diagonally instead of vertically. Since there are tokens distributed along the control line, compensating buffers must be introduced. Because of the additional geometric constraint that the outputs of the delete gates must be unobstructed to the east, these compensating buffers must be made "too long", and a counter-compensating buffer introduced on the opposite branch of the cycle. These are highlighted in Figure 6.7.

Now, the next buffering problem involves balancing cycles introduced in the addition stage. Because each adder has two inputs, a cycle is introduced by each, which enters through one input and returns out the other. By introducing a tree structure, we reduce the number of buffering problems to be solved from $O(n)$ to $O(log(n))$. For conciseness complete analysis is given for powers of two, although the strategy presented here is valid for any n. In this case a geometric placement strategy allows a closed form solution for placing adders when moving to a successive power of two.

A final change made to the multiplier which we use in the final version of the matrix multiply is to use sequence generators as in Figure 4.8a. These start saving area with $n = 16$, so they are present in Figure 6.8, but not in 6.7.

6.6.3 Matrix Multiplier

The pattern is probably clear by now – the circuit presented in Chapter 4 was suboptimal – and the repairs presented in the previous two sections aren't enough to achieve maximal throughput in the matrix multiplier. Fortunately there is only one cycle that needs to be balanced with the insertion of a buffer. Figure **??** shows where this cycle is in the functional block diagram, and highlights where the adjustment is to be made. Figure 6.8 highlights where the buffer was inserted.

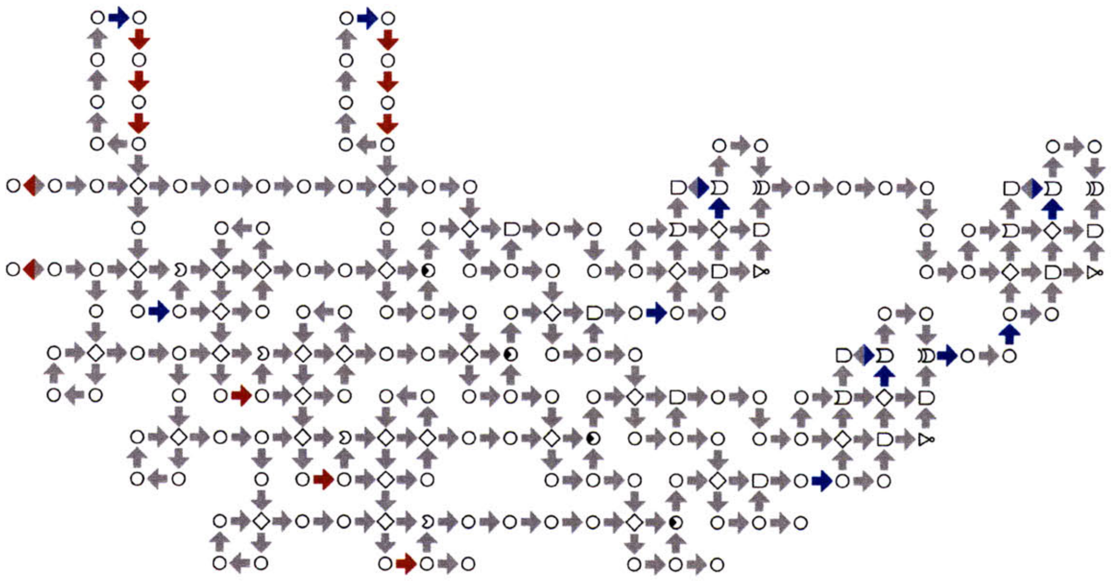


Figure 6.7: ALA High-Throughput Multiplication

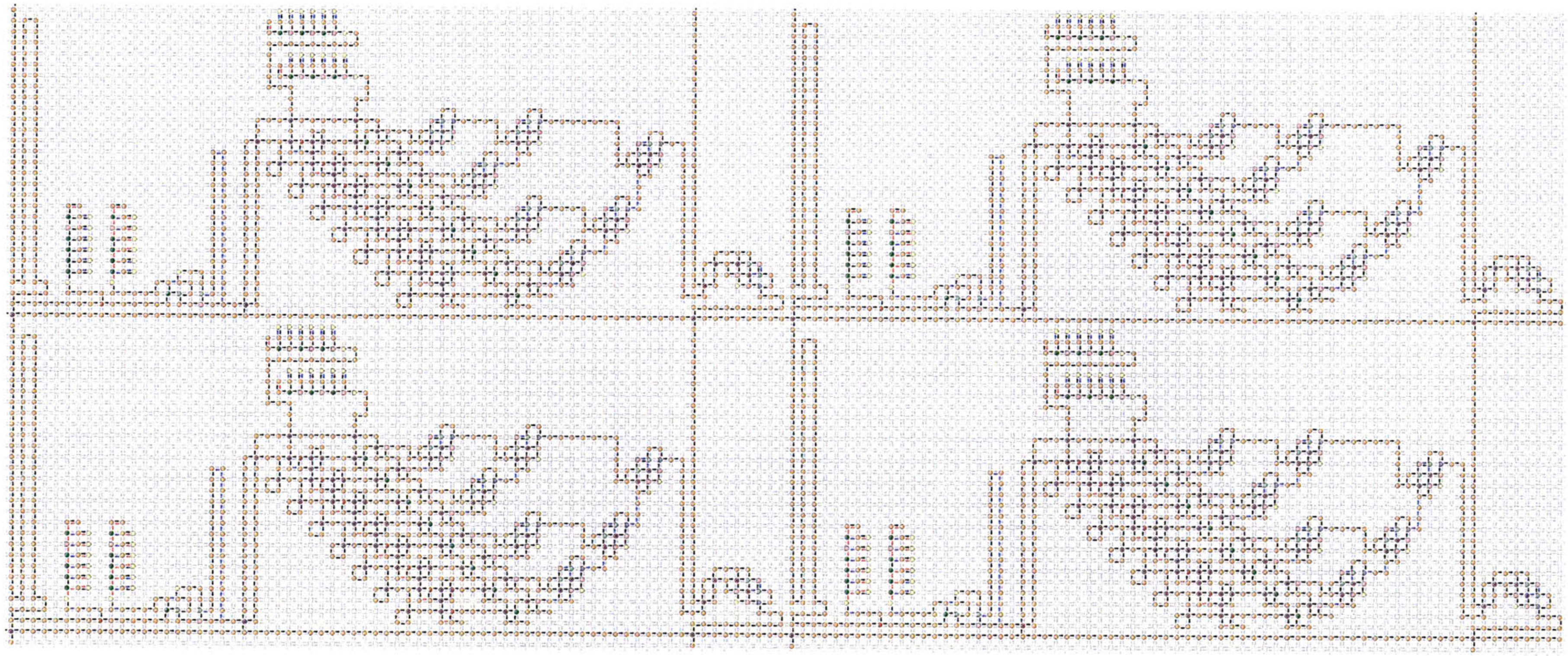


Figure 6.8: Pipelined 2 $\times$ 2 matrix multiply in ALA

Chapter 7

A Guide to "Snap": Hierarchical Construction with Performance Guarantees

In Chapter 6 we saw two ways of optimizing throughput – one was a global optimization algorithm, and one was "by-hand," seeking out unbalanced cycles and creatively constructing one-off solutions. There were two reasons to solve by-hand even with knowledge of a global algorithm. One is that the global algorithm didn't know how to embed in the plane, and the other is that the global algorithm provided no assistance in generating parametric designs. Through hand-analysis it is possible to consider what happens when some parameter is incremented – whether the increment preserves optimality of all cycles.

In this section we'll look at a method for *constructive optimization*. This is a constructive procedure that generates optimal circuits so long as the input specification satisfies certain constraints. A large number of powerful and useful algorithms can be formulated this way, while many others cannot. There are two important features of this procedure, which will be referred to as *DAG-flow construction*. The first is that it is a hierarchical procedure – the procedure is iterated, beginning with small modules and iteratively making larger modules out of modules. The second is that the algorithm is a forward construction procedure that never requires backtracking to lower levels in the hierarchy. Certain limited information is preserved after each module construction, and the rest is "black-boxed" – no longer needed to carry out construction at higher levels.

7.1 Hierarchy and the Abstraction of Modules

Specifying programs hierarchically is a good top-down way for programmers to mentally manage the task of designing very complex programs. In ALA one side effect is that it provides a template for building circuit optimality from the bottom up.

Our throughput algorithms were previously adapted to operate on cell-level circuit descriptions and compute global properties. Since the complexity of the algorithm grows as $O(n^2)$, it is not practicable to apply this method as circuits get large. One way of managing this is to carefully assemble circuits that use regularity in a parametric way, so as to use small global computations to provide guarantees that are invariant under parametric scaling (optimization by-hand). This works for something like an integer multiplier that scales in a very regular way. Another much more flexible and generally applicable way to manage the complexity of performance characterization as circuits scale is to abstract functional modules. In this way a knowledge of internal details of component modules is not necessary in order to characterize a circuit assembled from many modules. Only a small amount of information (linearly proportional to the number of inputs and

Figure 7.1: Merging Complete Graphs with Glue

outputs of the module) needs to be assembled in order to compute our latency and throughput metrics for these aggregate circuits.

One advantage of abstracting modules is that performance guarantees can be maintained without complexity blowing up. Another equally significant win is the ability to abstract. On one hand we want to expose the programmer to the costs of transporting data which are physically unavoidable. On the other hand, we want the programmer to be able to abstract function away from geometry, since the starting point of writing a program is a concept of what the program should do, and not how it should occupy space. A particular procedural abstraction we call DAG-flow allows the programmer a natural abstraction while tacitly preconditioning geometry and building in the connection between functional proximity and geometric proximity that is necessary to respect physical scaling.

Circuits must be assembled to minimize the complexity of buffering for optimal throughput. Given an unordered rat's-nest of modules and connections, a variety of heuristics could be applied in order to achieve optimal throughput. We observe, however, that programmers are unlikely to think in unordered rat's-nests – we expect it to be much more natural to assemble modules from submodules, and place use these larger modules in combination with other modules to accomplish intermediate and global goals.

7.2 The Induction Step: Making a Module out of Modules and Preserving Performance

We saw in Chapter 6 that the property of a cycle that can limit throughput is "unbalanced" cycles in the finite priority graph. In order to do incremental connection/merging of circuit "modules," what we need to show is that there is a way of guaranteeing that any new cycles that are formed by the merging are able to be "balanced" by construction. Now it will suffice to show that any two modules can be connected up – all larger mergings can be constructed from these single steps.

Let M_1 and M_2 be two max-throughput modules that we'd like to connect together to form a new module that also has max-throughput. The modules are specified with inputs and outputs, and the merging procedure is defined to connect some outputs of one module – say, M_1 – with inputs of another module – now M_2. This way, all of the connections between modules go one way. Critically, it cannot be the case that some connections also connect the outputs of M_2 with the inputs of M_1.

For each module we have potential values associated with the nodes, which are constructed at the lowest level by computing the shortest path metric[1], and are then updated at each aggregation step. Now, within a single module it is generally the case that there is some path between any input or output port p_1 and any other one p_2. Let's assume this is the case – if not, a module can be considered as more than one module – so that this assumption eventually holds. Using the potential values $\phi(p_1)$ and $\phi(p_2)$ we can deduce the cost/weight of the path between p_1 and p_2. Now suppose that p_1 and p_2 are outputs of M_1 and q_1 and q_2 are inputs of M_2 and we want to connect p_1 with q_1 and p_2 with q_2. This connection generates a node cycle in the graph (p_1, p_2, q_2, q_1), which can also be considered as an edge cycle (p, x_2, $-q$, $-x_1$) , labeled as in Figure 7.1. The problem of buffering is thus formulate as choosing weights of x_1 and x_2 that both balance the cycle – i.e. have the property that $p + x_2 - q - x_1 = 0$ – and can be geometrically embedded in ALA. The critical observation to make about this formula is that, since p and q have the same direction, if you increase the length of p the sum becomes more positive, and if you increase the length of q the sum becomes more negative – that is – whatever the initial state of the geometric embedding is, it can be perturbed so as to balance the cycle. If the sign of both were the same, this wouldn't be the case – we might have a negative sum and no knob to turn to balance it – since "shortening" edges is not always possible.

[1]see, for example [Ahuja et al., 1993]for more on shortest path / potential values

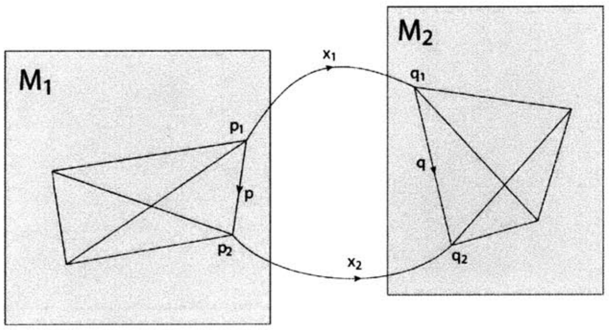


Figure 7.1: Merging Complete Graphs with Glue

Now that we've seen how to balance the cycle created by a pair of edges that join two modules – now what happens when we bring a third (or a k-th) edge into the picture? Now that M_1 and M_2 are joined, weighting for the edge from p_3 to q_3 is imposed by the prior connections made. If, however this weight corresponds to a path which is shorter than the geometry allows (is geometrically infeasible), then we need to buffer the other edges (the first $k-1$) in order to achieve the zero sum necessary. Let x_3 be the shortest path which is geometrically feasible. In the weighting equation, buffering other edges corresponds to introducing a factor Δ as follows

$$\begin{aligned} p + x_3 - q - x &> 0, \\ p + x_3 - q - (x + \Delta) &= 0, \end{aligned} \tag{7.1}$$

and if it is necessary to extend x_3 in order to buffer the other edges x , another factor γ can be added to all since Eq. 7.1 implies that

$$p + (x_3 + \gamma) - q - (x + \Delta + \gamma) = 0. \tag{7.2}$$

The trick that makes utilizing this fact particularly easy in ALA is that introducing a factor γ to all prior edges (p_i, q_i) can be accomplished by translating the module M_2 in the plane and extending the edges with shortest paths along the grid. Adding Δ is slightly less trivial, but can be accomplished with a variety of constructive heuristics.

7.3 DAG-flow: The Anatomy of a Sufficient Framework

In this section I go through the most important mechanisms necessary for implementing the promised hierarchical low-complexity forward construction procedure with performance guarantees.

7.3.1 Input Syntax and Assembly Procedure

In section 4.4 it was mentioned that connections between modules in *Snap* circuits are constrained to be top-to-bottom and left-to-right. The reason for this constraint is that it allows for a constructive procedure whereby any circuit constructed from max/high-throughput submodules is guaranteed to also be high-throughput, based on the result from Section 7.2. Using the primitives discussed in 4.4, I define a syntax for specifying a circuit or module as a series of connected stages in which data flows from left to right and top to bottom.

```
new_module = modCons[{{A,B,...},glue[{n1,n2},{n3,n4},...],C,D,...}]
```

In DAG-flow we specify a circuit construction procedure that alternates between vertical and horizontal concatenations. We begin by stipulating that individual modules have their inputs on the left and outputs on the right – arbitrarily choosing an axis. We introduce two primitive operations – (i) producing module columns by vertical concatenation of modules that operate in parallel, and (ii) horizontal concatenation of sequences of columns that are connected unidirectionally. Without loss of generality, we assert that these connections be directed from left to right, and specify them as permutations that map outputs of one column to inputs of the next. These are not strictly permutations since we allow fanout of single outputs to multiple inputs.

Once we have a specification of such a circuit, we begin the layout process with a pass of assembling columns in which we equalize width. Then, beginning by fixing the left-most column at the x=0, we add columns one-by-one, connecting them with smart glue, which takes as input potential values associated with the outputs and inputs in question, and buffering properly. This procedure could also be performed in parallel in a logarithmic number of sequential steps – pairing and gluing neighboring columns preserves the ability to optimally buffer connections between the larger modules thus assembled. In order to augment the class of circuits that can be defined using DAG-flow, we also introduce a rotation operator, which allows us to assemble sequences of columns, then rotation them and assemble columns with these as modules.

7.3.2 Construction Primitives and Module Objects: Inheritance and Derivation of Attributes

Concatenation In section 4.4 we saw that the horizontal and vertical concatenation operators, `hcat` and `vcat`, were critical for enabling the specification of basic spatial relationship between modules. When implemented in the language *Snap*, the operators operate on module objects – that is, named data structures that can be queried for various attributes. In this case important attributes are width, height, and the location of inputs and outputs. The product of a concatenation of modules is a new module, and this module's attributes must be derived from those of its component modules. The contents of such a *mat* module is a list of component modules and their offsets relative to the origin of the new module. For example, if I `hcat` together two modules M_1 and M_2 each of width $= 5$, the new module M_3 has an attribute $\texttt{mat}[\texttt{M}_3]$ with value $\{\texttt{M}_1, \{\texttt{0},\texttt{0}\}, \texttt{M}_2, \{\texttt{5},\texttt{0}\}\}$ and $\texttt{width}[\texttt{M}_3] = \texttt{10}$. For horizontal inputs from the west and outputs to the east, the rule is that for `hcat` the inputs of the left-most module (M_1 in the example) become the inputs of M_3, and the outputs of the right-most module become the outputs of the new module. The vertical input and output attributes of M_3 are concatenated from those of M_1 and M_2. Analogous rules apply for `vcat`. If implementing a node-potential scheme based on the theory in Section 7.2, the concatenation operators also accumulate node potential values. New node potential values must only be assigned to nodes that are inputs or outputs of the new module. If we assume that concatenation is applied to a pair (the general case can be derived by repeated application of pair-concatenation), and assume that throughput has been optimized, the potential an output node r of M_3 can be computed as follows: select any output p of the left module M_1 along with its corresponding input of M_2. The potential of r is the potential of p plus the weight of the one-unit path from p (which equals $1/2$ when throughput is maximal), plus the potential difference between r and q.

Rotation The rotation operator can be used to derive vertical operations from horizontal ones. It derives attributes in a straightforward way similar to what's described for `hcat` and `vcat`. Because of the constraints on direction of dataflow only two rotation operators are necessary – a rotation `rotateW` that operates on modules with inputs to the west and rotates 90 degrees clockwise, and a rotation `rotateN` that operates on modules with inputs to the north and rotates 90 degree counterclockwise. One example of a derived attribute for `rotateW` would be that vertical inputs of the new module are the horizontal inputs of the old module. One slight non-triviality is that, to enforce module specification with the origin at the lower left (or wherever one chooses), translation must be performed after rotation. In the next section we'll see that rotation is useful for adapting a west-east glue generating routine to generate north-south glue.

Smart Glue Glue modules are treated by the concatenation operators just as any other module. However, in the construction procedure, smart glue generation has a chance to peek ahead in time to prepare the glue to buffer optimally. Stages are constructed first so that glue modules always receive as input a left module and a right module. Some algorithm is applied to generate an optimally buffered glue module which is geometrically compatible with the left and right modules, and this module is then horizontally concatenated with its neighboring modules.

At the time of writing, the glue algorithm in use is not "smart" – it has the capability of joining inputs and outputs along shortest paths and any additional adjustment necessary has thus far been done "by-hand" as demonstrated in 6.6. The procedure I propose is to begin with a solution that satisfies geometric embedding, and then perturb it within the embedding until the cycle is balanced. The nature of some of the heuristics to fully implement this procedure – hence making the glue "smart enough" – are suggested in section 7.2, but none of these have yet been worked out in full detail. The only thing that makes creating optimal buffering at all tricky is the fact that "bumps" inserted into wires to buffer them may collide with existing structures – which requires modifying either the existing structure, or not getting the buffering you want in the new buffer. When you can't have the buffering that you compute by Equation 7.1 it is usually necessary to overshoot, as in Equation 7.2, and modifying counter-balancing segments to compensate. This was the case in with the multiplier example in Section 6.6.2. In that example though we weren't afforded the luxury of aligning the inputs and outputs vertically so as to allow the application of generic strategies that apply to this situation.

Chapter 8

Conclusion

I have taken you on a tour of what happens at the limit of parallelism – this is one scheme where logic and interconnect are blended together in a small space for high-density, high-throughput computation, illustrating the benefits of aligning physics and computation. These benefits are a coherent sum of tried and true strategies: use parallelism, avoid unnecessary data transport, and customize your hardware to your application. The HPC-On-A-Chip application approaches customization as quick customization: the process of building ALA in hardware is a generic process – assembling tiles in a grid – but it's custom because the program maps one-to-one onto hardware and hence optimizes in hardware the particular program it represents. The other context for the performance figures represented here is reconfigurable hardware - RALA. Bringing this into the realm of practicality will require more development and inventiveness, but it ultimately promises most of the benefits of HPC-On-A-Chip without the need to physically assemble a new chip for each new program.

Recall that the biggest problems with HPC today are power-consumption, ease of design and ease of programming. It's not enough to be competitive on power – and we saw that ALA holds promise as a substrate for an intuitive and scalable language where programming is done more as spatial assembly of blocks than as specifying sequences of steps – it is visual data-flow language like Simulink, but instead of large functional blocks and continuous time, ALA constructs universal logic beginning with single-bit operations placed in discrete spatial locations, and operates asynchronously. The language automatically optimizes throughput using results from graph theory. Making the program bigger means changing a parameter and building a bigger or a smaller piece of hardware, and not performing a global redesign.

In this thesis I focused on high-throughput applications for doing large quantities of highly-pipelined operations. This takes advantage of asynchrony as an enabling feature for extensibility, but there's a family of advantages that apply in other parts of the programming parameter space. In particular, in programs with a low-level, spatially and temporally irregular activity pattern, the asynchronous property of ALA can be beneficial because parts of the grid that aren't performing computation don't consume power. So a computation can be a sparse network of varying tasks, and each task is optimized for speed and power efficiency at some different location on the grid. One application for this type of computation would be to process sparse audio-video data where computation only happens when data comes in, and the procedure performed on the data may be data dependent, so that different parts of the chip are activated by different data.

These preliminary results warrant further exploration of ALA as a paradigm for hardware and software. In order to try out more sophisticated numerical computations, such as matrix decomposition, a next step is to implement some LAPACK-like routines. In parallel, there are a number of important questions to address. For example, I posited that circuits can be packed very tightly, but have not yet explored packing algorithms. Also, the automatic throughput maximization prohibits feedback loops, but there are cases where feedback loops can be optimally pipelined. One example is the serial adder - the circuit contains a 1-bit feedback loop that processes the carry bit. Also, the selectCopy circuit presented in Chapter 4 contains a feedback loop and is maximally pipelined. A worthy topic of further investigation is whether logic transformations that perform this pipelining algorithmically can be identified and generalized. Another important topic for further exploration is fault tolerance - majority voting has been shown to be sufficient for correcting logic errors, but currently we don't have a mechanism for recovering a circuit when a gate fails to produce a token. There are a variety of strategies out there, and these should be inspected for applicability to ALA.

I have shown that scalability and performance can be achieved in the ALA model. I attribute the success of the aligned strategies of parallelization, minimization of data transport, and customization to their collective similarity to physics. For completeness it deserves to be mentioned that there is one very important property of physics that we haven't yet brought into ALA - and that is reversibility. The effortless way in which nature computes using chemical and thermodynamic processes – contrasted with our many-kilowatts – suggests that we're still doing something very differently. In quantum mechanics, processes are reversible and no information is lost. The gates presented in Section 3.1 perform irreversible logic - the inputs that went into a computation cannot be recovered through knowledge of the outputs, and this implies dissipation of heat (see Feynman, 2000). Embodying reversible operations in an ALA-like hardware architecture has the potential to harness all of the other benefits of aligning physics and computation and in addition cut power consumption by orders of magnitude. At present, however, it isn't clear how to engineer it – that will require some very deep invention.

Appendix A

Formal Results

A.1 Stability under asynchronous updates

Update Sequences

Theorem 5. *The possible update sequences for the ALA circuit (N, S^0) are exactly the topological orderings for P_N where v_i^t means that the gate v_i gets updated the t-th time.*

Proof. An edge $e = (v_i, v_j)$ starts with $|S_e^0|$ tokens and every update of v_i increases the number of tokens on e by one while every update on v_j consumes one token from e. Thus the number of tokens on an edge $e = (v_i, v_j)$ if v_i has updated t times and v_j has updated t' times is exactly $|S_e^0| + t - t'$. At any given point a transition v can update iff all its input gates are filled with one token and all its outputs are empty. Therefore v^t can happen iff any input gate w has already updated exactly $t_i = t - |S_{(v,w)}^0|$ times and every output gate u has updated exactly $t_o = t - 1 + |S_{(v,w)}^0|$. The dependency graph P_N contains by definition for every v exactly the edges (w_{t_i}, v_t) and (v_t, w_{t_i+1}) for an input gate and the edges (u_{t_o}, v_t) and (v_t, u_{t_o+1}) for an output gate. This guarantees that v^t occurs in every topological order between the t_i-th and the $(t_i + 1)$-th update of any input gate and the t_o-th and $(t_o + 1)$-th update of any output gate. This proves the claim. Finally note that for every gate v in N v^t occurs before v^{t+1} in any topological ordering of P_N since for every t there are edges $(v^t, u^{t'}), (u^{t'}, v^{t+1})$ where u is a neighbor of v in N. □

Corollary 6. *An ALA circuit N is deadlock free iff P_N is acyclic.*

Since we are only interested in deadlock free ALA circuits we will from here on call the directed dependency graph also dependency DAG.

Corollary 7. *If the ALA circuit N is connected then every gate no gate updates more than n times more than any other gate. Thus every gate has the same throughput, i.e. $T(v) = T(v')$ for any v, v' in N.*

Proof. Since N is connected there is a path of length at most n from v to v' in N. This leads to a directed path from v^t to a $v'^{t'}$ with $t' \leq t + n$. Since this is true for any ordered pair v, v' this proofs that between any gates in N the number of updates can differ at most by n. With time to infinity the throughput of any two nodes is thus the same. □

Realization

With these tools we can now prove a very general theorem about the behavior of an ALA circuit in a model that measures time and does not rely on β-updates. For this we assign each v^t update of a gate v a time delay $\delta(v^t)$ and assume the following general update model. If a gate v is ready to fire the t-th time because all its inputs are full and all outputs clear it updates with a delay of $\delta(v^t)$.

Theorem 8. *The time when a gate v updates the t-th time in the above model is exactly the length of the longest directed path to v^t in the dependency DAG P_N where the length of a path is the sum of the delays δ of the nodes on the path.*

Proof. We prove this by induction on the order in which updates occur. The first update v^1 gets updated at time $\delta(v^1)$ since v is ready and does not have to wait for any other updates to happen. For the induction step we look at the time of an update v^t which happens $\delta(v^t)$ after the occurrence of the last neighborhood update v had to wait for clears an output of provides an input. Let's call this update $u^{t'}$ and assume it happens at time x. By induction hypotheses there is a directed path to $u^{t'}$ of length x. This path can be extended to give a path to v^t of length $x + \delta(v^1)$ which is exactly the time v^t happens. It remains to show that there is no longer path to v^t. Such a path would consist of a path to a transition $u'^{t''}$ in the neighborhood of v of length x' and than an edge to v^t. Since this path is supposed to be longer we have $x' > x$ and again by the induction hypothesis this means that the transition $u'^{t''}$ occurred after $u^{t'}$. This contradicts the choice of $u^{t'}$. □

Note that this is setting is quite general. It allows to model different update timings for different gates or gate types or the introduction of stochastic or perturbed update delays. The β-update model is also captured as the special case of all delays δ being 1.

Appendix B

Benchmarking Code

This code is written in *Mathematica.* Commands that refer to "webs" access a back end written in C/C++ that simulates ALA circuits.

```
dataList = {bpw, dim, dim, cbpo,iniPh, iniPhB,nU, iniBL, iniWL, iniOpL, eBL, eWL,
   eOpL,  eChL, enrOp, eP};
dataElementNames = ToString /@ dataList;
gatherMatMulData[dim_, logBpw_,itr_:10,fnSuppl_:""] :=
    Module[ {leftIn, topIn, mx, mxLeftInCrs, mxTopInCrs, mxVOutCrs, iniPhOuts,
    iniPhIn1s, iniPhIn2s, fullIniPhData, cbpo, lastFirstOutputTime,
    lastFirstWordOutputTime, lastLastOutputTime., firstOutputTime,
    firstFirstWordOutputTime, firstFirstChOpOutput, iniOpL, iniBChL,
    iniWChL, iniOpChL, upperLatIndB, lowerLatInd1, lowerLatInd2,
    eOpL, eChL, bpw, firstInputTime, iniBL, iniWL, lowerLatInd,
    eBL, eWL, dat1, ePWhole, eP, lambda, nU,
    strMat, dat, outFireTimesIni, in1FireTimesIni, in2FireTimesIni,
    iniPhB, iniPh, in1FireTimesEq, in2FireTimesEq, outFireTimesEq,
    data, writeMatMulData, out, mt, oSeq, repIn, vRepIn, iMatCCV,
    vZero, iMatCCHV, lastOutInd, lastOutCr,
    enrOp, enrInL, enrInT, ePLeft, ePTop, ll, lll, eChLL
    },
        (*initialization/parameters*)

        lambda = 1/2;
        bpw = 2^logBpw;
        cbpo = dim bpw;(*channel bits per operation*)
        nU = dim bpw itr;


        (*COMPUTE INITIAL PHASE LENGTH*)

        (*construct mat and compute coordinates*)
        {strMat, leftIn, topIn, mx} =
        strMatMulHT[dim, logBpw, True];
        mxLeftInCrs =
         innerMatWebCrs[strMat, mx, inputs, #] & /@ Range[Length[inputs[mx]]];
        mxTopInCrs =
         innerMatWebCrs[strMat, mx, vInputs, #] & /@ odds[Range[Length[vInputs[mx]]]];
        mxVOutCrs =
 innerMatWebCrs[strMat, mx, vOutputs, #] & /@ evens[Range[Length[vOutputs[mx]]]];
(*output times*)
outFireTimesIni = {};
Do[resetMatWeb[strMat];
   dat = execMatCoor[strMat, oCr, nU];
   AppendTo[outFireTimesIni, odds[dat]], {oCr, mxVOutCrs}];
iniPhOuts = pickEquilibriumBegin /@ outFireTimesIni;
(*left input times*)
in1FireTimesIni = {};
Do[resetMatWeb[strMat];
   dat = execMatCoor[strMat, iCr, nU];
   AppendTo[in1FireTimesIni, odds[dat]], {iCr, mxLeftInCrs}];
iniPhIn1s = pickEquilibriumBegin /@ in1FireTimesIni;
(*top input times*)
in2FireTimesIni = {};
Do[resetMatWeb[strMat];
   dat = execMatCoor[strMat, iCr, nU];
   AppendTo[in2FireTimesIni, odds[dat]], {iCr, mxTopInCrs}];
iniPhIn2s = pickEquilibriumBegin /@ in2FireTimesIni;
fullIniPhData = {Thread[{mxLeftInCrs, iniPhIn1s}],
  Thread[{mxTopInCrs, iniPhIn2s}],
  Thread[{mxVOutCrs, iniPhOuts}]};
iniPh = Max[#[[2]] & /@ Join[iniPhIn1s, iniPhIn2s, iniPhOuts]];
iniPhB = Max[First /@ Join[iniPhIn1s, iniPhIn2s, iniPhOuts]];


(*COMPUTE LATENCIES*)

(*INITIAL PHASE*)

firstInputTime =
 Min[First /@ in1FireTimesIni, First /@ in2FireTimesIni];
lastFirstOutputTime = Max[First /@ outFireTimesIni];
(*this is time of last channel to fire first bit \ *)
(*minus time of first firing of first input*)
iniBL = lastFirstOutputTime -
  firstInputTime;
lastFirstWordOutputTime = Max[#[[bpw]] & /@ outFireTimesIni];
iniWL = lastFirstWordOutputTime - firstInputTime;
lastLastOutputTime = Max[#[[cbpo]] & /@ outFireTimesIni];
iniOpL = lastLastOutputTime - firstInputTime;
firstOutputTime = Min[First /@ outFireTimesIni];
iniBChL = lastFirstOutputTime - firstOutputTime;
firstFirstWordOutputTime = Min[#[[bpw]] & /@ outFireTimesIni];
iniWChL = lastFirstWordOutputTime - firstFirstWordOutputTime;
firstFirstChOpOutput = Min[#[[cbpo]] & /@ outFireTimesIni];
iniOpChL = lastLastOutputTime - firstFirstChOpOutput;

(*EQUILIBRIUM PHASE*)

(*drop firings from initial phase to get firings in equilibrium phase*)
{in1FireTimesEq, in2FireTimesEq, outFireTimesEq} =
Map[(Drop[#, iniPhB - 1]) &, {in1FireTimesIni, in2FireTimesIni,
  outFireTimesIni}, {2}];
```

```
upperLatIndB = Max[computeLatInd /@ outFireTimesEq];
lowerLatInd1 = Min[computeLatInd /@ in1FireTimesEq];
lowerLatInd2 = Min[computeLatInd /@ in2FireTimesEq];
lowerLatInd = Min[lowerLatInd1, lowerLatInd2];
eBL = upperLatIndB - lowerLatInd;
eWL = eBL + 2*(bpw - 1);
eOpL = eBL + 2*(cbpo - 1);
eChL = Max[computeLatInd /@ outFireTimesEq] - Min[computeLatInd /@ outFireTimesEq];


(*COMPUTE ENERGY*)

(*compute by giving one set of inputs and BurstUpdateUntilDone.*)
mt = mx;
vZero = rotateWMod[eWire0];
oSeq = Append[rep[0, bpw - 1], 1];
repIn = hcatMat[{dataWireMod[oSeq], copyWord[dim, logBpw]}];
vRepIn = rotateWMat[repIn];
{topIn, iMatCCV} =
 matConsStageVAlign[Riffle[rep[vRepIn, dim], rep[vZero, dim]], mt,
  True];
{leftIn, iMatCCHV} =
 matConsStageAlign[rep[repIn, Length[inputs[mt]]], iMatCCV, True];
(* then need to subtract off the power of the input mats *)
derCellCount;
(* which output to monitor for completion of single execution *)
lastOutInd = Position[outFireTimesIni, lastLastOutputTime][[1, 1]];
lastOutCr = innerMatWebCrs[iMatCCHV, mt, vOutputs, 2 lastOutInd];
(* factor of two is from "evens" in def of mxVOutCrs *)
dat1 = execMatCoor[iMatCCHV, lastOutCr, cbpo];
enrOp = GetCellUpdateCounter[];
derCellCount;
execMat[leftIn, cbpo];
(* use first output as stopping condition because I know all channels are identical *)
enrInL = GetCellUpdateCounter[];
derCellCount;
execMatCoor[topIn, First[webVOutputs[topIn]], cbpo];
(*execMat[topIn, cbpo]; *)
enrInT = GetCellUpdateCounter[];
enrOp = enrOp - enrInL - enrInT;


(*COMPUTE POWER*)

(*this computation relies on averaging over a sufficient number of output bits*)
resetMatWeb[strMat];
BurstUpdateWeb[web[strMat], iniPh]; resetMatWeb[leftIn];
BurstUpdateWeb[web[leftIn], iniPh]; resetMatWeb[topIn];
BurstUpdateWeb[web[topIn],iniPh]; derCellCount;
BurstUpdateWeb[web[strMat], 2 bpw];
ePWhole = GetCellUpdateCounter[]; derCellCount;
BurstUpdateWeb[web[leftIn], 2 bpw];
ePLeft = GetCellUpdateCounter[]; derCellCount;
BurstUpdateWeb[web[topIn], 2 bpw];
```

```
    ePTop = GetCellUpdateCounter[];
    eP = (ePWhole - ePLeft - ePTop)/(2*bpw) // N;
    (*the only reason two firings isn't sufficient is copy/delete cycles*)
    out = {bpw, dim, dim, cbpo, iniPh, iniPhB, nU, iniBL, iniWL, iniOpL, eBL, eWL, eOpL,
      eChL, enrOp, eP};
    data = Thread[{dataElementNames, out}];
    writeMatMulData :=
        Export[dataFileName["matMul", bpw, dim, dim,fnSuppl] ,data];
    Print[writeMatMulData];
    data
];
```

Appendix C

Example Layout Code

C.1 Periodic Sequence generation

```
onesGates = {
{{0,0},wire,{{N,1}}},
{{0,1},wire,{{S,x}}}
};

dublGates = {
{{0,0},copy,{{W,x},{N,x}}},
{{0,1},wire,{{N,x}}},
{{0,2},not,{{S,0}}},
{{1,0},and,{{W,x},{N,x}}},
{{1,1},wire,{{N,0}}},
{{1,2},not,{{S,x}}}
};

dublOnesGates = {
{{0,0},copy,{{W,x},{N,x}}},
{{0,1},wire,{{E,x}}},
{{1,0},wire,{{W,x}}},
{{1,1},wire,{{S,0}}},
{{2,0},xor,{{W,x},{N,x}}},
{{2,1},wire,{{W,x}}},
{{3,0},and,{{W,x},{N,x}}},
{{3,1},wire,{{W,x}}}
};

apOsc[n_] :=
    Module[ {bd (*binary digits*),mls (*module list*)},
        bd = IntegerDigits[n,2]; (* binary representation of n *)
        bd = Drop[bd,1]; (* first significant bit is implicit *)
        mls = bd /. {1->Sequence[dubl,dublOnes],0-> dubl};
        (* generate module list from binary representation *)
        mls = Prepend[mls,ones]; (* source of ones is one-bit base case *)
        hcatMat[mls] (* generate mat module *)
    ];
```

```
evenCopies[logN_,m_] :=
    hcatMat[{apOsc[m],Sequence@@rep[straightDubl,logN]}];
```

Bibliography

The Green Grid, 2010. URL `http://www.thegreengrid.org`.

R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, New Jersey, 1993.

Edwin Roger Banks. *Cellular Automata.* PhD thesis, Massachusetts Institute of Technology, 1970. URL `http://hdl.handle.net/1721.1/5853`.

K. Chen. Circuit Design for Logic Automata. Master's thesis, Massachusetts Institute of Technology, 2008.

Kailiang Chen, Forrest Green, and Neil Gershenfeld. Asynchronous logic automata asic design. *to be submitted*, 2010.

Wu chun Feng and Kirk W. Cameron. Green500. URL `http://www.green500.org`.

John Conway. The game of life. *Scientific American*, 1970.

International Business Machines Corporation. Ibm highlights, February 2007. URL `http://www-03.ibm.com/ibm/history/documents/pdf/2000-2006.pdf`.

David Dalrymple. Asynchronous Logic Automata. Master's thesis, Massachusetts Institute of Technology, 2008.

Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):385–418, 2004. ISSN 1084-4309. doi: http://doi.acm.org/10.1145/1027084.1027085.

Ali Dasdan, Sandy S. Irani, and Rajesh K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 37–42, New York, NY, USA, 1999. ACM. ISBN 1-58133-109-7. doi: http://doi.acm.org/10.1145/309847.309862.

Jack B. Dennis. Modular, asynchronous control structures for a high performance processor. *ACM Conference Record: Concurrent Systems and Parallel Computation*, 1970.

Jack B. Dennis. A preliminary architecture for a basic data-flow processor. *Project MAC Report*, 1980a.

Jack B. Dennis. Data flow supercomputers. *IEEE Computer*, 1980b.

GJ Olsder JP Quadrat F Baccelli, G Cohen. *Synchronization and Linearity: An Algebra for Discrete Event Systems.* Wiley, 1992.

Richard P. Feynman. *Feynman Lectures On Computation.* Westview Press, 2000.

Neil Gershenfeld, David Dalrymple, Kailiang Chen, Ara Knaian Forrest Green, Erik D. Demaine, Scott Greenwald, and Peter Schmidt-Nielsen. Reconfigurable asynchronous logic automata (rala). *ACM POPL'10*, 2010.

F. Green, K. Chen, A. Knaian, and N. Gershenfeld. Asynchronous Logic Automata ASIC Design. *manuscript*, 2010.

Forrest Green. ALA ASIC: A Standard Cell Library for Asynchronous Logic Automata. Master's thesis, Massachusetts Institute of Technology, 2010.

Simon Haykin, John Litva, Terence J. Shepherd, T.V. Ho, J.G. McWhirter, A. Nehorai, U. Nickel, B. Ottersten, B.D. Steinberg, P. Stoica, M. Viberg, and Z. Zhu (Contributor). *Radar Array Processing.* Springer Verlag, 1992.

R. A. Howard. *Dynamic Programming and Markov Processes.* The M.I.T. Press, Cambridge, Mass., 1960.

Advanced Micro Devices Incorporated. Amd history, 2010. URL `http://www.amd.com/us/aboutamd/corporate-information/Pages/timeline.aspx`.

Martha Mercaldi Kim, Mojtaba Mehrara, Mark Oskin, and Todd Austin. Architectural implications of brick and mortar silicon manufacturing. *SIGARCH Comput. Archit. News*, 35(2):244–253, 2007. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1273440.1250693.

D.E. Knuth. *The Art of Computer Programming, v.3 Seminumerical Algorithms.* Addison-Wesley, 3rd edition, 1998.

Rajit Manohar. Reconfigurable asynchronous logic. *Integrated Circuits Conference*, 2006.

N Margolus. An embedded dram architecture for large-scale spatial-lattice computations. In *Proceedings of the 27th International Symposium on Computer Architecture*, New York, NY, USA, 2000. ACM.

A.J. Martin, A. Lines, R. Manohar, M Nystroem, and P Penzes. The design of an asynchronous mips r3000 microprocessor. *Advanced Research in VLSI*, 1997.

Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. Top500, June 2010. URL `http://www.top500.org`.

Claudio Moraga. On a case of symbiosis between systolic arrays. *Integr. VLSI J.*, 2(3):243–253, 1984. ISSN 0167-9260. doi: http://dx.doi.org/10.1016/0167-9260(84)90044-0.

David Patterson. The trouble with multicore. *IEEE Spectrum*, July 2010.

C.A. Petri. Nets, time, and space. *Theoretical Computer Science*, 153:3–48, 1996.

NVIDIA Corporation, 2010. URL `http://www.nvidia.com/page/corporate_timeline.html`.

Stephen Wolfram. *A New Kind of Science.* Wolfram Media, 2002.

Neal E. Young, Robert E. Tarjan, and James B. Orlin. Faster parametric shortest path and minimum balance algorithms, 1991.