Asynchronous Logic Automata for Large-Scale Circuits and Systems

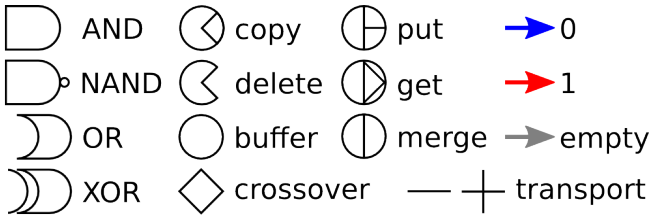
W911NF-09-1-0542
Final Report
May 15, 2011

Introduction

This seedling seeks to evaluate the impact of asynchronous logic automata (ALA) on the scaling of large-scale circuits and systems. ALA aligns computational and physical descriptions, with cells that asynchronously pass and process state tokens. The distance that a token travels equals the amount of time it takes, the number of operations that can be performed, and the amount of information that can be stored. These resources are coupled as they are in the underlying physics. ALA programs are portable across any process technology that can do the cell updates; if tokens are passed correctly locally, the system will operate correctly globally.

ALA

ALA is based on passing state tokens between locally-connected cells on a lattice, where the interconnect is 2D or 3D based on the actual dimensionality of the underlying hardware. A link can be empty, or contain a token that is true or false. When cells have valid tokens on their inputs and empty tokens on their outputs they pull the former and push the latter. In the cell set used here, AND, NAND, OR, and XOR are included for logic (to ease design), and tokens are manipulated with cells for creating, destroying, switching, buffering, and transporting tokens.



ALA has been implemented for multicore clusters, embedded microcontrollers, and custom silicon. RALA adds reconfigurability; the focus here is on understanding the performance and programming of ALA once configured.

Relationships

ALA is best understood not as a new model, but rather as the asymptotic intersection of a number of familiar models:

- it's multicore with a processor for every logical operation
- it's a form of dataflow, with each gate a graph node
- it's RISC, with the instruction set reduced to logical operations
- it's a gate array with a single operation in the logic block, and nearest-neighbor interconnect
- it's an ASIC with a one-to-one mapping between programs and circuits
- it's a Petri net where the network is a lattice

- it's cellular automata where logic is the rule table and global clocks are not needed
- it's asynchronous logic with the asynchronous dependencies implicit in the gates
- it's a systolic array where the data flow is arbitrary
- it's a form of paintable or amorphous computing that takes advantage of the regularity of periodic fabrication processes

Benefits

Below we show that ALA can be implemented at about one third the density of synchronous logic or SRAM. In return for this overhead it provides universal communication, computation, and storage that is:

- Efficient: linear-time algorithms include sorting and matrix multiplication
- Parallel: no thread scheduling
- Portable: no technology dependencies
- Homogeneous: no built-in bottlenecks, fixed resource allocation, or hot spots
- Asynchronous: no static power (other than leakage)
- Flexible: no static architecture
- Scalable: no system scale factors
- Extensible: no system granularity
- Verifiable: representation is maintained from high-level program to low-level hardware
- Reliable: no global yield requirement
- Fast: no global clock, propagation at gate delay

Hardware

In this project, the design of a library of ALA cells based on handshaking for token passing was completed and modeled in a standard 90nm CMOS process. With a typical energy per operation of 0.1 pJ and latency of 0.1 ns, these compare favorably with current best practices in both gate arrays and HPC systems.

This initial ALA design had a space overhead of 50-100 transistors per cell. An effort to reduce that resulted in a second-generation design that has the structure of an SRAM memory, with the global address lines replaced with local logical connections sensing tokens. The SRAM states are used to represent the token presence and value, and the cell firing condition. This approach brings the overhead down to tens of transistors per cell, approximately 1/3 the density of static memory or synchronous logic alone.

One immediate application is in a "pick-and-place" ASIC design workflow, with a one-to-one map between an ALA program and an equivalent ASIC. Because the only design rules are local, the chip will operate correctly if the program does. This simplification of ASIC workflows is one of the primary goals of the seedling in addressing the ~$100M cost of a large-scale full-custom design.

There is an even more significant implication for fabrication, which we hope will lead to the ability to produce an ASIC in an afternoon (instead of a year). All possible 2D ALA cell types and connections can be represented by roughly 20 tiles, each with about 20 transistors. As an alternative to the overhead added by electrical reconfigurability (at least an order of magnitude for the multiplexers and state storage), we are investigating the feasibility of taking advantage of complementary work on digital fabrication to be able to do high-throughput placement of these tiles on a regular substrate to additively assemble an ASIC. We anticipate in future work prototyping this possibility with chip-scale discrete components, taping out test chips with the universal tile set, investigating die singulation,

placement, and interconnect, and studying programming models for their physical reconfiguration.

Software

Work on ALA algorithms has progressed in quantifying and optimizing throughput and latency, and in implementing mathematical methods for high-performance computing. Efficient implementations of ALA arithmetic were developed, followed by a library of core BLAS (Basic Linear Algebra Subprograms) routines. Current work is using these to implement the LU decomposition for solving systems of equations.

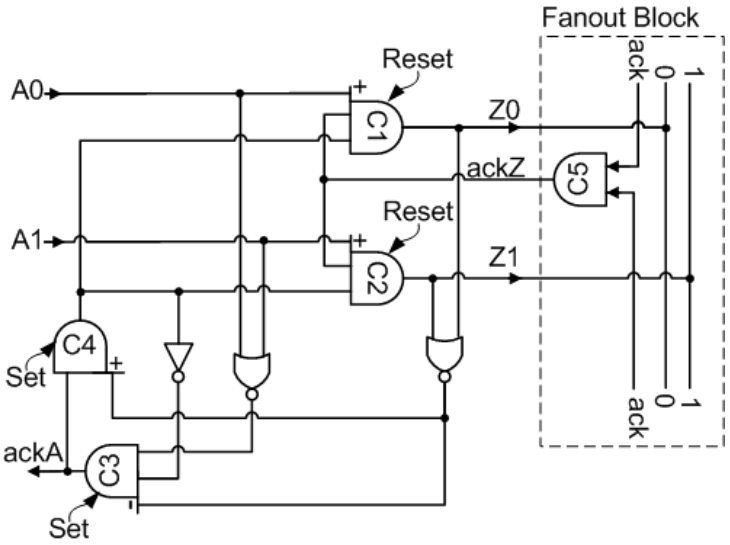
The programming effort is supported by work on design tools, both graphical and textual, maintaining the equivalence of program constructs and ALA circuits. An efficient parallel simulator has been developed for debugging and benchmarking the performance of larger programs. These token counts can be combined with transistor-level cell simulation to predict system performance, with initial power estimates up to an order of magnitude better than today's largest systems.

These results are described in detail in the following sections on Circuits, Algorithms, Tools, Conclusions, and Publications.

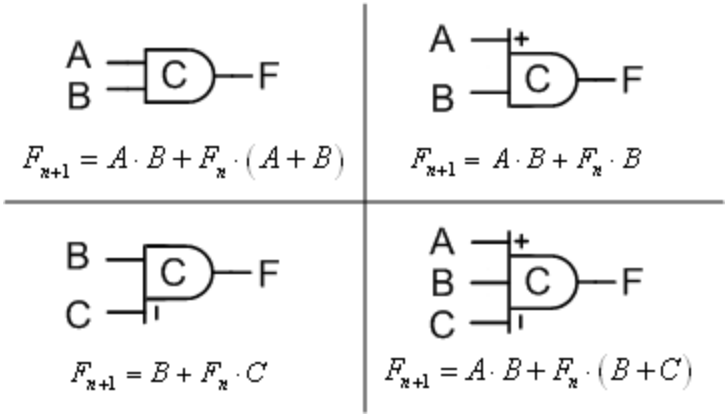
Circuits

Standard Cell Library with Initial Cell Design

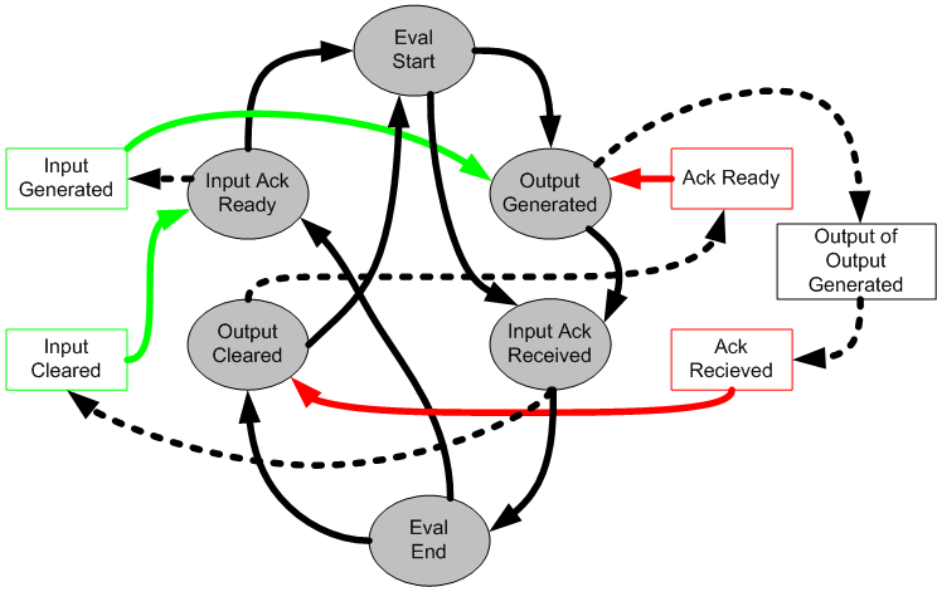
The initial cell design was based on a precharged full buffer (PCFB) as shown here.



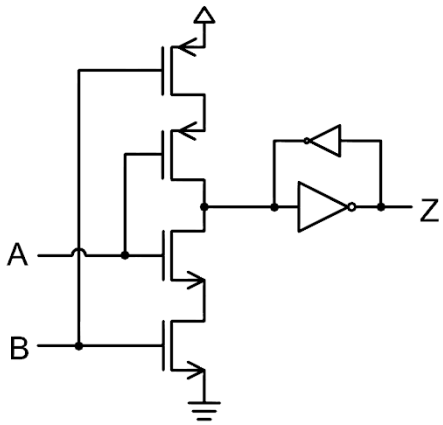
This approach uses Muller C-elements to hold state and synchronize signals. Logical functions and the corresponding signals are shown below for some typical C-elements.



The following is a state diagram for the PCFB. Each of the ovals represents a transition on one of the C-elements inside the buffer. For example, "Eval Start" represents C4 in the PCFB schematic transitioning from high to low. Boxes represent transitions in neighboring cells, e.g. "Input Generated" indicates when either C1 or C2 in the previous cell have transitioned from low to high. Solid black arrows represent internal dependencies and colored arrows represent dependencies on external transitions. The two arrows pointing in to the "Output Generated" node indicate that this event must occur after "Eval Start" (C4 going high) and "Input Generated" (either C1 or C2 from the input cell) have occurred. Dashed arrows represent dependencies of external transitions on local events. Each of these arrows has some delay value associated with it. The time for any event can be modeled as the maximum of any input event plus the delay from the connecting arrow.

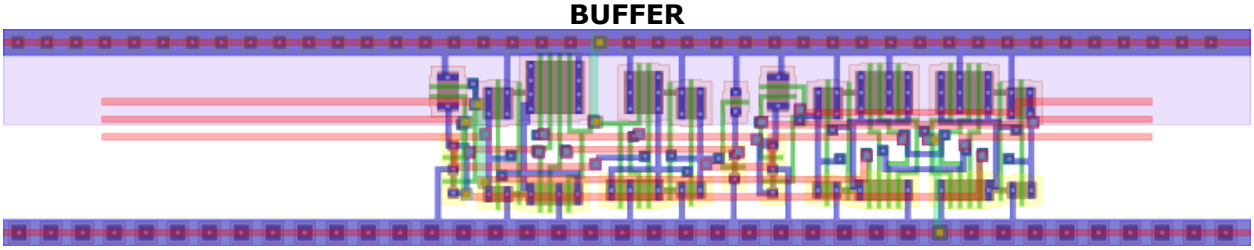


The C-elements in the initial design use a dynamic logic style with keepers. The two-input C-element implementation is shown here.

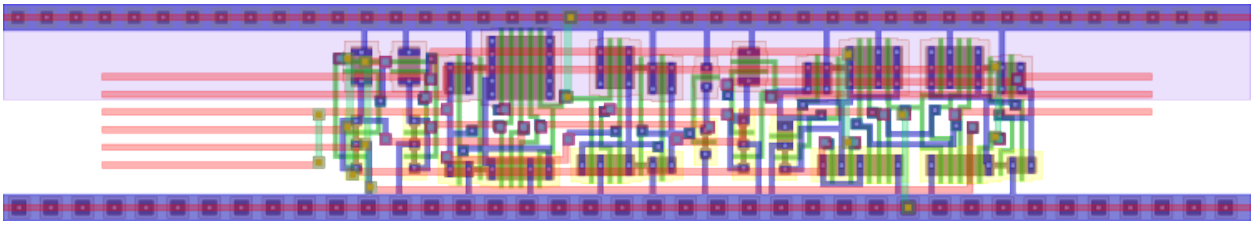


The following six figures show layouts of cells in our initial design in a generic 90nm process. By placing inputs and outputs in fixed locations we were able to directly add tiles to select inputs and outputs (i.e. input A comes from the north). Because they must be placed on a grid, each of these cells occupy the same area of 122 μm^2. This size is limited by the buffered crossover. When placed on the grid, cells are used in vertically flipped versions that share adjacent N-wells, power rails, ground rails, set connections, and reset connections.

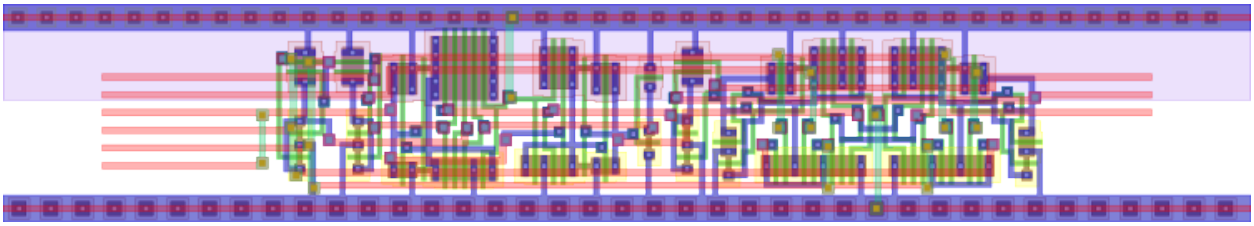
BUFFER



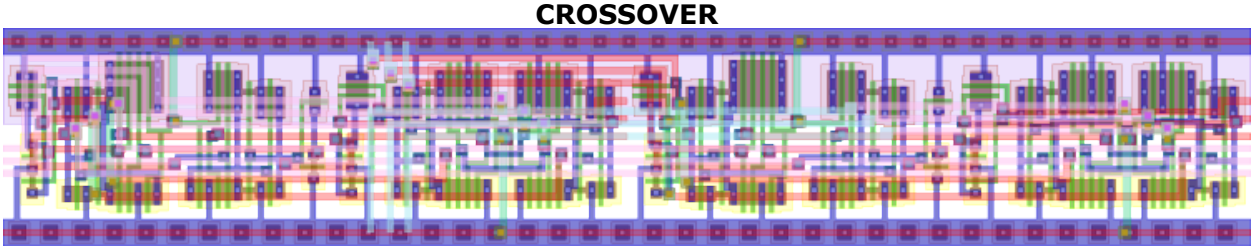
NAND



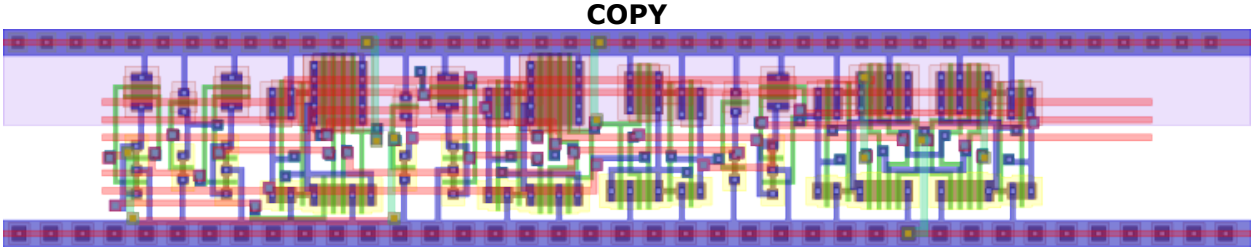
XOR



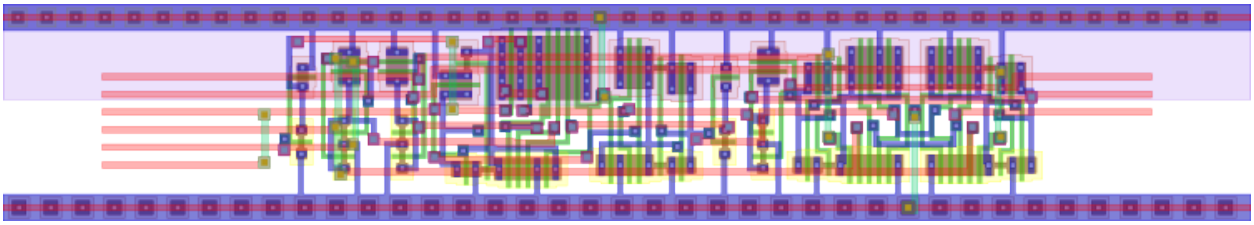
CROSSOVER



COPY



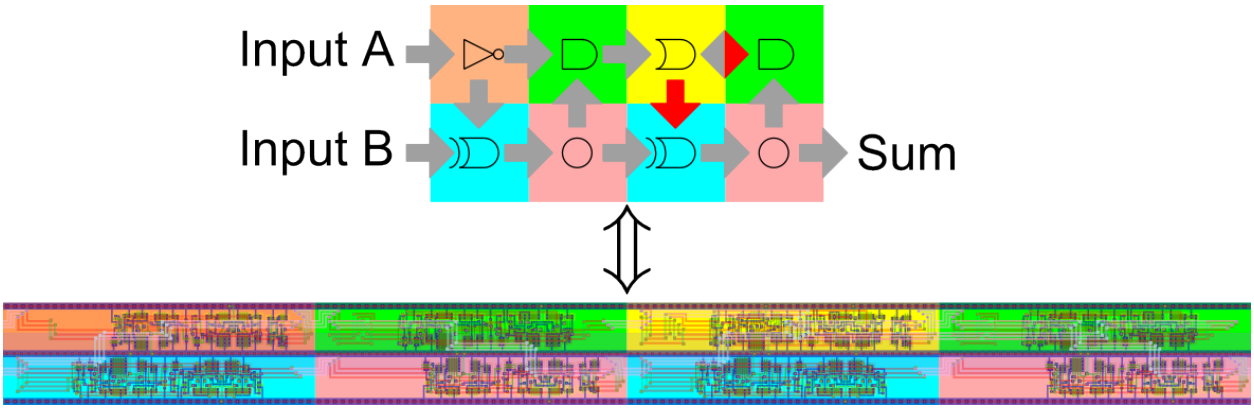
DELETE



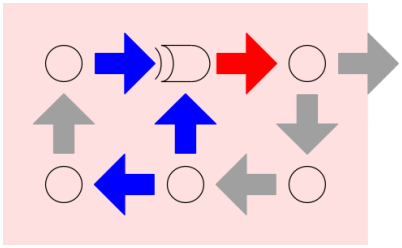
One-to-one Mapping

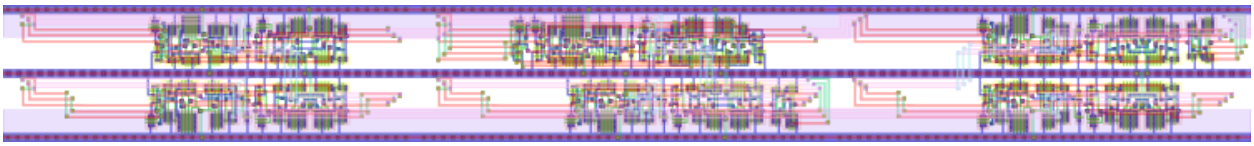
Each of these cells directly corresponds to an ALA cell which allows a direct conversion from ALA designs to hardware. The following two figures show an adder design and a linear feedback shift register (LFSR) design as ALA and as layouts. This equivalence is one of the primary program goals.

Adder



3 Bit Linear Feedback Shift Register





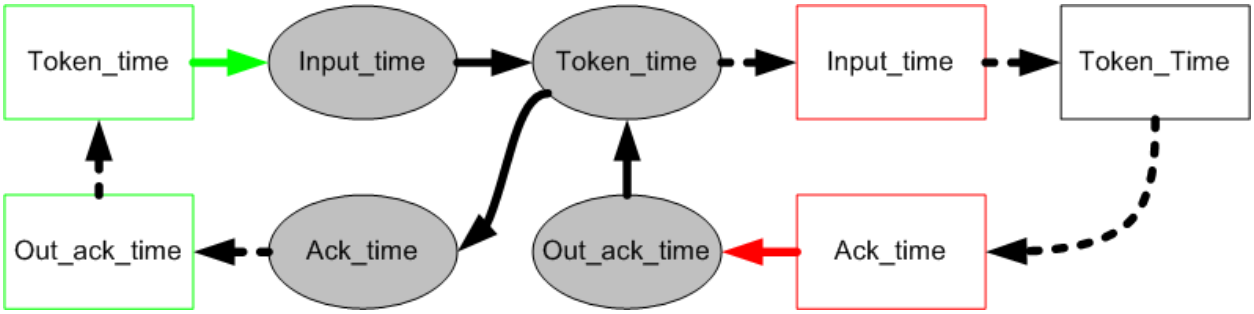
Initial Cell Performance

We used a generic 90nm process development kit from Cadence for performance estimates. The following table summarizes post-layout simulations in Spectre of cell performance in our initial design.

Cell	Throughput	Latency	Energy/Op	Transistors
BUFFER/INVERTER	1.71 GHz	0.0893 ns	0.144 pJ	52
AND/NAND/OR/NOR	1.41 GHz	0.105 ns	0.163 pJ	60
XOR	1.27 GHz	0.115 ns	0.168 pJ	64
CROSSOVER	1.71 GHz	0.0893 ns	0.144 pJ	104
COPY	1.37 GHz	0.0909 ns	0.231 pJ	75
DELETE	1.50 GHz	0.105 ns	0.170 pJ	63

Discrete Event Simulation

Because ALA cells have simple interactions we can significantly simplify the state diagram to hide internal states. Simplifications for some nodes, like "Eval Start" and "Eval End", can be done by adding new dependencies from their input events to their output events. Further simplification can be done by analyzing critical paths and removing redundant connections. We created the following simplified model and added timing and power consumption information from more detailed simulations.



The asynchronous handshaking in ALA means that no global event queue is needed for

behavioral simulation. Consider a situation where the event that clears the output for a cell should happen before the input arrival event. It is trivial to compute the time for the output-generated event regardless of which of the triggering events was processed first. The simplified model also results in needing only to find three delay numbers per cell to characterize its timing. Despite its simplicity, this works well for prediction of timing 'bubbles' where buffers are mismatched. Unoptimized implementations of this model enable interactive simulations for designs that are too large to handle in Spectre.

	Throughput (MHz)			**Energy per Token (pJ)**		
ALA Design	Post-Layout	High Level	Error	Post-Layout	High-Level	Error
Serial Adder	665	776	17%	1.68	1.53	-8.9%
Multiplier	605	734	21%	14.5	14.6	0.7%

While placing inputs and outputs in fixed locations and adding interconnect wiring reduced the number of unique cells needed, we've found these wires to be responsible for at least a 15% degradation in speed and to introduce errors in behavioral performance simulations. To test this hypothesis we ran simulations that did not include the inter-cell wiring parasitics (though still including intra-cell parasitics). The following table shows that out simulator was much more accurate for these predictions. In future cell designs we will include cell interfaces to improve on this.

	Throughput (MHz)			**Energy per Token (pJ)**		
ALA Design	No Cell Wiring	High Level	Error	No Cell Wiring	High-Level	Error
Serial Adder	773	776	-0.39%	1.51	1.53	1.3%
Multiplier	751	734	2.26%	14.2	14.6	2.8%

Comparison of ALA to Full Custom and FPGA Design

We have compared circuit simulations of the initial ALA design with traditional designs in similar technologies. The two most detailed comparisons were between the 3 bit LFSR shown above with a full custom implementation as well as addition and multiplication in ALA with FPGA implementations.

We designed a 3 bit LFSR in the same process that we were using for our simulations. The following table shows a comparison between the custom design and the initial ALA design. This comparison does not include the cost of clock generation or distribution in the energy or transistor count of the custom design. It also doesn't have any safety margin in the clock speed to account for variations in fabrication parameters, temperature, or power supply. By comparison the ALA design requires no external clock and will automatically adjust its speed to compensate.

Design	**Spectre Throughput**	**Spectre Energy**	**Transistor Count**
ALA 3 bit LFSR	1.23 GHz	0.717 pJ	340
Custom 3 bit LFSR	5.68 GHz	0.0196 pJ	78
Overhead	4.61	36.6	4.36

We implemented serial addition and multiplication (4x4 to 8 bit) in Verilog and synthesized it for a Xilinx Virtex 5 FPGA. The following table shows a comparison of the FPGA

performance to the ALA performance. Although the ALA implementation used substantially more energy per bit for multiplication, the throughput is much higher. To better compare these we have computed the energy delay product (EDP) and energy delay squared (EDP2) product which normalizes tradeoffs in speed *vs* power consumption. It is also worth noting that the Virtex 5 is fabricated in a 65nm instead of the 90nm process used by our ALA designs.

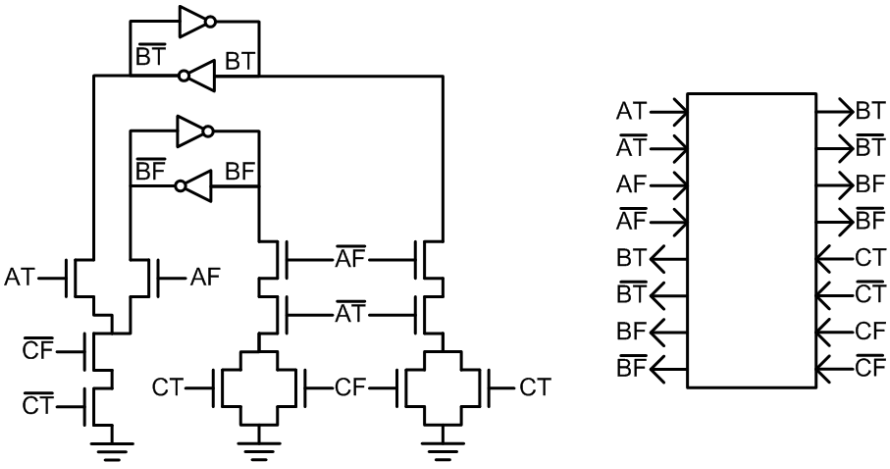
Comparison of initial ALA and FPGA implementations

Operation	Throughput	Energy/Bit	Energy*Delay	Energy*Delay^2
ALA Serial Adder	665 Mbps	1.68 pJ	2.52 pJ/GHz	3.80 pJ/GHz^2
FPGA Serial Adder	115 Mbps	1.65 pJ	14.3 pJ/GHz	124 pJ/GHz^2
ALA Serial Multiplier	605 Mbps	14.5 pJ	24.0 pJ/GHz	39.6 pJ/GHz^2
FPGA Serial Multiplier	116 Mbps	2.85 pJ	24.6 pJ/GHz	211 pJ/GHz^2

Depending on the metric we see that performance of the initial cell design is already on par or well above FPGA performance. The motivation for using EDP2 is that scaling voltage will improve energy consumption quadratically while only reducing speed linearly. EDP compensates better for the parasitics that start to matter at small feature sizes. We expect that scaling to the same feature size, and using our improved designs (next section), will make the ALA implementations simultaneously faster and lower power.

New ALA Cell Design

Rather than proceeding with fabrication of the initial design, we have experimented with several variations to reduce cell overhead. A half-buffer cell design that integrates several of these techniques is shown here (not shown are the two reset transistors). It uses dynamic logic as in the initial design, but signals are passed with their complement which saves several gates. Additionally both sides of the keeper inverters are driven similar to a SRAM cell.



New Cell Simulation

	Throughput	Energy/Bit	Transistors
New Cell	2.22 GHz	0.0657 pJ/Bit	22

20% Layout Penalty	1.85 GHz	0.0788 pJ/Bit	22
Old Cell	1.71 GHz	0.144 pJ/Bit	52

A typical synchronous register requires about 12 transistors so the hardware overhead of the new design is likely to be no more than 2-3 times the area of a synchronous implementation.

Two-Phase Handshaking

Explicitly representing empty wire states requires many extra wire transitions which each dissipate power and take time. By using a two-phase handshake it is possible to eliminate these extra transitions. For naive implementations, the increased complexity of the logic can result in a net performance loss, however the limited number of ALA cells lends itself to optimization. We are investigating a two-phase handshake which could potentially as much as halve power and double throughput.

In the current hardware designs we use a two-wire interface between cells to encode true, false, and empty. A two-phase handshake uses one wire to encode the data, and one wire to encode a parity bit. Cells fire when their parity matches all of their output cells and doesn't match any of their input cells. For initialization we start with all cells having the same parity, then we invert the parity signal between cells where a token should initially be placed.

To better understand the two-phase handshake consider the following example. T or F represent true and false tokens with a low parity bit and T' and F' represent tokens with a high parity bit. We start with a chain of 5 buffer cells passing data to the right. The leftmost cell is initialized to a F' token and all other cells initialized to T as shown here:

F', T, T, T, T

The second token matches the parity of its output and doesn't match the parity of its input so it can fire by transitioning to a F' token.

F', F', T, T, T

It would now be valid to insert an additional token at the left of the chain or for the third cell to fire. The second cell is forced to hold its state until its data has been passed on. Let's assume that the third cell fires:

F', F', F', T, T

After two more steps the F' token will have propagated to the last cell in the chain:

F', F', F', F', F'

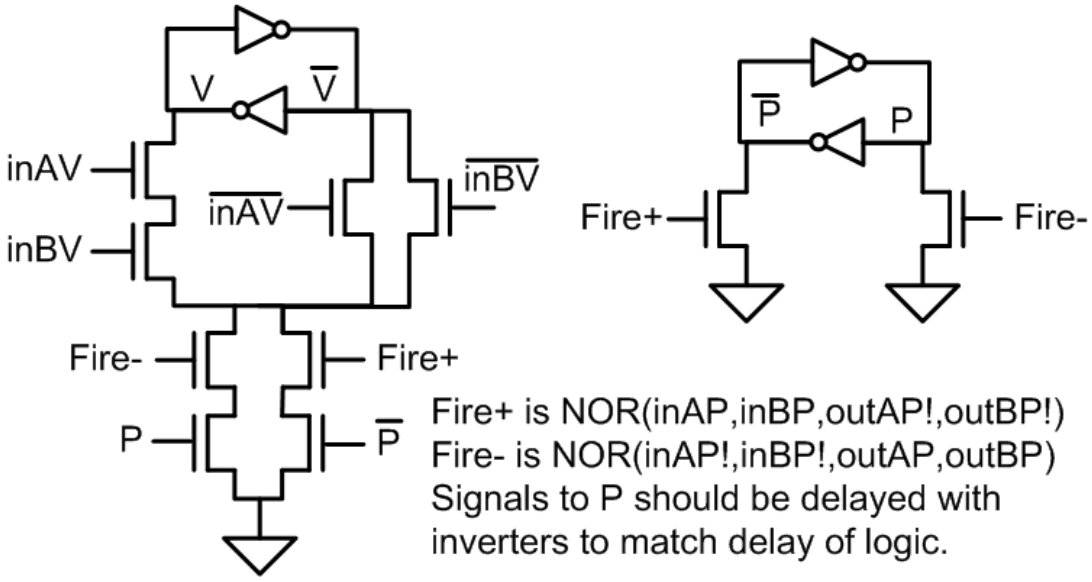
We can still store one token per cell as we could with empty tokens by alternating parities as shown here:

F', T, T', F, F'

Another metaphor for understanding a two phase handshake is that we are tracking the boundaries between tokens. We need the parity bit so that it is possible to encode consecutive tokens with the same value.

Choice of Representation and its Impacts on Hardware

Ideally there would be only one wire transition between each pair of tokens. This would minimize power lost from token transitions and simplify their timing. This is possible by encoding T as 11, F as 00, F' as 01 and T' as 10. However this requires looking at both wires from all inputs and outputs to determine when a cell can fire. In the case of a two-input two-output cell this requires a single function of 8 inputs. Encoding parity and value bits separately improves performance; the following schematic shows a possible implementation of a two-input two-output NAND cell:



ASIC-in-an-Afternoon

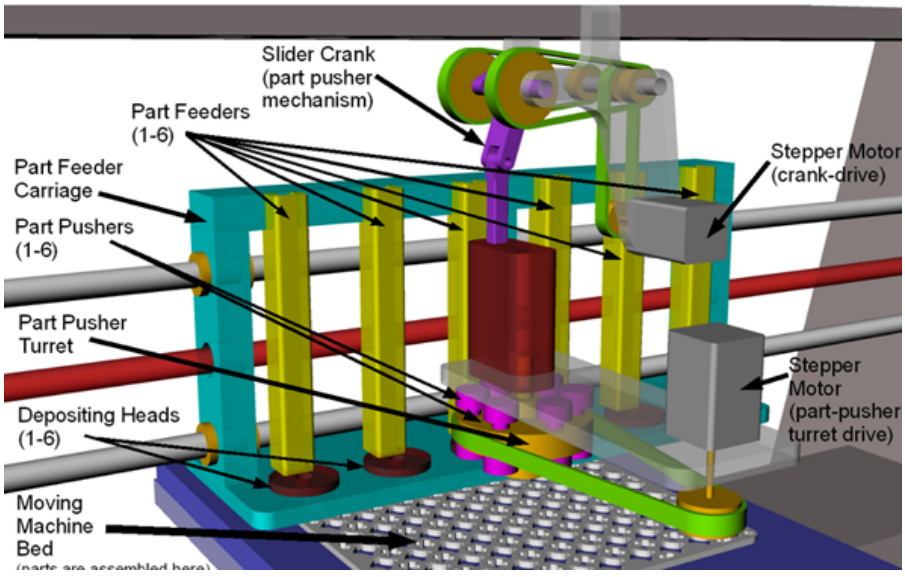
The design and manufacture of an application specific integrated circuit (ASIC) can be a long and expensive process for even simple designs. Because ALA is composed of a small set of discrete blocks it offers an opportunity to use digital assembly to prototype high performance electronics at a drastically reduced cost. Given a supply of tiles for each kind of gate, a million-cell circuit could conservatively be assembled in a few hours.

- *Pins*: For a cell to have an input and an output as well as power, ground, and reset lines on a single face requires 11 pins with quad-rail data encodings. To support full buffers an additional 2 pins might be needed for acknowledge signals. The total pins per side would be less than 13 in the worst case, but easily as few as 8.
- *Cell Size*: Using a bonding pitch of 30 μm, or a larger 100 μm interconnect pitch combined with multiple rows of connections, would allow 13 pins to fit on the edge of a 500μm cell.
- *ASIC Size*: If assembled in 3D, a million ALA cells fill a 100 voxel cube. Assuming a conservative 500μm cell size the total ASIC size would be a 5 cm, and be proportional to the required number of cells.

- *Power Consumption*: The initial design consumes .1 pJ per cell update and operates at 1 GHz. If all cells are always active the total power consumption would be 100 Watts. The cell update rate and dissipation will decrease as the supply voltage is decreased; this could be done dynamically to regulate the load. This also assumes that every cell is filled and updating at full speed which is a worst-case assumption, implying a 15 Peta-cell-ops update rate. Assuming a factor of 10 decrease in dissipation (through a combination of better design, dynamic voltage scaling, better aspect ratio, and smaller activity factor) the total cooling needed would be about 670 W / m^2 which is comparable to bright sunlight.
- *Assembly time*: Current pick-and-place machines are capable of placing over 100,000 parts per hour which is roughly 30 Hz. Assuming a million cells it would take 10 hours to assemble the ASIC with existing tools. However the assembly process can take advantage of the discrete locations and regular structure of the parts, potentially operating orders of magnitude faster.
- *Materials cost*: A typical packaged surface mount transitor costs on the order of $0.01 in quantities of 100,000. At this tile cost per cell a million cell ASIC would cost $10,000. Even at this price the process would be competitive with existing ASIC prototyping options. Surface mount resistors are close to $0.001 in similar quantities, which would reduce the ASIC cost to $1000.
- *Number of parts*: Although there are less than a dozen cell functions that are required, the possible combinations of inputs and outputs would need to be included. Including all possibilities in 3D would correspond to 150 component types, which may be reduced by taking advantage of symmetries.

In addition to computation it would be possible to integrate display, sensors, actuators, and power storage into the structure of such an additively-assembled ASIC. If assembled with a reversible process one could imagine creating custom systems with powerful integrated computation that could quickly be recycled for completely different tasks.

Future work will be investigating and prototyping the feasibility of these additive assembly processes, and looking at the tradeoffs between physical and electrical models of reconfigurability.

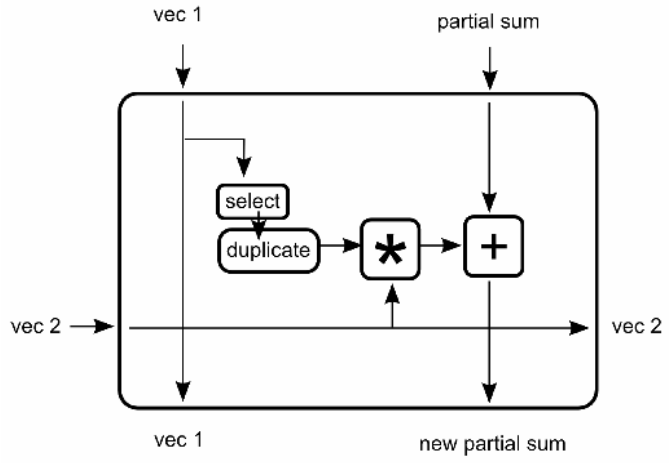


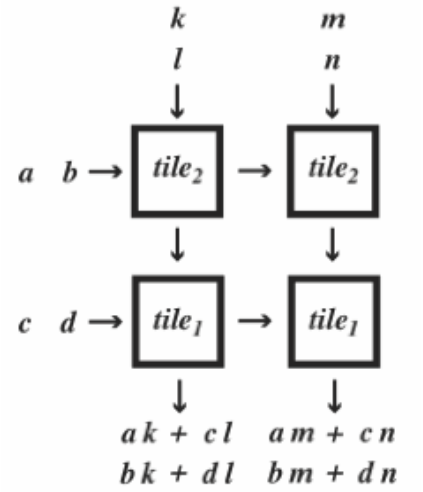
BLAS Cost Comparison

Having developed a workflow that allows us to map designs directly into hardware we estimated performance relative to existing implementations. We chose matrix vector multiplication and estimated the total power consumption and throughput for an ALA design and compared it with data from a similar benchmark executed on CPUs, GPUs, and FPGAs.

The benchmark executed was multiplication of a 256 by 256 entry square matrix by a 256 entry vector on 64 bit operands. Energy consumed per op and throughput were both measured. Data for CPU, GPU and FPGA implementations is based on execution with double-precision floating point, from http://research.microsoft.com/pubs/130834/ISVLSI_FINAL.pdf. Because the floating point implementation in ALA was incomplete we compared these values to a 64 bit fixed-point implementation. Preliminary designs for floating-point operations indicate that this is a reasonable approximation, however a 128 bit fixed-point implementation was also compared as an upper bound.

The implementation of the benchmark is based on the matrix multiplier described in the "Tools" section of this document.





The figure above shows a small matrix-matrix multiplication. To assemble a matrix-vector multiplication we use a single column of identical tiles.

Evaluation Technique

The bitsliced simulator (below) was used to calculate the total number of cell firings required to produce each bit of the output. This value was multiplied by the maximum possible consumed power per cell to estimate the total energy per bit. Energy per bit was then multiplied by the word length and the number of tiles used to get total energy per operation.

Because the design was constructed to have high throughput it would produce output bits as fast as the slowest gate (XOR which runs at 1.23 GHz/bit). Because the output occurs on a single serial channel this speed must be scaled by the number of bits per word and by the number of words per vector.

To estimate manufacturing cost we compared based on transistor count. We assumed cells must be placed on a grid with room for the largest cell (buffered crossover which requires 104 transistors in the initial design) at each site.

Single tile for 64 bit word:
0.23 picojoules/cell firing×2146.8 cell firings/bit×64 bits/output
= 3.2×10^{-8} joules/word
1.23 GHz/bit×1 word/64 bits
= 19.2×10^{6} word/second in steady state
2586 gates×104 transistors/max gate
= 268944 transistors/tile

Array of 256 64-bit tiles:
2146.8 cell firings/bit×64 bits/output×256 words/vector×256 tile steps/iteration
= 9.0×10^{9} cell updates/iteration
3.2×10^{-8} joules/word×256 words/vector×256 tile steps/iteration
= 2.1×10^{-3} joules/iteration
1 output channel at 19.2×10^{6} word/second in steady state×1 vector/(256 words)
= 7.5×10^{4} iterations/second
256 tiles/array×268944 transistors/tile
= 6.9×10^{7} transistors/array

Single tile for 128 bit word:
0.23 picojoules/cell×4140.14 cell firings/bit×128 bits
= 1.22×10^{-7} joules/word
1.23 GHz/bit×1 word/128 bits
= 9.61×10^{6} words/second in steady state
4965 gates×104 transistors/gate
= 516360 transistors/tile

Array of 256 128-bit tiles:
4140.14 cell firings/bit×128 bits×256 words/vector×256 tile steps/iteration
= 3.5×10^{10}
1.22×10^{-7} joules/word×256 words/vector×256 tile steps/iteration
= 8.0×10^{-3} joules/iteration
256 tiles/array×516360 transistors/tile
= 1.3×10^{8} transistors/array
1 output channel at 9.61×10^{6} words/second in steady state×1 vector/(256 words)
= 3.8×10^{4} iterations/second

Scaling

The cell designs here assume a 90nm process which is significantly larger than the compared hardware. Above the scale where constant field approximations break down, gate delay is approximatively linear in feature size and energy per operation is approximately cubic. Simply scaling the existing design from a 90nm to a 65nm feature size (as used in the compared FPGA and CPU) would likely improve speed by around 40% and reduce energy per operation by around 60%.

Comparison to Existing Implementations

We looked at timing, power consumption and amount of hardware necessary. The following tables summarize ALA timing and power consumption for our initial design relative to data gathered in http://research.microsoft.com/pubs/130834/ISVLSI_FINAL.pdf as well as estimated transistor counts. The transistor numbers ignore external components such as memory.

Implementation	**Speed**	**Power Consumption**	**Estimated Area**
ALA	13 µS/output vector	480 iterations/joule	0.069×10^9 transistors
ALA(65nm est)	9.4 µS/output vector	1300 iterations/joule	0.069×10^9 transistors
Multicore	12 µS/output vector	790 iterations/joule	0.29×10^9 transistors
FPGA	50 µS/output vector	1800 iterations/joule	1.1×10^9 transistors
GPU	100 µS/output vector	80 iterations/joule	1.4×10^9 transistors

We have developed an end-to-end workflow that allows us to map high level designs directly into integrated circuits. Although our current tools and designs are preliminary, they are already competitive with many existing alternatives. In addition they offer many other benefits including portability, scalability, and verifiability.

Reconfiguration

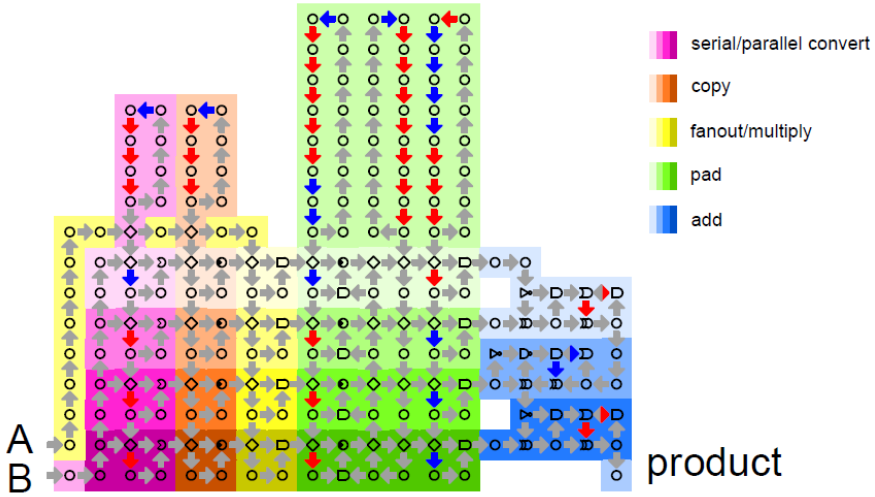
The current implementations for ALA allow us to map designs to fixed-function hardware. Although the scope of the current work has not included reconfigurable implementations, there is a spectrum of possible options for reconfigurable designs for future work. A direct option where each cell can be directly configured in-band, as described in our *Reconfigurable Asynchronous Logic Automata* reference. While this approach offers the best benefits for scalability and portability, the relatively complex logic in each cell would introduce a significant overhead. Another alternative would be to virtualize the logic for reconfiguration in an array of processors that update state from RAM. By amortizing the cost of the processor over many cells the total number of transistors needed is limited only to the amount of stored state (about 20 bits for full configuration and tokens). The speed of this approach is likely to be limited by the memory bandwidth needed to read and update cells. A naive approach of reading the full state of each cell for each update would require the bandwidth of a modern DDR3 interface (around 10 GB/second) for every 4 cells.

ALA cells could potentially allow very small volume production of custom designs. All cell types can be made using only slightly different combination of a handful of blocks. For example changing between AND, NAND, and OR only requires swapping pairs of input wires. Masks represent a substantial fraction of chip fabrication costs. If functionality can be

selected using a single custom mask to set interconnect and cell type, the other masks could be reused allowing significantly cheaper fabrication. If the custom mask was the final layer it might be possible to stockpile partially completed chips (reducing turn time) and use a larger (and thus cheaper) feature size or directly set configuration on top of the die with a tool such as a Focused Ion Beam or laser.

Algorithms

The algorithms effort has developed ALA implementations of arithmetic with parametric designs for both fixed- and floating-point representations, assembled these in a library of the core BLAS (Basic Linear Algebra Subprograms) used in scientific computing, and developed techniques for characterizing and optimizing their performance.

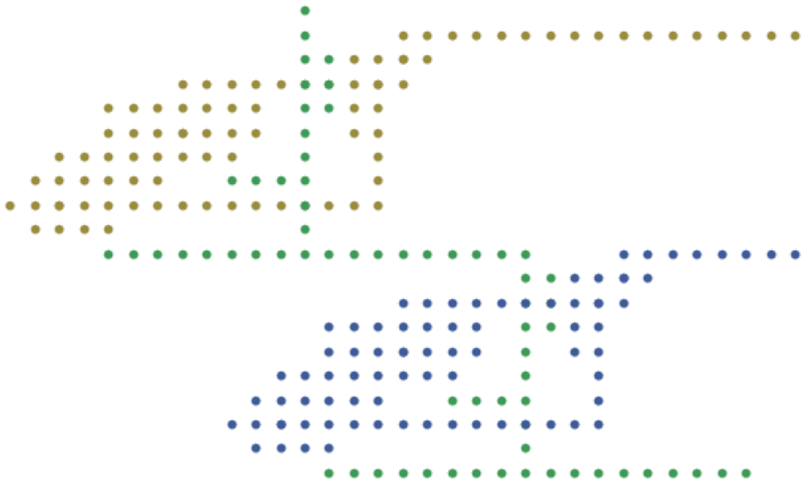


Performance Analysis

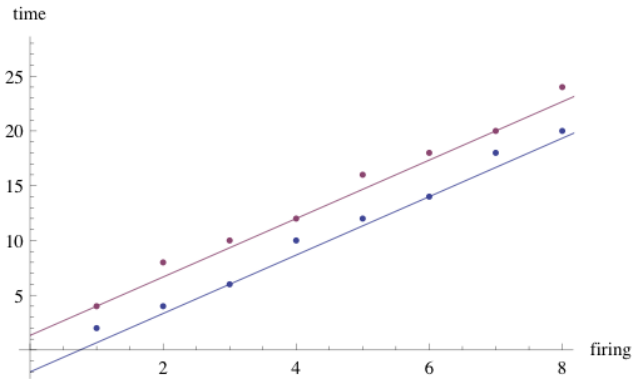
ALA algorithms were initially optimized manually. To accommodate larger designs a *MathALA* environment was developed (as a front end to *Mathematica*). This provides direct computation of throughput, and layout of circuits using a directed acyclic graph flow model.

We have produced a complete analysis of the performance of ALA circuits (see references), including copy and delete gates. When copy and delete gates are programmed with periodic behavior, we can directly characterize and optimize performance. If copy and delete are programmed to behave in a data-dependent way, throughput and latency must be computed by specifying inputs.

One enabling step was the characterization of timing domains. Copy and delete gates modify the rate of token flow between their inputs and outputs. In order to characterize circuits it is necessary to (i) partition the circuit into contiguous regions that operate at the same rate, (ii) apply an algorithm that uses this information to compute throughput.



We have formalized the definition of latency in ALA circuits, and defined procedures to compute it directly (documented in the references).



Abstraction of Modules

Our throughput algorithms were initially adapted to operate on cell-level circuit descriptions and compute global properties. Since the complexity of this algorithm grows as $O(n^2)$, it is not feasible 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. This works for something like a matrix multiply 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 outputs of the module) needs to be assembled in order to compute our latency and throughput metrics for these aggregate circuits.

We can use this fact to derive a procedure that can then be performed hierarchically, starting from the bottom up. Small modules are successively assembled into larger modules, which are then combined into yet larger modules. This way the complexity of an iteration can be bounded by the number of modules assembled in it, and the total number of such operations necessary to assemble a very complex circuit is proportional to the

number of unique modules assembled within it times the log of the total number of such primitive modules. We can expect the number of unique modules used to specify a very complex circuit to be a small integer - given that even constraining ourselves to only addition, subtraction, multiplication, and division covers a great many powerful and useful circuits.

DAG-flow

One advantage of abstracting modules is that performance guarantees can be maintained without complexity diverging. 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. A particular procedural abstraction we call *DAG-flow* provides the programmer with 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.

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.

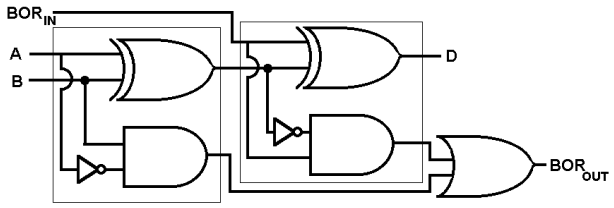
Once we have a specification of such a circuit, we begin layout with a pass of assembling columns in which we equalize width. Then, beginning by fixing the left-most column at the x=0, we add columns one-by-one, connecting them with smart glue, which takes as input potential values associated with the 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 rotate them and assemble columns with these as modules.

DAG-flow Circuit Examples

Although *DAG-flow* was conceived with "large" modules in mind, it can also be used to lay out individual gates in arbitrary acyclic configurations. In order to satisfy the constraint of left inputs and right outputs, we add a wire gate on top of a binary gate to make a two-gate module.

Here is an example of the design procedure for a very small module - one that performs

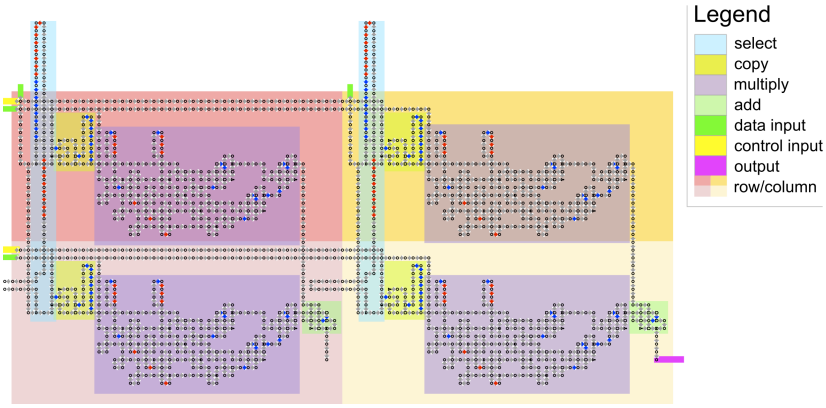
subtraction. We begin with a standard binary logic diagram:



(source http://www.sheffield.ac.uk/physics/teaching/phy107/logicsub.html). We map this into DAG-flow format.

This design logically connects the borrow out to the borrow in so that the circuit functions a streaming sequential subtractor.

Using the same procedure with larger blocks we can construct a matrix multiplier.



These abstractions will be useful in achieving our next programming goal of implementing LU decomposition for solving systems of equations.

Floating point in ALA

A floating point adder was implemented in ALA, shown below:

The stages of the floating point operation were abstract into functional modules, which are chained together in sequence in a bit-wise way, as is visible in the graphic. The sequential stages of floating point addition as implemented here are:

1. ***parse*** -- in which the exponents are separated from mantissas
2. ***compare exponents*** -- exponents are compared, which determines which of the mantissas will be shifted prior to addition, and by how much.
3. ***swap*** -- arguments are swapped so that the smaller input can be flowed into the following step
4. ***shift mantissa*** -- the smaller argument is down-shifted and truncated.
5. ***signed add*** -- the aligned mantissas are added or subtracted according to their signs.
6. ***normalization*** -- the final sum/difference is normalized so that the highest order bit in the output is significant.

For this we used *MathALA*, as the *Snap* language (below) had not yet matured at that point.

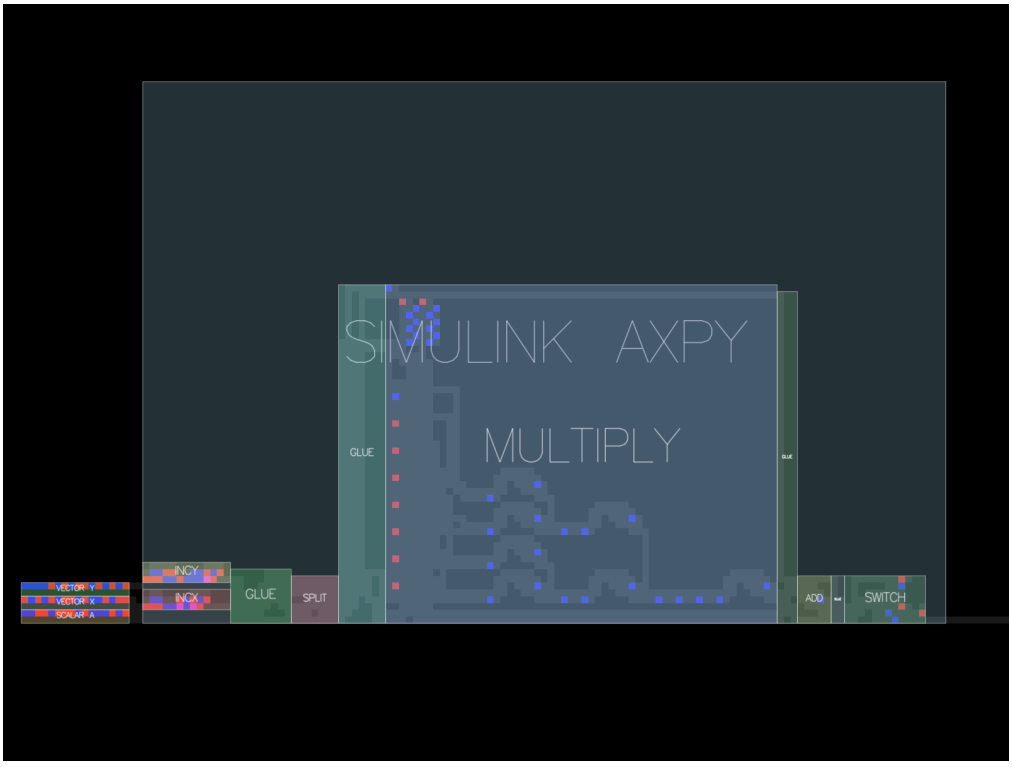
The BLAS

The Basic Linear Algebra Subprograms are a set of basic operations out of which higher-level linear algebra routines can be built. Programming early computing hardware required platform-specific, one-off code to be optimized by end-users. The BLAS were proposed in the 1970's as a set of operations to be optimized for systems in order to make writing efficient, portable code possible.

We have implemented a core set of BLAS in ALA to show that HPC can be built on top of it, in the same abstract way that it is done in current serial hardware. By doing it in ALA the advantages of bit-level parallelism and parametric scalability can be obtained, in contrast to current practices for serial hardware. The BLAS that have been implemented are dot, axpy, swap, copy, and scal. Note that in ALA the final two are subsets of sswap and saxpy, respectively. These were implemented with integer modules, which could be replaced with floating-point.

We did some example implementations in *Simulink* (below), and others in the *MathALA* language. Here is the swap operation as programmed in *MathALA*:

And here is the axpy operation programmed in *Simulink*:



Videos are available of the operation of these routines.

Scaling of Mathematical Operations

Operation	*latency*	*throughput*	*power*
min-latency parallel add	O(sqrt(n))	sqrt(n)*1/2	O(sqrt(n))
multiplier	O(n)	1/2	O(n log n)
divider	O(nm)	1/2	O(nm)
matrix multiply	O(n k)	1/2 n	$O(k^2 n \log n)$

Comparison to Jaguar

Comparing scaling to current supercomputers, Jaguar achieves 1.8 Petaflops at 7 MW (source:top500.org). Assuming that we are able to replace logical wires with physical wires at 1/10 the cost of logical wires, a single-precision FLOP costs about 150 cell firings per bit times 32 bits, or about 5,000 cell firings. This means that 1.8 Petaflops cost $5*10^3$ * $1.8*10^{15}*.15*10^{-12}$ J = 1.35 MW. This is using the initial cell design. Now, considering that the new cell design uses a factor of 2.5 less transistors per logical cell, and observing that roughly half of the estimated power in the current design derives from logical cells, we get .7 + .7/2.5 = roughly 1 MW to perform the same task. This neglects cooling costs, however heat transfer will be simplified by the homogeneity of the dissipation. Future work will tighten this estimate with both simulation and measurements.

Tools

For ALA programming, we've developed both visual and textual programming languages, and efficient simulators for multi-core clusters and microcontrollers.

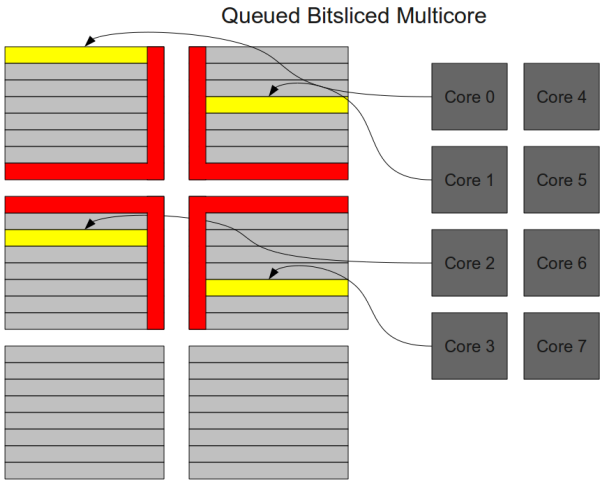
Bitsliced Simulator

As the size of the circuits to simulate grew from dozens, to thousands, to millions of cells, we needed faster simulators. Initially in the development of the high-throughput matrix multiplier, we used a simulator in the *PyALA* layout tool. The exact memory footprint of a cell in this simulator is hard to estimate, as it was interpreted in Python, but it was on the order of 10^3 bytes per cell. Order 10^4 cell updates per second could be achieved. Very quickly, this simulator grew inadequate in space and time. Next we started exporting our designs to be simulated in *rala_util*, which is implemented in C. A cell in *rala_util* takes order 10^2 bytes to represent, and order 10^6 cell updates per second could be achieved. Unfortunately, the memory footprint proved to be the limiting factor. In a 50x50 matrix multiplier, the 3 million cells used up all the memory on a cluster node, meaning any larger circuits required paging. A new style of simulator clearly had to be written.

Drawing inspiration from a previous embedded ALA simulator that we wrote, a new C library was written; *librala*. Previous simulators we had written evaluate ALA circuits in the straightforward way: iterate through all cells pending updates, check their configuration in some data structure, and then update them. The *rala_util* library works this way. Unfortunately, branching is one of the most expensive operations on a modern processor, and this method of evaluation will branch when it checks the type of the cell to be evaluated. This leads to a question: can all ALA cell types be evaluated by a single piece of branchless code? A natural representation is to treat ALA evaluation as a big Boolean expression, with the configuration of a cell, and its inputs as variables. We found such a Boolean expression that represents our current set of cells efficiently, using 20 bits to represent the configuration of a cell, and 8 bits to represent the state of that cell's buffers.

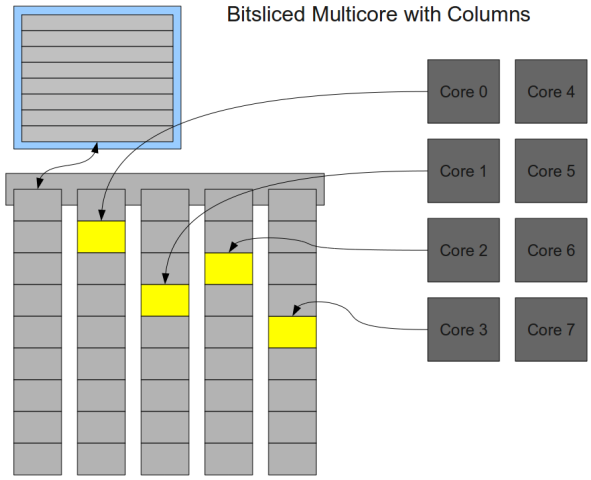
This uses a total of 28 bits = 3.5 bytes per cell. Next, this Boolean expression can be evaluated in parallel on many cells simultaneously, by packing configuration variables into machine words which are operated on in parallel using bitwise logic instructions. This gives a representation in which 64 cells share a set of 28 separate 64-bit words to collectively encode their configuration and buffer states. This is an ALA bitslice. Next, we use 64 of these ALA bitslices to encode a 64x64 block of ALA. Updating an ALA block consists of evaluating this Boolean expression for each slice in the block, and then pushing some results into neighboring slices. This gives extremely good locality of reference. An implementation was made in which eight cores simultaneously updated a grid of 64x64 blocks.

Words of configuration being processed in cartoon 64x64 blocks in parallel:



Each gray row represents the 28 words of 64-bits that make up a row of cells. Each core is operating on a single row at a time (yellow) and pushing results back into the adjacent rows. If each core operates on a single block, then read/write order is unpredictable in some bits of some words (red). Even with appropriate locking to guarantee correctness, this is a major performance detriment.

Columns being processed in parallel, where each column is a set of 64x64 blocks:



Because of the problems of simultaneously updating adjacent blocks, a new model was made in which columns of ALA blocks are updated in parallel. If one simply separates out columns being updated such that they do not share any neighbors being updated simultaneously, cache issues are mitigated, and one can effectively parallelize the ALA computation (see graph below for parallelization).

Benefits of this representation over previous representations:

1. Cells can be represented with extreme compactness. As a cell's location is implicit, a cell can be represented completely in 20 bits of logic configuration + 8 bits of token state = 3.5 bytes. For large repetitive circuits, the configuration bits can be shared among identical modules, giving an amortized cost of 1 byte per cell. (Order 10^0, for comparison with previous numbers.)
2. As the update rules of ALA are represented as the evaluation of a Boolean expression, by packing inputs into machine words, n cells may be updated in parallel, where n is the width of the machine word in question. The cluster we are simulating on has 64-bit words, and thus 64 cells may be updated in parallel.
3. An unrolled version of the update function is completely branch-free, re-entrant, and parallelizable. Because of the entirely local nature of ALA computation, one may update arbitrary portions of the circuit simultaneously, and not compromise the end computation. Therefore, we used OpenMP to parallelize the simulator to all eight cores of each cluster node.
4. This simulation technique gives a "superluminal" model of ALA simulation, whereby tokens may propagate arbitrarily far in a particular direction in which the simulator is biased in a single update cycle (the direction "down" a column). By manipulation of the update patterns of the simulator, one may extend this bias in all four directions, giving a type of ALA simulation which gives equivalent results to the burst update model, but which takes fewer update cycles for many circuits.

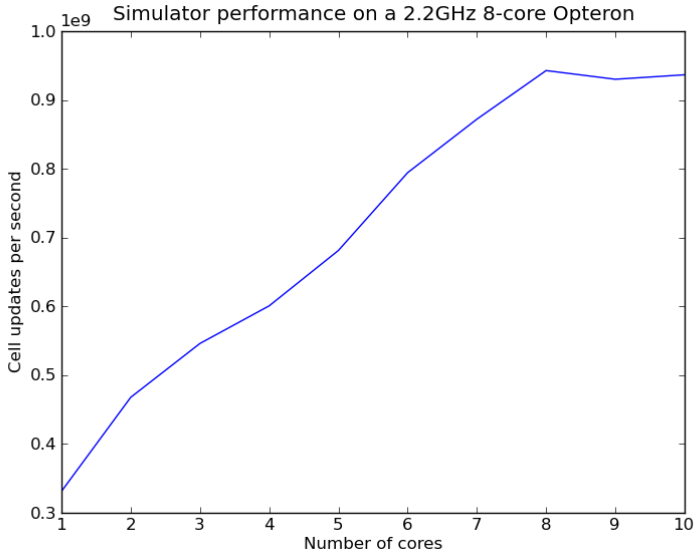
For compatibility with current mathematical models of throughput, *librala* was also made to support a burst update mode, whereby this superluminal token transport is prevented.

The downsides of this representation:

1. Exact figures on count of cells fired are slow to get without population count instructions, which are not universal, and require making code less cross-platform.
2. One pays to update unused space around modules, as the representation encodes position of cells implicitly, and therefore requires some waste. This has been mitigated by allowing sparse "holes" in the grid of 64x64 blocks, which we have implemented.

Three interfaces were written to *librala*: C/C++ interface, a Python interface, and a Mathematica interface. In addition, *librala* supports the *rala_util* circuit exchange format. This means all of our previous circuits can be directly imported into this simulator.

librala simulator performance:



This plots cell updates per second (linear scale, up to 10^9) against number of cores on a large sample circuit. As expected, the performance levels out at 8 cores, which is the number in a cluster node. The maximum at cores=8 is approximately 900 million cell updates per second. (Approximately 10^9 updates per second, for comparison with previous numbers.)

With these very fast update rates, animating the progress of extremely large circuits in real time became possible, so an extension was added to *librala* which renders circuits such that each cell is mapped to a single pixel, allowing for large circuits to be efficiently displayed.

A screenshot from the new visualization tool:

Pictured is the lower left corner of the 50x50 high-throughput matrix multiplier. This image comprises about 10% of the overall circuit. Each rendering now takes approximately 50ms, compared to the 16 minutes it took to render the circuit with our previous tools. Updating the entire multiplier takes approximately 200ms, allowing for animation at approximately 4 frames per second.

Multicore to Multiprocess

As the size of circuits we have wanted to simulate has scaled up, the limiting factor has become memory constraints. Simulation performance stays mostly constant with circuit size, with the exception of two major drops; when circuit grows larger than the cache, and when the circuit grows larger than available RAM. The cost of paging is so prohibitively high that we have not explored much simulation of circuits larger than a few gigabytes of RAM footprint. Unfortunately, the OpenMP parallel simulator doesn't directly help us with this constraint, as OpenMP merely enables us to parallelize to multiple cores that all share a common fixed memory pool. While extensions to OpenMP which alleviate this problem exist (namely Cluster OpenMP) an initial review indicated they were likely to perform worse than a custom solution.

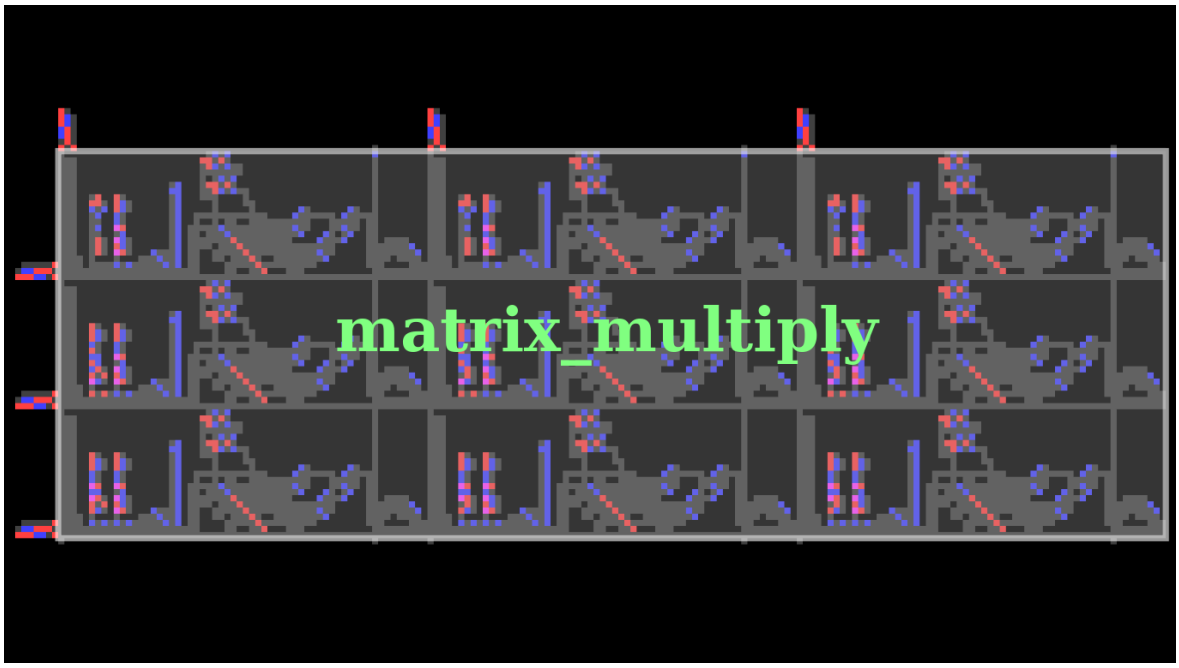
To parallelize ALA to multiple machines, the *rala_server* and *rala_client* programs were written. The server program is configured with an ALA circuit diced into chunks, as well as a list of their interconnections, or pipes. As clients connect to the server, they are given a chunk of the larger circuit, as well as a list of the relevant pipes in and out of the circuit chunk. Whenever a client detects a token on a gate in its list of pipes, it sends that token to the appropriate peer who then acknowledges the token. Network latency and throughput are almost irrelevant, due to a convenient scaling property: if a machine is simulating an n by n square of ALA, the time taken to simulate a single update goes as the area of the square, $O(n^2)$, whereas the time taken by network communication goes as the perimeter of the square, $O(n)$. In the limit of increasingly fast machines handling increasingly large chunks each, the communication overhead of this simulation strategy goes to zero. Currently, as updating a 20000x20000 patch (approximately 1 GB of memory footprint) takes approximately 300ms, and can produce at most 40000 tokens on the boundary, one must be able to handle at most approximately 120000 tokens per second. In practice, we have not designed circuits requiring such dense interconnect at edges, making general purpose multi-machine simulation easily achievable without performance loss over even dial-up connections.

Hierarchical Visualization

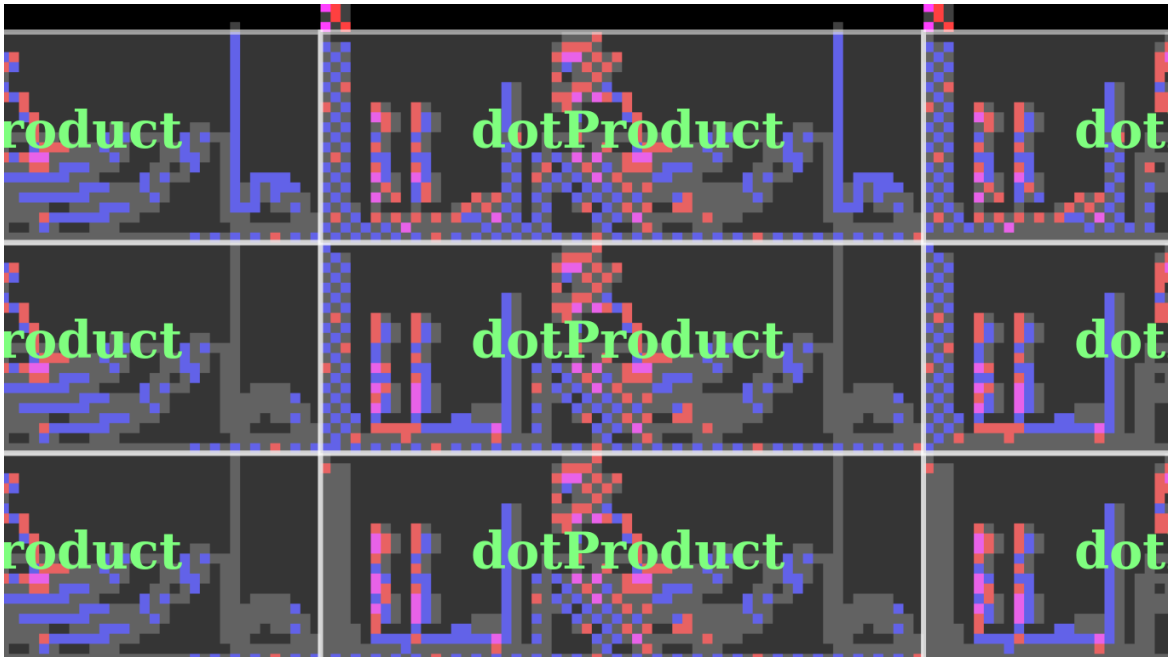
Practical large-scale designs typically have repetition and hierarchical structure. No large program has ever been written without the use of routines to encapsulate functionality. Given this, viewing a large ALA circuit via rendering methods similar to those above gives rapidly decreasing emphasis to crucial details. For example, two enormous ALA blocks might be connected by a single skinny wire, a detail which would disappear were one to view the circuit at a scale where both blocks are visible. This sort of disappearance of connections can be seen in the matrix multiplier image above; it takes close inspection to see the web of skinny interconnections that surround the dot product units.

To address this, we implemented a method of automatically rendering which reveals the hierarchical structure of the circuit being shown explicitly, by boxing and labeling functional blocks, and further by highlighting small interconnects.

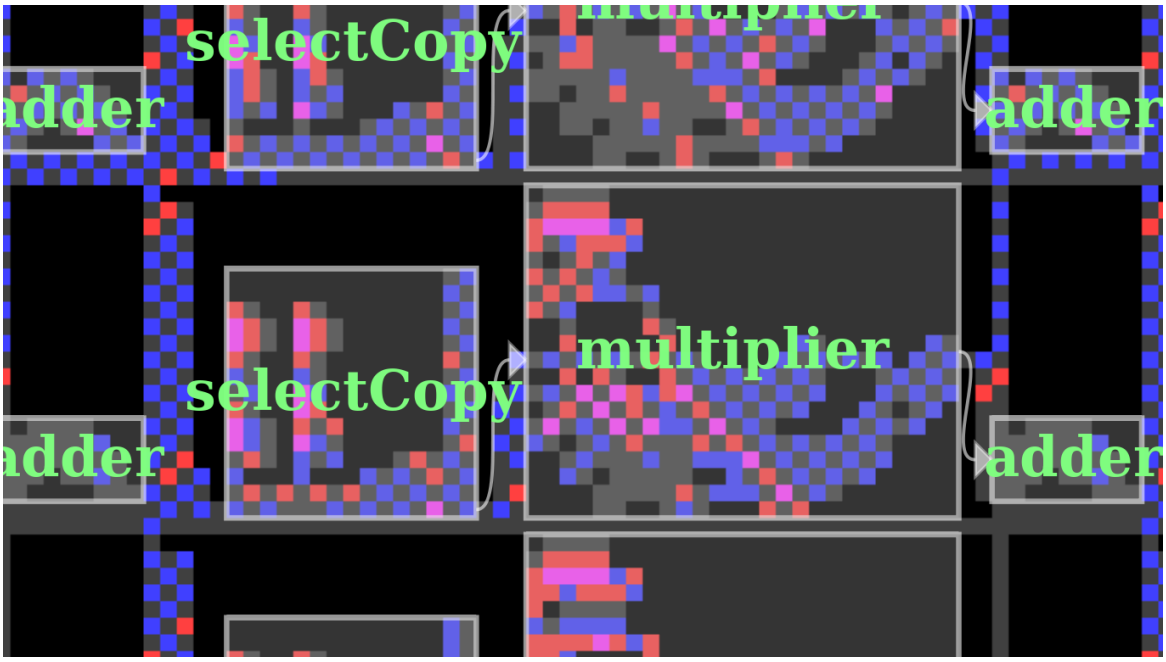
The top level of a 3x3 matrix multiplier:



An intermediate level, where the matrix multiplier has decomposed into an array of dot product units:



Yet another level deeper, where each dot product has decomposed into chained select/copy, multiply, then accumulate:



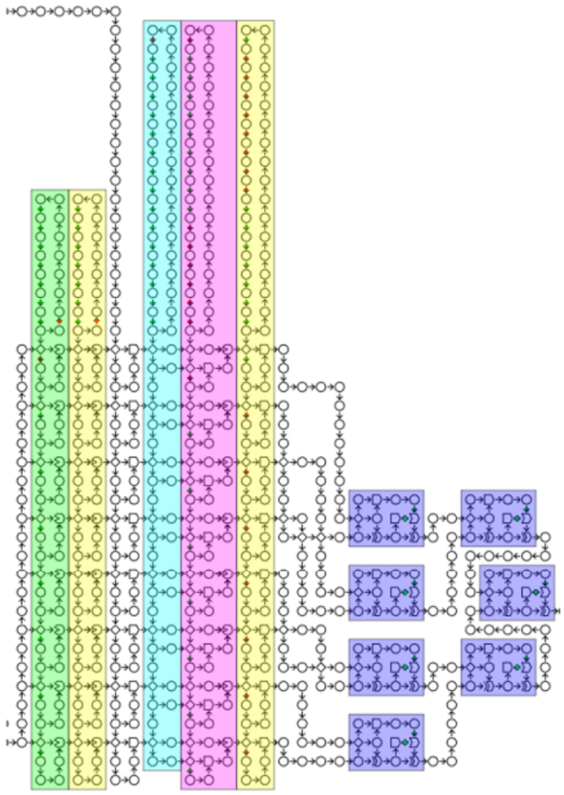
Videos are available of this animation.

Snap: A robust language for ALA circuit design

There were two goals in creating Snap -- the first was to make a dedicated language and syntax for defining ALA circuits, and the second was to make the implementation executable without proprietary software (as with Mathematica in *MathALA*). The result, Scala Snap, has allowed us to implement spatial layout of circuits with intuitive syntax and primitives that straddle the divide between logical structure and geometry. An implementation of integer multiplication demonstrates these features of Snap.

```
def scalar_multiply (n: Int) =
  vc(WIRE(">", "v"),
     hc(fan_out(n), select(n),
        duplicate(n), multiply(n),
        pad(n), reduce(n/2, d => add)))

def select(n: Int) = {
  val bits = Range(0, n).map(_ != 0);
  hc(vc(bit_loop(bits), phased_fanout(bits)), GlueZip(),
     vrep(n, binary(delete)))
}

def duplicate(n: Int): Module =
  hc(vc(bit_loop(bits), phased_fanout(bits)), GlueZip(),
     vrep(n, binary(copy)))

def multiply(n: Int) =
  hc(fan_out(n), vrep(n, binary(and)))

def phased_fanout(init: Array[Boolean]) = {
  val bits = Range(0, n).map(_ != 0);
  vc(WIRE(">","v>"),
     vrep(n-2, i => WIRE("v",">v/%", init(i))),
     WIRE("v",">"))
}
def reduce(n: Int, f: Int => Module) = {
  val max_depth = (log(n)/log(2)).toInt;
  vc(reduce2(max_depth, n-1, f),
     hrep(n, () => hc(f(0), noop())))
}

def bit_loop (b: Array[Boolean]) =
  vc(hc(WIRE("<","v/%", b(1)),    WIRE("^","<")),
     vrep(bits.length-2, i =>
     hc(WIRE("v","v/%", b(i+2)), WIRE("^","^"))),
     hc(WIRE("<","v>"),          WIRE(">","^/%", b(0))))
```

This code produces the following multiplier, in which the colored (or white) blocks, from left to right, correspond to select, duplicate, multiply, pad (two blocks), and reduction by a tree of adders:



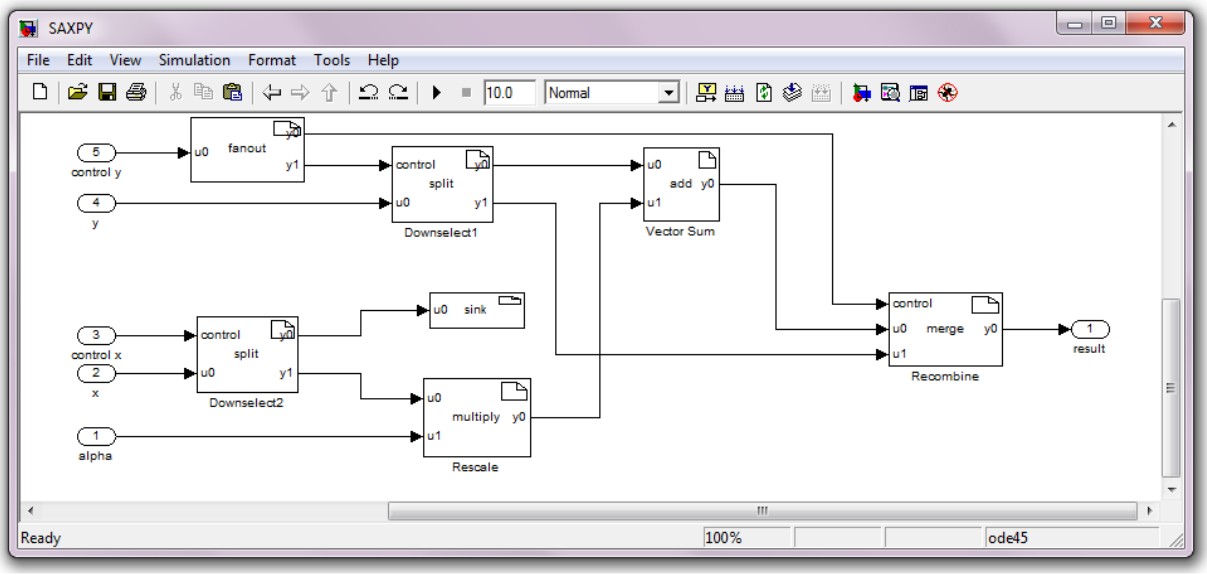
This circuit design and workflow are notable because (i) both function and geometry are apparent from the source code, and (ii) the design is parametric in the word length of the inputs.

SIMULINK ALA

In addition to a textual circuit description languages, we saw a need for a more graphical design process. The workhorse of this process is the direct constructive procedure we developed for taking an arbitrary dataflow graph and embedding it in the plane. For compatibility, we based our tool on reading a dataflow graph from Simulink's file format, augmented with a set of distinguished blocks to represent the primitive ALA gates. The complete workflow consists of:

1. Designing an ALA circuit in Simulink, and saving an output file.
2. Running a compiler on the Simulink file, which looks for specifically named magic blocks, and produces Snap code based on the graph that it deduces.
3. Run the produced Snap program to generate an actual ALA circuit.

All three steps of this workflow, laying out a scaled vector addition (AXPY) function block, out of primitive blocks:




```
matCons[{{wiremod, wiremod, wiremod, wiremod, wiremod},
{{1, 1}, {2, 3}, {3, 2}, {4, 5}, {5, 4}},
{wiremod, split, fanOut, wiremod},
{{1, 4}, {2, 6}, {3, 3}, {4, 5}, {5, 1}, {6, 2}},
{split, serialMult[5], wiremod, sink},
{{1, 2}, {2, 1}, {3, 3}, {4, 4}},
{wiremod, adder, wiremod},
{{1, 3}, {2, 2}, {3, 1}},
{switchnonblocking}, {1, 1}, {wiremod}}]
```

(On top, the Simulink. Bottom left, Snap. Bottom right, the finished circuit.)

Conclusions

The seedling accomplished its goals of providing ALA:

- visual and textual programming languages
- efficient parallel multicore and microcontroller simulators
- libraries for arithmetic and BLAS (basic linear algebra routines)
- CMOS cell library with a one-to-one mapping from an ALA program
- performance modeling with favorable comparisons

These encouraging results have a number of natural next steps:

- chip fabrication to validate the design development and transistor-level modeling
- development of an ALA LAPACK library over the seedling's BLAS routines
- implementation of system solutions that would benefit from ALA, such as embedded ports of mature HPC applications that are limited by power, size, cost, or weight

Along with these development areas, the seedling has raised research questions including:

- *A*A*: rather than tokens representing logic states, they could be simplified to just presence and absence, or generalized to pass words. What level of token (and corresponding cell) complexity is appropriate for what applications?

- *virtualization*: ALA commits a cell for each bit stored. This allows arbitrary integration of computation with storage, but adds significant overhead over DRAM. How can speed be traded off against density with virtualized cell processors updating local lattice patches?
- *reconfiguration*: the seedling did not pursue in-band reconfigurability because of the overhead added by RALA. For many applications this will be essential; preliminary results are promising for a model of reconfiguration that is a discrete analog of synthetic biology. How can reconfiguration be introduced to transport cells as well as states?
- *fabrication*: an ALA design is realized in a small set of cells, each containing tens of transistors. Can these be separately fabricated and singulated and then placed on demand to produce an ASIC in an afternoon?
- *device physics*: there are a number of physical mechanisms that perform conditional operations that are analogous to ALA token-passing, including dark states in cavity QED, electron transport through a Coulomb blockade, and mechanical bistability. How can ALA be implemented with less overhead by taking advantage of more device physics?

Publications

Submitted

Reconfigurable Asynchronous Logic Automata. Neil Gershenfeld, David Dalrymple, Kailiang Chen, Ara Knaian, Forrest Green, Erik D. Demaine, Scott Greenwald, and Peter Schmidt-Nielsen, POPL'10, January 17-23, Madrid, Spain (2010).

Matrix Multiplication with Asynchronous Logic Automata. Scott Greenwald. Master's thesis, Massachusetts Institute of Technology, 2010.

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

Draft

Asynchronous Logic Automata ASIC Design. Kailiang Chen, Forrest Green, Neil Gershenfeld. To be submitted to *IEEE Trans. Computer-aided design of integrated circuits and systems*, 2011.

ALA: Process Independent Verifiable Logic. Peter Schmidt-Nielsen, Kailiang Chen, Neil Gershenfeld. Manuscript, 2011.

Snap: A language for dataflow and layout of asynchronous logic automata. Jonathan Bachrach, Scott Greenwald, Bernhard Haeupler. Manuscript, 2011.

Efficient profiling, simulation, and modular assembly of circuits in ALA. Scott Greenwald, Bernhard Haeupler, Forrest Green, Peter Schmidt-Nielsen. Manuscript, 2011.

Mathematical operations with fine-grained pipelining in ALA. Scott Greenwald, Bernhard Haeupler, Neil Gershenfeld. Manuscript, 2010.