Reconfigurable Asynchronous Logic Automata (RALA)

Reconfigurable Asynchronous Logic Automata

(RALA)

Neil Gershenfeld David Dalrymple Kailiang Chen Ara Knaian
Forrest Green Erik D. Demaine Scott Greenwald Peter Schmidt-Nielsen

MIT Center for Bits and Atoms
gersh@cba.mit.edu

Abstract

Computer science has served to insulate programs and programmers from knowledge of the underlying mechanisms used to manipulate information, however this fiction is increasingly hard to maintain as computing devices decrease in size and systems increase in complexity. Manifestations of these limits appearing in computers include scaling issues in interconnect, dissipation, and coding. Reconfigurable Asynchronous Logic Automata (RALA) is an alternative formulation of computation that seeks to align logical and physical descriptions by exposing rather than hiding this underlying reality. Instead of physical units being represented in computer programs only as abstract symbols, RALA is based on a lattice of cells that asynchronously pass state tokens corresponding to physical resources. We introduce the design of RALA, review its relationships to its many progenitors, and discuss its benefits, implementation, programming, and extensions.

Categories and Subject Descriptors C.1.3 [*Other Architecture Styles*]: Adaptable Architectures

General Terms Design, Languages, Performance, Theory

Keywords Reconfigurable, Asynchronous, Logic, Automata

1. Introduction

Computers are increasingly distributed systems, from multi-core processors to multi-processor servers to multi-server systems. However, this physical reality is typically not reflected in how they are programmed. Common distinctions among programming languages include whether they are imperative or declarative, compiled or interpreted, or strongly or weakly typed. However, these distinctions all assume that computation happens in a world with different rules from those that govern the underlying computing machinery. Each of these program types manipulates arbitrary symbols, unconstrained by physical units.

This is a fiction that is increasingly difficult to maintain. Because the computational descriptions and physical realities diverge, ever-faster switches, networks, and memory have been required to eliminate interconnect bottlenecks, and the productivity of both programmers and processors in large systems has not kept pace with the underlying device performance.

POPL'10, January 17–23, 2010, Madrid, Spain.

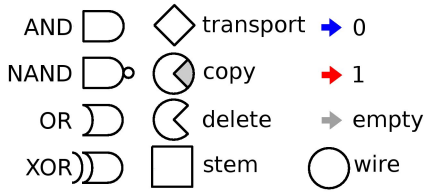

Figure 1. RALA cells and token states. For clarity, AND cells with duplicated inputs used for logical wires transporting and buffering tokens are drawn as circles.

The one thing that most programming models don't describe is space: geometry is bound late in the scheduling of a program's execution. This division dates back to the theoretical foundations of computation; Turing machines [37] and von Neumann architectures [40] localize information processing in a logical unit, without regard to its spatial extent. There is an equally-long history of geometrical descriptions of computation, from Wang tiles [42], to cellular automata [41], to billiard balls [25]. However, these models have primarily been aimed at modeling rather than implementing logic. Although special-purpose computers have been developed for spatial computing models, they've typically been emulated rather than equivalent to the underlying hardware [36].

Physical dynamics, in comparison, are inherently distributed, parallel, asynchronous, and scalable [22]. RALA (Reconfigurable Asynchronous Logic Automata) seeks to capture these attributes by building on a foundation that aligns computational and physical descriptions. Computation, communication, and storage then become derived rather than assumed system properties.

RALA is based on a lattice of cells (in 2D or 3D) that locally pass tokens representing logical states [13]. In the implementation presented here there are 8 cell types, shown in Figure 1 and with functions described below. Four of the cells are used for logic, two for creating and destroying tokens, one for transporting tokens, and one for reconfiguration. When these have valid tokens on their inputs and no tokens on their outputs they pull the former and push to the latter.

In RALA, the distance that information travels is equal to the time it takes to travel (in gate delay units), the amount of information that can be stored, and the number of operations that can be performed. These are all coupled, as physical units are.

RALA can be understood as the end-point, and intersection, of many existing computational paradigms, building the essential elements of each into every cell:

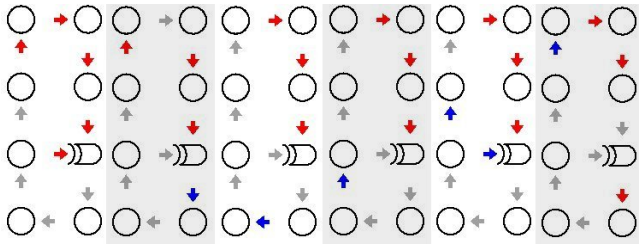

Figure 2. Asynchronous operation of a RALA LFSR.

- It is similar to cellular automata, but eliminates the unphysical need for a global clock, and does not require many cells for logic [5] or deterministic asynchronous operation [2].
- It is a Petri Net [30] on a lattice with nearest-neighbor connections.
- It is a form of RISC [28], where the instructions are reduced to a single logical operation.
- It is a multicore processor that effectively has a core for every bit.
- It is a field-programmable gate array with single-gate logic blocks and nearest-neighbor switch matrices; it does not need a clock, because of its homogeneity a fitter is not needed to map a logic diagram onto corresponding device resources, and a larger program can transparently be distributed across multiple smaller chips.
- It is a systolic array [21] that allows for arbitrary data flow and logic.
- It is a form of dataflow programming [3, 14, 20] in which the graph itself is executable, without requiring event scheduling.
- It is a form of reconfigurable [24] asynchronous [32] logic [8] in which a logical specification is equal to its asynchronous implementation because each gate implements deterministic asynchronous operation.

The following sections discuss RALA operation, benefits, implementation, programming, and extensions.

2. Operation

Logic: The RALA implementation used here has four logical cell types, AND, NAND, OR, and XOR. NAND alone is universal [19]; the others are included to simplify logic design. Figure 2 shows the operation of an XOR cell in a Linear Feedback Shift Register (LFSR), with the feedback loops carried by AND cells with duplicated inputs.

Transport: in Figure 2 the AND cells with duplicated inputs serve two purposes, moving tokens for interconnect, and buffering them for delays. However, not all of these are needed as buffers; RALA includes a non-blocking transport cell that connects its input and output physically rather than logically, reducing the overhead in time and energy for the latter. The transport cell is also used as a crossover, to allow tokens to pass in perpendicular directions.

Copy and Delete: RALA cells implicitly create and destroy tokens when they perform fan-in and fan-out operations, however asynchronous algorithms can require explicit token creation and destruction when controlled changes in token numbers are necessary. A token at the designated data input of a copy or delete cell is passed unchanged if the control input token is a 0; a 1 control input causes the copy cell to replicate a data token, and a delete cell to consume it.

fold direction:	forward=000 backward=001 right=010 left=011 up=101 down=110 end=111
gate:	and=000 nand=001 or=010 xor=011 transport=100 copy=101 delete=110 stem=111
input 1:	forward=000 backward=001 right=010 left=011 up=101 down=110 not used=111
init 1:	0=00 1=01 x=10
input 2:	forward=000 backward=001 right=010 left=011 up=101 down=110 not used=111
init 2:	0=00 1=01 x=10 stay connected=11
output directions:	forward backward right left up down: 0=off 1=on

Table 1. RALA stem cell configuration code; a right-hand rule coordinate system is assumed.

Reconfiguration: because RALA aligns the description of hardware and software, it cannot assume the presence of an omnipotent operating system with global access to system resources. Instead, control of program execution must be implemented locally, within the system. RALA accomplishes this with "stem" cells, which, like their biological counterparts, can be differentiated into other cell types. All RALA cells have internal state to store their configuration (cell type and inputs); stem cells expose this state so that it can be set by input tokens. Program geometry is specified by the universality of folding linear paths [12]. A linked string of stem cells pass data tokens to the terminal cell, which interprets them according to the code in Table 1. These will cause the terminal cell to either differentiate and detach, with its input cell becoming the new terminal cell, or to extend the string in a direction specified relative to its input, linking to a neighboring cell that is converted back to the stem cell state and becomes the new terminal cell. Figure 3 shows the growth of the LFSR in Figure 2. Because each differentiated cell waits for valid inputs and outputs, the circuit turns on consistently without requiring global communication or coordination [27].

3. Benefits

Each RALA cell provides a primitive unit of time, space, state, and logic. Benefits of this uniformity include:

Simplicity: a program is completely specified by the locations and connections of eight cell types.

Portability: any process technology that can perform cell updates can execute the same RALA program, whether tokens are passed between system servers, server processors, processor cores, microcontroller arrays, CMOS cells, or atom lattices.

Scalability: the only design rules are within each cell; as long as cells are updated locally, the system will operate correctly globally.

Flexibility: because each cell can buffer, transform, and transfer tokens, a system's interconnect, memory, and processing can be dynamically allocated based on applications and workloads. Word sizes for memory addressing or arithmetic operations can be adapted to a program's needs.

Performance: RALA execution times are determined by the distance information travels, not clock cycles, so that, for example, both sorting and matrix multiplication become linear-time algorithms (Figures 5 and 7). The speed of logical operations is

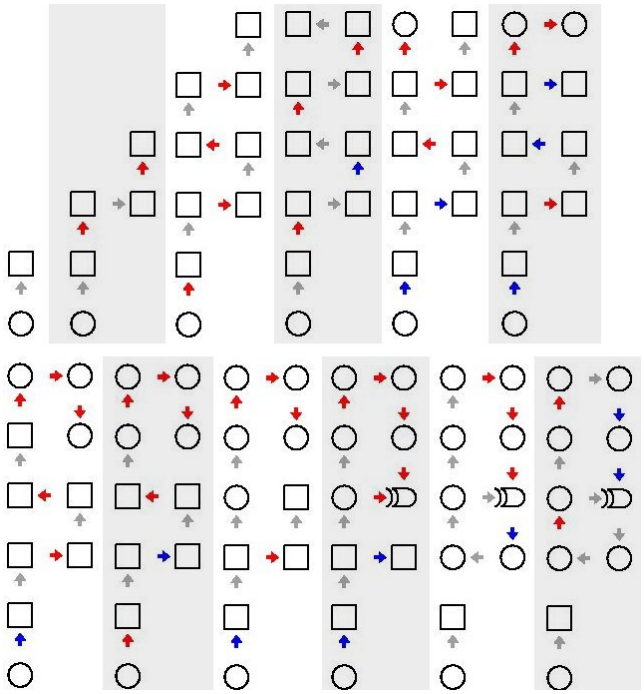

Figure 3. Asynchronous steps in the RALA construction of an LFSR from a source cell streaming tokens through a stem cell.

the local gate propagation delay, rather than a worst-case global clock.

Security: Once they're configured, ALA programs run on what is effectively special-purpose hardware rather than a general-purpose processor; attacks that are based on shared resources, such as buffer overruns, are prevented by spatially separating programs.

Power: there is no static power consumption, other than leakage; power is needed only for token transitions. Although leakage is an increasingly severe limit as gates scale down in feature size, RALA can instead grow up in system size because it is not limited by clock domains or chip boundaries.

Yield: because RALA is constructed from simple cells, a fabrication process need produce only hundreds rather than hundreds of millions of transistors, and then repeat these units. Faulty cells can be accommodated by adding spatial error-correction to RALA programs. This significantly relaxes today's severe constraints on optical flatness, planarization, and defect density, allowing existing processes to be used much further along the semiconductor scaling roadmap [29].

Programmability: conventional computer programming requires resource allocation. In a multi-threaded system this entails spawning and synchronizing threads, in a multi-processor system assigning processes to processors, or in a dataflow system scheduling and executing messages. These actions are implicit in the design of a RALA program, because they are effectively performed by each cell.

Verification: the cost of large-scale full-custom ASIC design has become prohibitive for all but the most important applications, projected to past $100 million by the 32-nm node [15]. Taping out a chip requires a substantial effort to validate the design, including transistor sizing, transition timing, clock skew, crosstalk, and parasitics [18]. RALA divides this into software simulation of a system and the much simpler hardware task of verifying the operation of a single cell. An added benefit is that a standard cell library of RALA cell types can be used to immediately convert a RALA program into an equivalent special-purpose ALA ASIC.

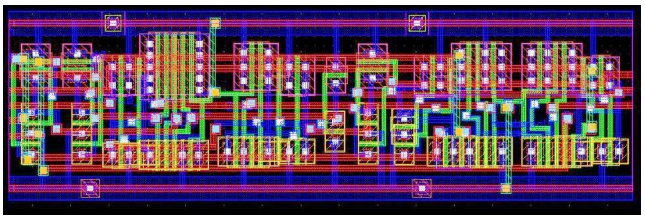

Figure 4. CMOS ALA AND cell.

4. Implementation

In a conventional processor, the logic of updating a RALA cell requires roughly 10 instructions. Therefore, a $100 microprocessor that executes 10^9 instructions per second can update 10^8 cells per second; at 10 W that corresponds to 10^{-7} J per cell update and $\$10^{-6}$ per cell/s. For a $1 microcontroller that executes 10^7 instructions per second at 10 mW, the corresponding numbers are 10^{-8} J per cell update and again $\$10^{-6}$ per cell/s.

A non-reconfigurable ALA cell can be implemented with roughly 100 transistors (shown in Figure 4), and a reconfigurable RALA cell with about 1000, in dual-rail CMOS logic; in a 90 nm design this requires on the order of 10^{-13} J to update a cell, with a propagation time of 1 ns [11]. Assuming a comparable cost of $\$10^{-6}$ per transistor [23], this is $\$10^{-14}$ per cell/s. While these estimates have ignored prefactors, the many orders-of-magnitude difference in energy and cost are due to corresponding differences in the number of transistors needed to natively implement RALA *vs* simulate it on a general-purpose computer.

RALA might appear to incur overheads due to the absence of dedicated interconnect or functional units. State-of-the-art gate delays are on the order of ps over micron features [35], corresponding to ns to cross mm-scale chips with logic rather than wires, comparable to current clock speeds. The largest HPC systems consume about 10 MW for 1 petaflop/s, or 10^{-8} J/flop; at 0.1 pJ/cell and assuming a 100 bit word size, a RALA adder requires approximately 10 cells for 10^{-10} J per add, and a multiplier 1000 cells for 10^{-8} J for a (linear-time) multiply. And at 100 transistors for a non-reconfigurable cell and 1000 transistors for a reconfigurable cell, area can be traded off against speed by virtualization with asynchronous cell processors containing memory and event queues, to reduce the number of transistors per cell to order unity.

5. Programming

The lowest level of RALA programming is explicit placement of cells types and connections. This is like full-custom logic, but is immediately executable; the preceding figures can be taken as executable programs.

A cell-level description misses the functional organization of a RALA program; one way to convey this is with a hardware description language. Figure 5 shows a linear-time RALA sort; here are sections of the definitions of its modules:

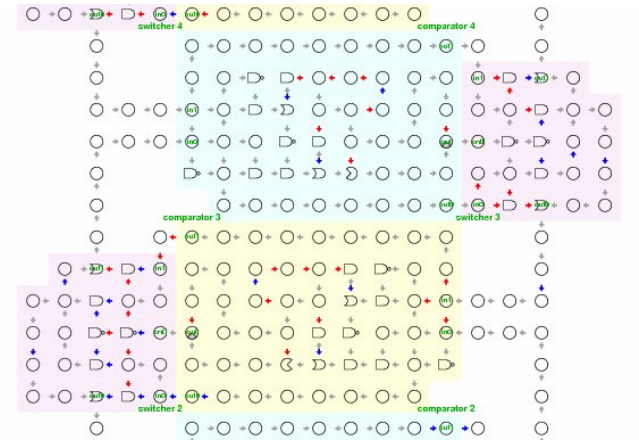

Figure 5. Section of a linear-time RALA sort, interleaving comparators and switchers.

```
#
# upper 10 detector
#
gate(l,'transport','wxsx',1,3)
gate(l,'wire','sx',1,4)
gate(l,'and','wxnx',2,3)
gate(l,'nand','wx',2,4)
#
# upper latch
#
gate(l,'or','wxn0',3,3)
gate(l,'and','sxex',3,4)
```

and their connections:

```
c2 = comparator(l,c1.x+c1.dx-1,c1.y+c1.dy,
   label='comparator 2',flip='x',color='#ffffdd')
s2 = switcher(l,c2.x-c2.dx,c2.y,
   label='switcher 2',flip='x',color='#faeefa')
wire(l,'x',
   c2.cell['out0'],
   s2.cell['in0'])
wire(l,'x',
   c2.cell['out1'],
   s2.cell['in1'])
```

Because programs are spatial structures, this is similar to the internal representation of a hierarchical parametric CAD system. Figure 6 shows a RALA programming tool built this way, with a CAD-like interface [34].

A RALA CAD diagram can be abstracted as a dataflow graph; because there is a one-to-one map between nodes and links in such a graph and RALA modules, dataflow representations of high-level languages [38] can be directly executed as RALA programs.

There is also a direct mapping of mathematics. Figure 7 shows linear-time RALA matrix multiplication, implementing the flow of information through the matrices. Along with the spaces represented by mathematical symbols, there is also a space occupied by mathematical symbols on a page; RALA can be used to solve problems in this space by converting the equations onto corresponding spatial structures [17]. This is computationally universal, providing a constructive implementation of constraint programming [33].

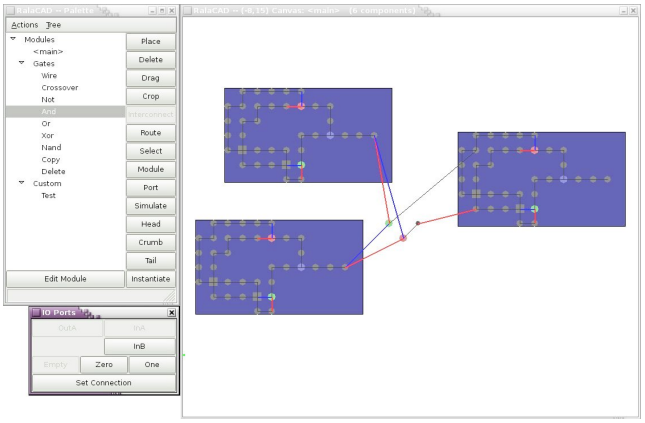

Figure 6. RALA CAD.

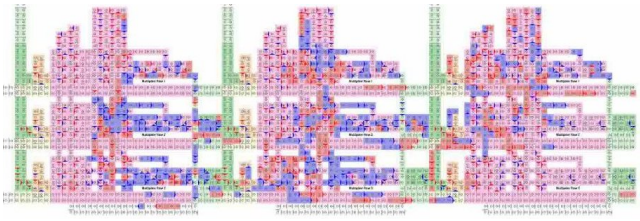

Figure 7. RALA linear-time matrix multiplication, with an array of adders and linear-time multipliers.

6. Extensions

RALA assumes a regular lattice in 2D or 3D, reflecting the construction of the computational substrate. It could be generalized to probabilistic operation with random cell locations and connections [1, 9], however the regularity of the lattice does not represent a significant scaling cost in fabrication; there are a number of high-throughput processes that can produce arrays of simple cells, including soft lithography, self-assembly, parallel direct-write, and printing digital materials [31].

Mixed-signal applications such as software radios conventionally digitize analog signals which are then processed digitally. Power can be reduced, and speed and noise margins increased, for these applications by bit-slicing at the end rather than beginning of the signal chain. Analog logic uses continuous device degrees of freedom to represent bit probabilities rather than bits; it operates in the state-space of the corresponding digital problem, but relaxes the binary constraint to be able to pass through the interior of the logical hypercube [39]. Programmability can be introduced with an array of soft cells (shown in Figure 8 [10]), which can be extended to RALA with asynchronous event logic. Real-time constraints can be enforced by the external timing of token creation or consumption.

There are a number of quantum systems that naturally implement the conditional asynchronous dependency of RALA cells, including multi-photon transitions in cavity QED [26], and electron passage through a Coulomb blockade [4]. Implementing RALA over these systems offers an alternative paradigm for quantum computing that is aligned with device physics. Because of the need for unitary operation, the universal logical cells would be replaced with single-bit rotations and a nonlinear two-bit interaction [6], and the token creation and destruction cells correspond to raising and lowering operators.

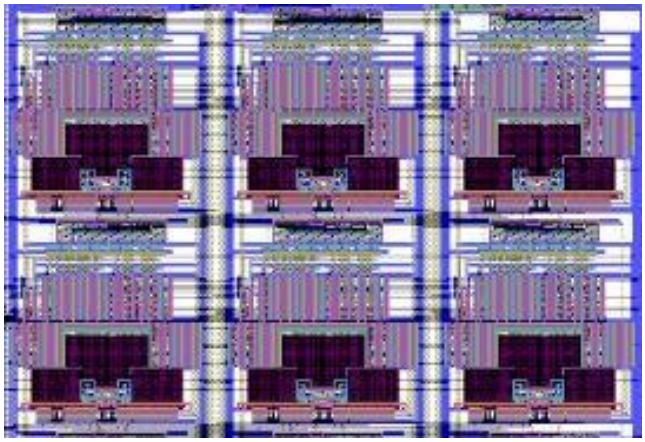

Figure 8. Analog logic automata cells.

RALA could also be implemented with classically reversible logic cells [16] to eliminate the energetic cost of creating and destroying tokens in logical operations. However, this does require either steady-state bit sources and sinks or running results back from outputs to inputs [7]. Non-reversible RALA still operates by transporting tokens, localizing the energetic cost in token creation and destruction.

The CMOS implementation described in the previous section stores configuration in static memory, and transfers tokens with logical handshaking. The transistor count can be reduced by using device physics for these functions, storing configuration on floating gates or in phase-change materials, and handshaking by sensing charge packets.

7. Conclusion

Although RALA is motivated by scaling limits, it can also be implemented over existing systems as an alternative programming paradigm offering simplicity and portability. Ongoing work is developing versions targeting low-level message passing on high-performance computing clusters, arrays of microcontrollers, ASICs, and nanostructures. Potential benefits include reducing power consumption, increasing speed, building larger systems, using smaller devices, adapting to workloads, and simplifying programming.

RALA should be understood not as a new programming model, but rather as the asymptotic limit of many familiar ones. But in this limit their intersection aligns physical and computational descriptions, providing an opportunity to revisit assumptions that date back to the origins of modern computing.

Acknowledgments

The development of RALA builds on the work of, and has benefited from discussions with, colleagues including Charles Bennett, Seth Lloyd, Marvin Minsky, Tom Toffoli, Norm Margolus, Bill Butera, and Mark Pavicic. This work was supported by MIT's Center for Bits and Atoms and the U.S. Army Research Office under grant numbers W911NF-08-1-0254 and W911NF-09-1-0542.

References

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. F. Knight, Jr., R. Nagpal, E. Rauch, G. J. Sussman, and R. Ron Weiss. Amorphous computing. *Commun. ACM*, 43:74–82, 2000.

[2] S. Adachi, F. Peper, and J. Lee. Computation by asynchronously updating cellular automata. *Journal of Statistical Physics*, 114:261–289, 2004.

[3] Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews of Computer Science*, pages 225–253, 1986.

[4] D. Averin and K. Likharev. Coulomb blockade of single-electron tunneling, and coherent oscillations in small tunnel junctions. *J. Low Temp. Phys.*, 62:345–373, 1986.

[5] R. Banks. *Information Processing and Transmission in Cellular Automata*. PhD thesis, MIT, 1971.

[6] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–67, 1995.

[7] C. Bennett. Time/space trade-offs for reversible computation. *Siam J. Comput.*, 18:766–776, 1989.

[8] G. Birtwistle and A. Davis, editors. *Asynchronous Digital Circuit Design*. Springer-Verlag, New York, 1995.

[9] W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, 2002.

[10] K. Chen, L. Leu, , and N. Gershenfeld. Analog logic automata. In *Proceedings of the IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 189–192, 2008.

[11] K. Chen, F. Green, and N. Gershenfeld. Asynchronous logic automata asic design. Preprint, 2009.

[12] K. C. Cheung, E. D. Demaine, and S. Griffith. Programmed assembly with universally foldable strings. Preprint, 2009.

[13] D. Dalrymple, N. Gershenfeld, and K. Chen. Asynchronous logic automata. In *Proceedings of AUTOMATA 2008*, pages 313–322, 2008.

[14] J. Dennis. Data flow supercomputers. *Computer*, 13:48–56, 1980.

[15] EE Times, 2009. March 17.

[16] M. Frank, C. Vieri, M. J. Ammer, N. Love, N. Margolus, and T. Knight, Jr. A scalable reversible computer in silicon. In C. Calude, J. Casti, and M. Dineen, editors, *Unconventional Models of Computation*, pages 183–200, Berlin, 1998. Springer.

[17] S. Greenwald, B. Haeupler, and N. Gershenfeld. Mathematical operations in asynchronous logic automata. Preprint, 2009.

[18] S. Hassoun and T. Sasao, editors. *Logic Synthesis and Verification*. Kluwer Academic Publishers, Norwell, MA, 2002.

[19] F. J. Hill and G. R. Peterson. *Computer Aided Logical Design with Emphasis on VLSI*. Wiley, New York, 4th edition, 1993.

[20] W. M. Johnston, J. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36:1–34, 2004.

[21] S. Kung. Vlsi array processors. *IEEE ASSP Magazine*, pages 4–22, 1985.

[22] S. Lloyd. Ultimate physical limits to computation. *Nature*, 406: 1047–1054, 2000.

[23] W. Maly. Cost of silicon viewed from vlsi design perspective. In *DAC '94: Proceedings of the 31st annual Design Automation Conference*, pages 135–142, 1994.

[24] R. Manohar. Reconfigurable asynchronous logic. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 13–20, 2006.

[25] N. Margolus. Physics-like models of computation. *Physica D*, 10: 81–95, 1984.

[26] R. Miller, T. Northup, K. Birnbaum, A. Boca, A. Boozer, and H. Kimble. Trapped atoms in cavity qed: Coupling quantized light and matter. *J. Phys. B: At. Mol. Opt. Phys.*, 38:S551–S565, 2005.

[27] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, 1967.

[28] D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8:25–33, 1980.

[29] P. S. Peercy. The drive to miniaturization. *Nature*, 406, 2000.

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

[31] G. Popescu, P. Kunzler, and N. Gershenfeld. Digital printing of digital materials. In *DF 2006 International Conference on Digital Fabrication Technologies*, 2006. Denver, Colorado.

[32] Proceedings of the IEEE. Special issue on asynchronous circuits and systems, 1999. 87:2.

[33] J.-F. Puget and I. Lustig. Program does not equal program: Constraint programming and its relationship to mathematical programming. *Interface*, 31:29–53, 2001.

[34] P. Schmidt-Nielsen. RALA CAD. To be published, 2009.

[35] M. Sokolich, A. Kramer, Y. Boegeman, and R. Martinez. Demonstration of sub-5 ps cml ring oscillator gate delay with reduced parasitic alinas/ingaas hbt. *IEEE Electron Device Letters*, 22:309–311, 2001.

[36] T. Toffoli and N. Margolus. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, Cambridge, MA, 1991.

[37] A. Turing. Computing machinery and intelligence. *Mind*, 59:433–560, 1950.

[38] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. In *IEEE 13th International Workshop on Logic Synthesis (IWLS)*, Temecula, CA, June 2004.

[39] B. Vigoda, H. Dauwels, M. Frey, N. Gershenfeld, T. Koch, H.-A. Loeliger, and P. Merkli. Synchronization of pseudo-random signals by forward-only message passing with application to electronics circuits. *IEEE Transactions of Information Theory*, 52:3843–3852, 2006.

[40] J. von Neumann. First draft of a report on the EDVAC, 1945.

[41] J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, 1966. Edited and completed by Arthur W. Burks.

[42] H. Wang. Proving theorems by pattern recognition – II. *Bell System Tech. Journal*, 40:1–41, 1961.