Computing (2011) 93:91–102
DOI 10.1007/s00607-011-0160-1

Aligning the representation and reality of computation with asynchronous logic automata

Neil Gershenfeld

Received: 15 October 2011 / Accepted: 24 October 2011 / Published online: 16 November 2011

Abstract There are many models of computation, but they all share the same underlying laws of physics. Software can represent physical quantities, but is not itself written with physical units. This division in representations, dating back to the origins of computer science, imposes increasingly heroic measures to maintain the fiction that software is executed in a virtual world. I consider instead an alternative approach, representing computation so that hardware and software are aligned at all levels of description. By abstracting physics with asynchronous logic automata I show that this alignment can not only improve scalability, portability, and performance, but also simplify programming and expand applications.

Keywords Logic automata · Spatial computing · Programming models · Parallel computing · Computer architecture

Mathematics Subject Classification (2000) 65F99 · 65Y04 · 65Y05 · 65Y10 · 68N15 · 68N17 · 68Q05 · 68W10 · 68W35

1 Introduction

There are many models of computation: imperative versus declarative versus functional languages, SISD versus SIMD versus MIMD architectures, scalar versus vector versus multicore processors, RISC versus CISC versus VLIW instruction sets. But there is only one underlying physical reality, accurately described by the Standard Model and general relativity from subatomic to cosmological scales.

N. Gershenfeld (✉)
The Center for Bits and Atoms, Massachusetts Institute of Technology,
Cambridge, MA, USA
e-mail: gersh@cba.mit.edu

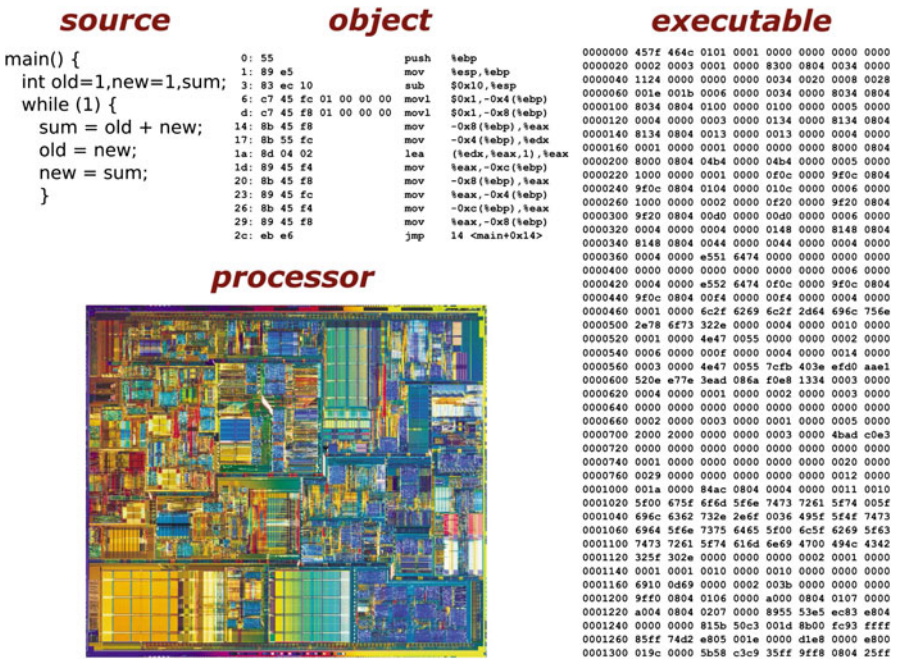

Fig. 1 Conventional representations of the calculation of Fibonacci numbers

This difference between the description and the reality of computation is leading to increasingly severe scaling issues, in order to maintain the fiction that the digital world is not physical. Ever-faster busses and networks are needed to avoid interconnect bottlenecks in topologies that differ from the actual dimensionality of space. Processors might occupy vertices of a higher-dimensional hypercube; memory access might be along a lower-dimensional index. If such a system is really growing in 3D there will be a divergence in the resources needed to map between the virtual and physical spaces. Programs that are described as a series of operations for a processor to follow require an optimization in compilation to make effective use of multiple processor cores that grows with the size of the program and the number of cores. Power is required to do nothing as well as to do something, unlike the origin of dissipation in irreversible processes.

Consider as an example the calculation of Fibonacci numbers. Figure 1 shows four representations of the algorithm, C source code, an object file produced by compiling it, an executable file targeted for a processor, and the processor that will execute it. These representations serve as abstractions that hide knowledge of the lower layers from those above.

Compare this to Fig. 2, zooming a map. There is again a hierarchy, from a building to a world, but the geometry does not change with magnification—the abstractions represent and respect the layers below them.

What if computation likewise respected physics, so that it was possible to zoom from software to hardware without changing geometry, and they scaled in exactly the

Fig. 2 Hierarchical structure of a zoom through a map

same way? Physics is the ultimate model of computation; nature is computationally universal [1,2]. What would computer science look like if it was not considered to be divorced from physical science?

This paper will provide an illustration of what answers to these questions might look like, based on asynchronous logic automata (ALA). Subsequent sections will review background history, introduce the ALA model, look at how programs can be written, describe implementations and their performance, and examine implications.

2 History

The origin of the divergence between computer science and physical science can be traced to the origin of computing. The theory of computational universality is based on Turing's machine [3], which has a head that processes symbols on a tape. These elements appear in von Neumann's prevailing architecture [4] as organs for memory and control.

While these are simplified abstractions that have had a great impact, they also reflect prevailing historical conceptions of what a machine or organism is. It is now appreciated that biological information processing is not localized but rather is distributed, from gene expression to regulatory networks to neural spiking to synaptic connectivity to brain morphology to social relationships. A faithful biologically-inspired model of computation would retain this fine-grained integration of the storage and manipulation of information, across a hierarchical modular construction.

In a Turing machine the head is distinct from the tape; in the laws of physics states are not separated from their interactions. Physical space does appear in theories of computational complexity as a resource to be traded off against time [5], but those theories do not provide insight into how to do that. Space also appears in a dataflow

model as a graph of how information flows, but the space that the graph represents is typically not the same as the one that the hardware occupies.

There is a parallel history of computing that is based on real rather than virtual space. Theoretically, an equivalence was shown between tilings of the plane and Turing machines, with the halting problem corresponding to asking whether a set of tiles can fill space without bound [6]. Space is central to the final problems studied by Turing, morphogenesis in reaction-diffusion systems [7], and von Neumann, self-reproduction in cellular automata [8].

CAs [9] and then PDEs [10] were subsequently shown to be computationally universal, although these were theoretical studies rather than practical proposals. CA computers were later built, programmed by the selection of the rule table and initial conditions rather than by sequential instructions [11]. While offering promising performance with simple hardware, the technological impact of these CA machines was limited by the competition with conventional silicon scaling, by the lack of programs and programmers, and by the requirement for synchronization of the cell updates.

The requirement for a global clock was relaxed in asynchronous CAs, with cells that update at random times. The behaviour of a synchronous CA depends on the order of updates; in an asynchronous CA, delay-insensitive structures distributed over many cells operate independently of their update order [12]. There is a (relatively) independent history of eliminating clocks in IC design with asynchronous logic, which in that context means circuits that are unclocked but deterministic. These are based on modules that enforce logical dependencies, which can be specified with a dataflow representation that is mapped onto reconfigurable logic [13]. The ALA model described in the next section links these approaches, with deterministic asynchronous updates on a regular lattice.

3 ALA

Asynchronous logic automata represent essential physical attributes, so that hardware and software can scale in the same way [14]. These include:

- Propagation: there can not be action at a distance; there is a velocity of information propagation. A model of connectivity that is based on anything other than proximity introduces an implicit size-dependent communication overhead; actual distances are typically hidden in a conventional architecture. ALA is based instead on a locally-connected lattice of cells that communicate by transporting state tokens.
- Information: a finite volume must contain a finite amount of information. In ALA, the links between each cell can be empty or occupied with a token that has a value of 0 or 1.
- Interaction: physically, logic is due to nonlinear interactions in product (or higher order) Hamiltonian terms. Tokens interact at ALA cells; when there are tokens present on a cell's inputs and absent on its outputs, it pulls from the former and pushes to the latter.

The corresponding cell choice is not unique; the version described here has 14 cell types, shown in Fig. 3: AND, NAND, OR, XOR for logic (NAND alone is universal;

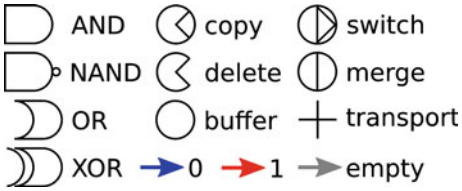

Fig. 3 ALA cells

the others ease layout), copy and delete to selectively create and destroy tokens, buffered and non-buffered transport cells, and merge and switch cells to join and select token sources.

These cells are connected to rectangular nearest neighbors in either 2D or 3D, based on the their actual physical construction. Each cell represents one unit of time, space, state, and logic; these are coupled as they are in the underlying physics. Communication, processing, and storage are all aspects of the same resource; the distance that a token travels is proportional to the time that it takes, the number of operations that can be performed, and the amount of information that can be stored.

ALA shares the token-passing of a Petri Net [15], the graphical construction of a dataflow architecture [16], the array of gates of a Field Programmable Gate Array, the dependencies of an asynchronous circuit [17], and the parallelism of a multicore processor. It differs in explicitly representing space, offering spatial as well as computational universality.

It is possible to relax the assumption of a periodic lattice with models of computing in random media [18,19], however that introduces overhead that's not needed given the availability of low-cost batch processes for nanofabrication, including stamping [20] and printing [21]. ALA can be restricted to reversible logic [22,23]; the CMOS implementation described below does use a dual-rail representation, with cells that explicitly create and consume tokens.

The relationship between a device physics model, an ALA diagram, and a computer program is analogous to that between molecular dynamics, lattice gas automata, and partial differential equations. Both intermediate representations coarse-grain the essential features of the microscopic description, and can be used to reproduce the macroscopic limit, while providing an alternative that can be simpler and more convenient to use, and that avoids historical assumptions.

4 Programming

Returning now to the example of generating Fibonacci numbers (Fig. 1), Fig. 4 shows this being done in ALA. It is based on the gate-level definition of a one-bit full adder. Because of the linkage in ALA between time, distance, storage, and logic, a one-bit adder becomes an arbitrary-precision adder as a number is streamed past. The recursive definition of Fibonacci numbers is realized by linking the output to the inputs, with the starting values and their time ordering provided by the initial token locations.

The Fibonacci example has one level of abstraction, from the gates to the adder. Figure 5 shows a larger example, a dot product that is part of an ALA implementation

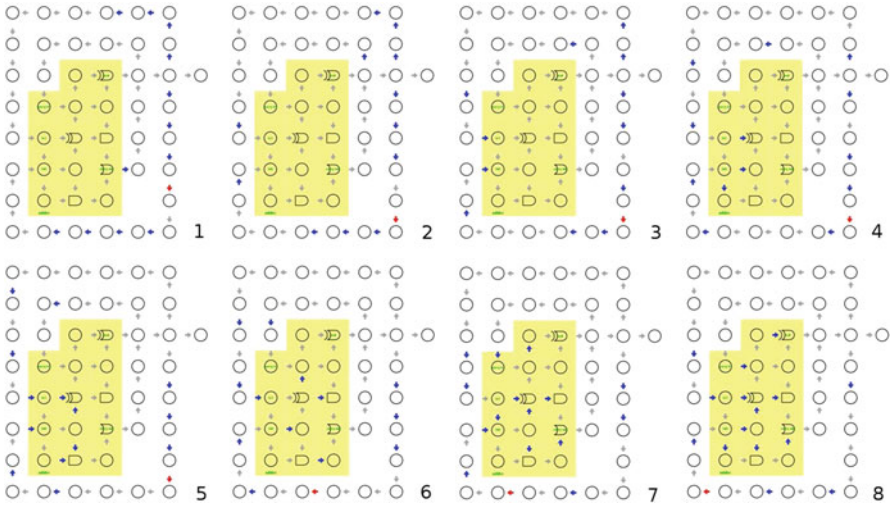

Fig. 4 Steps in the calculation of Fibonacci numbers with an ALA one-bit full adder

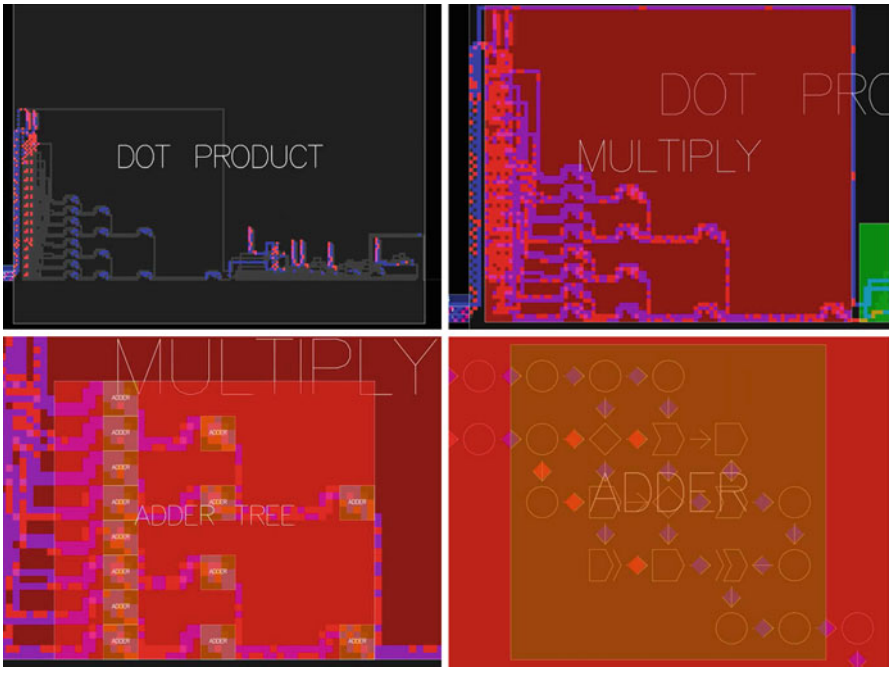

Fig. 5 Hierarchical structure of a zoom through an ALA dot product [24]

of linear algebra [24]. This is a computational equivalent to the zoom in Fig. 2 with a hierarchical structure that respects the spatial organization.

Mathematical operations in ALA require an execution time that's proportional to the distance that information travels [25] rather than the number of operations performed

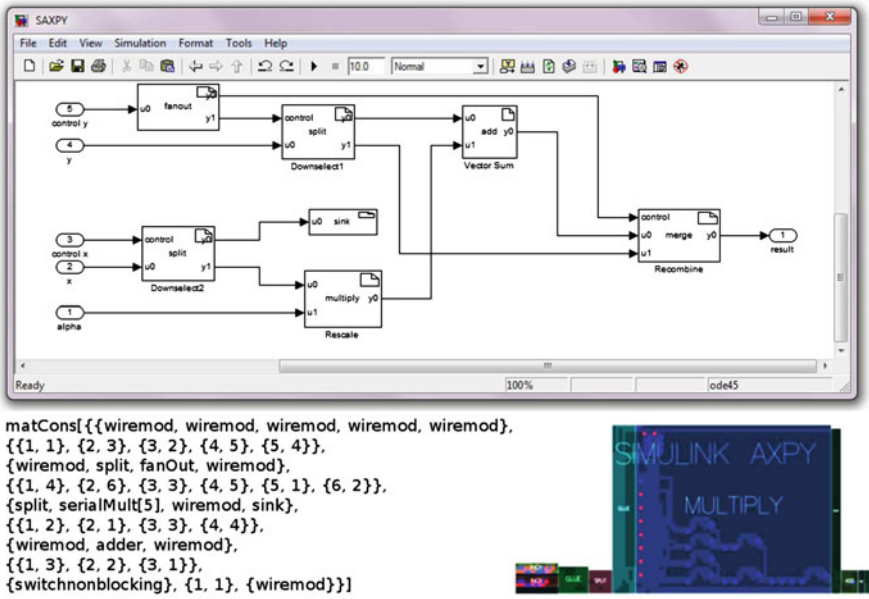

Fig. 6 Visual dataflow ALA programming environment, with hardware description language and ALA cell output [P. Schmidt-Nielsen (2011, Personal communication)]

and messages passed [26]. This means that in return for ALA's overhead (discussed below), many operations such as matrix multiplication and sorting become linear time.

The dot product in Fig. 5 was developed with a hardware description language for ALA, Snap [27]. This is similar in spirit to the use of VHDL or Verilog in an IC design workflow, but rather than exporting a netlist to place-and-route, relative spatial relationships are part of the logical description since an ALA program is specified by its geometry.

An alternative approach to ALA programming is to use a visual dataflow design interface, such as the one shown in Fig. 6 [P. Schmidt-Nielsen (2011, Personal communication)]. While conceptually similar to any dataflow environment, with a visual representation linking modules drawn from a library, there are two essential differences for ALA. There is not an implicit assumption of an external scheduler to manage execution of conventional code; it is self-timed by the cells. And the picture *is* the program: it is possible to zoom from a high-level diagram into individual cells. Like a hierarchical, parametric CAD model, functional relationships are represented in their real-space locations.

The one approach that has not been used for ALA programming is interpretation of existing procedural languages. This is because they typically assume an instruction pointer, and a separation between variables and program statements, both of which violate the physical formulation of ALA and would negate the benefits of its bit-level parallelism.

5 Implementation

ALA programs are portable: any implementation that can perform the local cell updates will operate identically globally, differing only in a tradeoff of speed, cost, size, and power.

One way ALA can be implemented is by emulation with a microprocessor. To avoid the overhead of branch instructions, cell states can be packed into words, all possible bit operations evaluated, and then results projected out in parallel. Estimating performance, a $100 processor with a 1 ns instruction time, 64 bit word, 10 instructions per parallel ALA update, and 10 W power consumption corresponds to 2×10^{-10} s per token update, 2×10^{-9} J per token update, and a combined figure of merit of 2×10^{-8} $·seconds per token update. At this update rate, a bit-serial word operation would take on the order of 10^{-8} s, compared to 10^{-9} for a native instruction.

Finer-grain parallelism is possible by emulation with a microcontroller. Assuming a $1 processor with a 100 ns instruction time, 16 bit word, 10 instructions per parallel update, and 10 mW power consumption, this corresponds to 6×10^{-8} s per token update, 6×10^{-11} J per token update, and a figure of merit of 6×10^{-8} $·seconds per token update. The update rate is reduced because of the slower clock and smaller word size, but the power per token update is also reduced because of the reduction in the total number of transistors that participate in an update.

A native CMOS cell can be implemented with tens of transistors per cell, a few times larger than what is required for a synchronous logic gate. A 90 nm cell library based on handshaking requires on the order of 10^{-13} J per token update, with a throughput of 10^{-9} s per token [28]. Dividing Joules per update by seconds per update gives 10^{-4} W. For a mathematical operation with ∼100 tokens passing through ∼100 cells, that corresponds 10^7 operations per second at 1 W. Thats ∼100× less efficient than state-of-the-art operations per Watt in high-performance computing, again due to the difference between a bit-serial operation and native instruction. The energy per token can be brought down with a smaller feature size, and with a simpler cell design based on linked memory cells.

To price CMOS ALA, the TSMC MOSIS Logic G 90 nm process gives a combined figure of merit of 1.8×10^{-13} $·seconds per token update. This is so much smaller than that for a microprocessor or microcontroller because of the enormous reduction in the number of transistors required for a cell update, which is not a primitive operation in those architectures. That overhead can be reduced still further, at the expense of speed, by storing virtualized cells in DRAM and updating with local cell processors.

A great benefit of ALA in an ASIC workflow is the one-to-one mapping from ALA cells to circuits, illustrated in Fig. 7. With this equivalence, chips can be taped out directly from an ALA design, with performance projections available from token counts in a high-level event simulation combined with cell device modeling [27].

Finally, the conditional dynamics of ALA's token updates can be implemented directly in device physics, with potential mechanisms including Coulomb blockade in a quantum dot [29], and two-photon transitions in a trapped atom in a high finesse optical cavity [30]. ALA's simple assumptions allow its cell states and their interactions to be represented by such individual physical degrees of freedom.

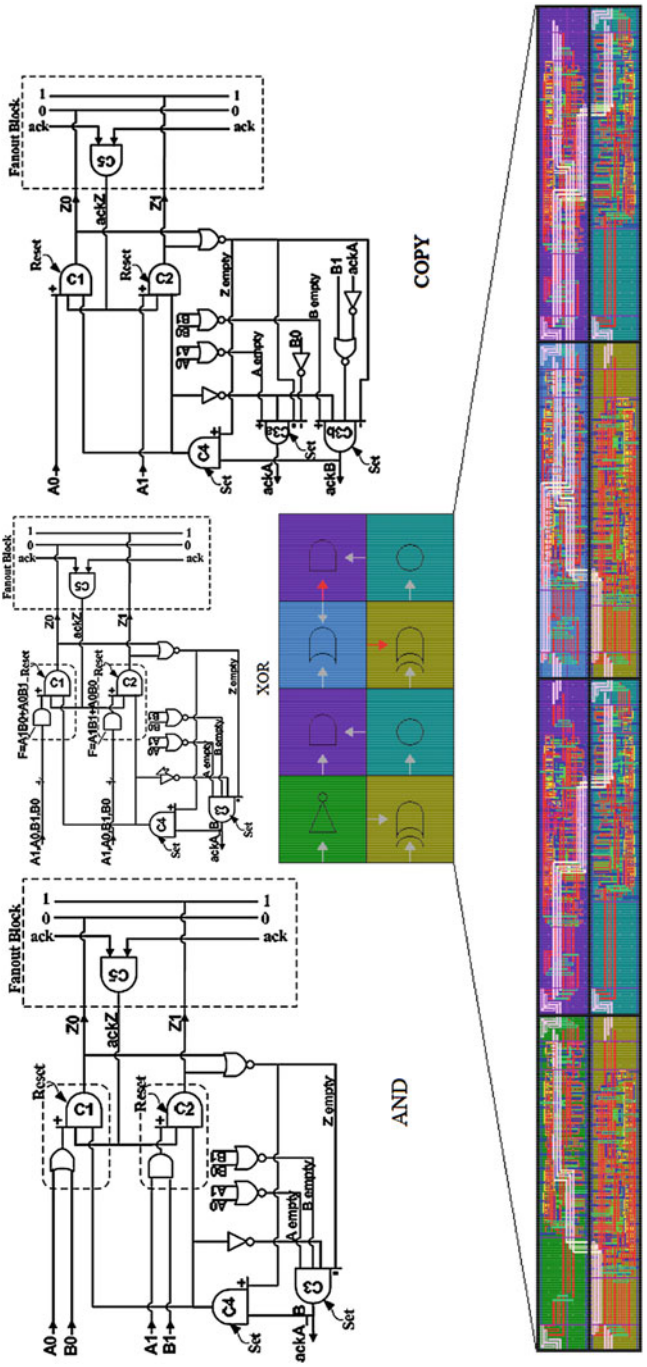

Fig. 7 One-to-one mapping from an ALA design to an ASIC [28]

6 Implications

Any ALA implementation will have an overhead relative to a custom design for a given process and application. This is warranted when the costs of design, verification, and testing dominate lifecycle costs, rather than the need to saturate performance limits. This is increasingly the case for large-scale application IC development. Just as the use of TCP and IP in the Internet is sub-optimal versus custom protocols for particular purposes but has offered tremendous scalability, interoperability, and accessibility, in return for its overhead ALA offers a number of benefits. These include:

Scheduling As seen in Sect. 4, in ALA it's not necessary to explicitly identify process threads, schedule processor time, or manage dependencies in a communication underlay. This does not eliminate the need for timing, but it is intrinsic to the representation rather than imposed by an execution environment.

Portability The implementations described in Sect. 5 can all run the same ALA program. Many languages have been ported to many platforms, but these typically make substantial assumptions about the resources required to support them. To execute ALA, all that is needed is to be able to perform cell updates and communicate locally. This means that ALA programs written now will be useful across a range of future technologies.

Reliability A billion-transistor processor will fail if individual transistors fail, requiring a fabrication process that can yield a billion transistors. Because ALA is assembled from individual cells, it can be built by stepping and repeating units of tens of transistors, allowing for either more aggressive process scaling or greater error margin. Error correction can naturally be introduced spatially, by replicating blocks of cells.

Scalability All of the design rules in an ALA implementation are contained within the cell. As a system grows, the resources available for computing, communications, and storage are all growing at the same rate, rather than introducing size-dependent bottlenecks or a coarse system granularity.

Efficiency ALA designs are less efficient than full-custom designs optimized for a particular task, but for arbitrary problems they run at the speed of the gate delay rather than a fixed clock, and consume power (other than leakage) only when and where there are tokens to process.

Verifiability A control system might be specified in a visual dataflow environment, exported to a high-level language, compiled to a low-level language, and executed on a microarchitecture. Each of these changes in representation can introduce errors, imposing a substantial effort to verify that what gets executed matches the original intention. In ALA the same representation is used throughout, so that as long as the cell updates are performed correctly embedded hardware will match a simulation.

Applicability Because ALA programs are spatial structures, they provide a natural mapping onto spatial problems such as display drivers and distributed control systems,

can be spatially extended to enhance system interfaces for power, heat transfer, and I/O, and can be built with incrementally variable sizes to match workloads.

7 Conclusion

I have reviewed the inspiration, implementation, and implications of the ALA model for computing. It is based on a belief that a fundamental error was made when computer science diverged from physical science; by basing the foundations of the former on those of the latter, hardware and software can be aligned at all levels of description.

I've argued that this is not just possible, it is preferable to address many of the issues in scaling information technologies. Interconnect bottlenecks, challenges in multicore compilation, quiescent power budgets, and diverging chip design costs are all symptoms of trying to compute in a world that differs from the one that we actually live in. These problems are simplified by explicitly recognizing time, space, state, and logic as coupled properties of a medium in which computing happens. These issues don't disappear, but they're moved to where they belong, the design of the problem solution, rather than appearing afterwards as a result of mapping between unrelated descriptions of a program and the platform that executes it.

There are many models of computation, but only one (standard) model of physics. Engineers don't get to pick which physical laws to obey when they work on a problem, but that's effectively what's done when an unphysical model of computation is chosen—the burden of emulating it is pushed off to system architects, compiler writers, and chip designers. That's worked for many years, but will be infeasible to continue to sustain. Instead of imposing increasingly heroic measures to maintain the fiction that software is virtual and hardware physical, their development can coincide with the recognition that both must ultimately satisfy the same constraints.

The ideas embodied in ALA have had a long and frequently parallel history in the development of computing. ALA can be viewed as their convergence, reducing instruction sets to a single logical operation, passing tokens on a graph that represents real space, using logic as the primitive operation of automata cells, implementing asynchronous timing in individual bits. Each of these introduces overhead relative to a solution optimized for a particular application, but the scaling of those optimizations will eventually have to match that of the physics used to implement them. Rather than accommodating that incrementally for each kind of hardware, software, and application, ALA provides an alternative foundation for computation that aligns the representation and reality of computation.

Acknowledgments ALA was developed with David Dalrymple, Ara Knaian, Kailiang Chen, Forrest Green, Scott Greenwald, Erik Demaine, Jonathan Bachrach, and Peter Schmidt-Nielsen, building on the work of Charles Bennett, Norm Margolus, Tom Toffoli, Seth Lloyd, and Isaac Chuang, with support from NSF, DARPA, DTO, and CBA's sponsors. I am grateful to all.

References

1. Margolus N (1984) Physics-like models of computation. Phys D 10:81–95
2. Lloyd S (2000) Ultimate physical limits to computation. Nature 406:1047–1054

3. Turing AM (1950) Computing machinery and intelligence. Mind 59:433–560
4. von Neumann J (1993) First draft of a report on the EDVAC. IEEE Ann Hist Comput 15(4):27–75
5. Lewis HR, Papadimitriou CH (1997) Elements of the theory of computation, 2nd edn. Prentice Hall PTR, Upper Saddle River
6. Chaitin G (2008) The halting probability via Wang tiles. Fundam Inf 86(4):429–433. http://dl.acm.org/citation.cfm?id=1487705.1487708
7. Turing A (1952) The chemical basis of morphogenesis. Phil Trans R Soc Lond B 237:37–72
8. von Neumann J (1956) Probabilistic logics and the synthesis of reliable organisms from unreliable components. In: Shannon C, McCarthy J (eds) Automata studies. Princeton University Press, Princeton pp 43–98
9. Banks RE (1971) Information processing and transmission in cellular automata. PhD thesis, MIT
10. Omohundro S (1984) Modelling cellular automata with partial differential equations. Phys D: Nonlinear Phenom 10(1–2):128–134
11. Toffoli T, Margolus N (1991) Cellular automata machines: a new environment for modeling. MIT Press, Cambridge
12. Lee FPJ, Adachi S, Mashiko S (2005) Delay-insensitive computation in asynchronous cellular automata. J Comput Syst Sci 70:201–220
13. Manohar R (2006) Reconfigurable asynchronous logic. In: Proceedings of the IEEE custom integrated circuits conference, pp 13–20
14. Dalrymple DA, Gershenfeld N, Chen K (2008) Asynchronous logic automata. In: Proceedings of AUTOMATA 2008, pp 313–322
15. Petri CA (1996) Nets, time and space. Theor Comput Sci 153:3–48
16. Arvind, Culler DE (1986) Dataflow architectures. Annu Rev Comput Sci 1:225–253
17. Nowick SM, Josephs MB, van Berkel CH (1999) Special issue on asynchronous circuits and systems. Proc IEEE 87(2):219–222
18. Abelson H, Allen D, Coore D, Hanson C, Homsy G, Knight TFJr, Nagpal R, Rauch E, Sussman GJ, Weiss R (2000) Amorphous computing. Commun ACM 43:74–82
19. Butera WJ (2002) Programming a paintable computer. PhD thesis, MIT
20. Younan X, Whitesides GM (1998) Soft lithography. Annu Rev Mater Sci 28:153–184
21. Ridley B, Nivi B, Jacobson J (1999) All-inorganic field effect transistors fabricated by printing. Science 286:746–749
22. Bennett CH (1973) Logical reversibility of computation. IBM J Res Dev 17:525
23. Margolus NH (1999) Crystalline computation. In: Hey A (ed) Feynman and computation. Perseus Books, Cambridge
24. Greenwald S (2010) Matrix multiplication with asynchronous logic automata. Master's thesis, MIT
25. Gershenfeld N, Dalrymple D, Chen K, Knaian A, Green F, Demaine ED, Greenwald S, Schmidt-Nielsen P (2010) Reconfigurable asynchronous logic automata: (RALA). In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '10, New York, NY, USA. ACM, pp 1–6
26. Ballard G, Demmel J, Holtz O, Schwartz O (2009) Minimizing communication in linear algebra. CoRR, abs/0905.2485
27. Chen K, Green F, Greenwald S, Bachrach J, Gershenfeld N (2011) Asynchronous logic automata asic design (preprint)
28. Green F (2010) ALA ASIC: a standard cell library for asynchronous logic automata. Master's thesis, MIT
29. Grabert H, Devoret MH (eds) (1992) Single charge tunneling: Coulomb Blockade phenomena in nanostructures. Plenum Press, New York
30. Tanji-Suzuki H, Chen W, Landig R, Simon J, Vuleti V (2011) Vacuum-induced transparency. Science 333(6047):1266–1269