Chaogates: Morphing logic gates that exploit dynamical patterns

William L. Ditto,[1] A. Miliotis,[2] K. Murali,[3] Sudeshna Sinha,[4,a)] and Mark L. Spano[5]
[1]*Harrington Department of Bioengineering, Arizona State University, Tempe, Arizona 85287-9309, USA*
[2]*J. Crayton Pruitt Family Department of Biomedical Engineering, University of Florida, Gainesville, Florida 32611-6131, USA*
[3]*Department of Physics, Anna University, Chennai 600 025, India*
[4]*Indian Institute of Science Education and Research (IISER), Mohali, Transit Campus, MGSIPAP Complex, Sector 26, Chandigarh 160 019, India*
[5]*NSWC, Carderock Laboratory, W. Bethesda, Maryland 20817, USA*

(Received 18 July 2010; accepted 25 August 2010; published online 28 September 2010)

Chaotic systems can yield a wide variety of patterns. Here we use this feature to generate all possible fundamental logic gate functions. This forms the basis of the design of a dynamical computing device, a *chaogate*, that can be rapidly morphed to become any desired logic gate. Here we review the basic concepts underlying this and present an extension of the formalism to include asymmetric logic functions. © *2010 American Institute of Physics*. [doi:10.1063/1.3489889]

The patterns of chaos are used to encode and to manipulate inputs so as to produce a desired output. This is accomplished by selecting out desired patterns from the infinite variety offered by a chaotic system. A subset of these patterns is then used to map the system inputs (or initial conditions) to the desired outputs. So this process offers a method to exploit the richness inherent in nonlinear dynamics in order to design computing devices with the capacity to reconfigure into a range of logic gates. The resultant morphing gates are termed *chaogates*. Arrays of such morphing chaotic logic gates can then be programmed to perform higher order functions and to rapidly switch between such functions. So this capacity for reconfigurability will conceivably give us the flexibility of programmable hardware, as well as enable us to obtain the optimization and speed of application specific hardware, within the same computer architecture. Here we briefly review the basic tenets of this computing paradigm and also provide extensions of the idea to obtain asymmetric logic gates.

I. INTRODUCTION

Chaotic systems are renowned for the richness of their dynamics. Whether one describes this richness as a sensitivity to perturbations or as a amalgamation of an infinite number of instabilities, even low-dimensional chaotic systems can express a stunning variety of different behaviors as a function of time, of their initial conditions, or of their parameters.

Here we review an emerging computing paradigm which exploits this pattern generating ability of chaotic dynamics. In 1998 it was noted that chaotic systems may be utilized to design computing devices.[1] In the early years the focus was on proof-of-principle schemes that demonstrated the capability of chaotic elements to do universal computing.[19] In subsequent years there has been much research activity to extend this paradigm.[1–17]

One of the most promising directions of this computing paradigm is its ability to reconfigure a single chaotic element as different logic gates.[2,3] In contrast to a conventional field programmable gate array element,[18] where reconfiguration is achieved by switching between multiple single-purpose gates, reconfigurable chaotic logic gates (RCLGs) are comprised of chaotic elements that morph (or reconfigure) through the control of the pattern inherent in their constitutive nonlinear element. Two input RCLGs have recently been built and shown to be capable of reconfiguring between all logic gates in discrete circuitry.[4–6] Additionally such RCLGs have been realized in prototype very large scale integrated circuits (VLSI) [0.13 μm complementary metal-oxide-semiconductor (CMOS), 30 MHz clock cycles]. Further, reconfigurable chaotic logic gate arrays, which morph between higher order functions such as those found in a typical arithmetic logic unit (ALU), have also been designed.[17] Thus architectures based on such logic implementations may serve as ingredients of a general-purpose reconfigurable computing device more powerful and more fault tolerant[10] than statically wired hardware.

In the following we present the basic concepts underlying chaos computing and describe methods to design nonlinear systems to flexibly yield all fundamental logic functions by "programming" different dynamical systems. We show that the patterns produced by varying the initial conditions and the parameters of a chaotic system, as well as the patterns produced by its time evolution, are amenable to performing computation. By combining both techniques, one can produce a computational device of immense power and elegance.

The outline of this article is as follows. In Sec. II we first recall the implementation of all fundamental logical operations by a threshold control mechanism.[2] In Sec. III we describe the use of time evolution of the nonlinear system to obtain a wide range of logic responses. In Secs. IV and V we

[a)]On leave from The Institute of Mathematical Sciences, CIT Campus, Taramani, Chennai 600113, India.

TABLE I. Truth table of the basic logic operations for a pair of inputs: I_1, I_2 (Ref. 19). The one-input NOT gate is given by NOT(0) is 1; NOT(1) is 0.

I_1	I_2	NAND	NOR	XOR	AND	OR	XNOR
0	0	1	1	0	0	0	1
1	0	1	0	1	0	1	0
0	1	1	0	1	0	1	0
1	1	0	0	0	1	1	1

introduce a more general formalism, capable of implementing *asymmetric* operations that distinguish between the inputs, thus extending the earlier formalism for symmetric gates. We conclude in Sec. VI with a brief discussion of some on-going technological implementations of these ideas, specifically, a VSLI implementation of chaotic computing in a demonstration integrated circuit chip.

II. GENERAL CONCEPT

The cornerstone of modern computer architecture is binary digital logic, the logic of the true and the false. *Boolean* logic is remarkable for its conceptual simplicity. For instance, it can be rigorously shown that any logic operation can be realized by adequate connection of NOR and NAND gates (i.e., *any* Boolean circuit can be built using NOR/NAND gates alone). This implies that the capacity for universal computing can be proven by the implementation of the fundamental NOR and NAND gates.[19]

We outline below a theoretical method for obtaining all basic logic gates with a single nonlinear system. The broad aim here is to use, in a controlled manner, the rich temporal patterns comprising a nonlinear time series in order to obtain a computing device that is flexible and reconfigurable.

Consider the *chaotic chip* or *chaotic processor* to be a one-dimensional system, whose state is represented by x, and whose dynamics is given by a nonlinear map $f(x)$.

In our scheme all the basic two-input logic gate operations (NAND, NOR, XOR, AND, OR, and XNOR) involve the following steps:

(1) The logic inputs I_1 and I_2 for two-input logic operations (see Table I) are encoded by the initial state x_0 of the system as follows:

$$x_0 \rightarrow x_{\text{gate}} + X_1 + X_2,$$

where the physical quantity $X_1(X_2)$ (which could be, for instance, a voltage) has value 0 when logic input $I_1(I_2)$ is 0, and has value δ when logic input $I_1(I_2)$ is 1, with δ being a positive constant. Like X_1 and X_2, x_{gate} is also a physical entity, such as a voltage, which can be varied to yield different logic outputs.

(2) Dynamical evolution over n time steps, resulting in the updated state $x \rightarrow f_n(x_0)$, namely, the nth iterate of the initial state. Specifically we take $n=1$ here, i.e., $x=f^1(x_0) \equiv f(x_0)$.

TABLE II. Necessary and sufficient conditions, derived from the logic truth tables, to be satisfied simultaneously by the nonlinear dynamical element, in order for it to have the capacity to implement the logical operations AND, OR, XOR, NAND, NOR, and NOT (cf. Table I) with the same chaogate.

Logic operation	Input set (I_1, I_2)	Output	Necessary and sufficient condition
AND	(0,0)	0	$f(x_{\text{AND}}) < x^*$
	(0,1)/(1,0)	0	$f(x_{\text{AND}}+\delta) < x^*$
	(1,1)	1	$f(x_{\text{AND}}+2\delta) \geq x^*$
OR	(0,0)	0	$f(x_{\text{OR}}) < x^*$
	(0,1)/(1,0)	1	$f(x_{\text{OR}}+\delta) \geq x^*$
	(1,1)	1	$f(x_{\text{OR}}+2\delta) \geq x^*$
XOR	(0,0)	0	$f(x_{\text{XOR}}) < x^*$
	(0,1)/(1,0)	1	$f(x_{\text{XOR}}+\delta) \geq x^*$
	(1,1)	0	$f(x_{\text{XOR}}+2\delta) < x^*$
NOR	(0,0)	1	$f(x_{\text{NOR}}) \geq x^*$
	(0,1)/(1,0)	0	$f(x_{\text{NOR}}+\delta) < x^*$
	(1,1)	0	$f(x_{\text{NOR}}+2\delta) < x^*$
NAND	(0,0)	1	$f(x_{\text{NAND}}) \geq x^*$
	(0,1)/(1,0)	1	$f(x_{\text{NAND}}+\delta) \geq x^*$
	(1,1)	0	$f(x_{\text{NAND}}+2\delta) < x^*$

(3) Determining the output:

Logic output is 0 if $f(x_0) \leq x^*$

and

Logic output is 1 if $f(x_0) > x^*$,

where x^* is a prescribed reference threshold.

Since the system is strongly nonlinear, in order to specify the initial x_0 accurately, one needs a control mechanism, here a threshold controller[20,21] to set the initial x_0.

In order to obtain all the desired input-output responses of the different gates we need to satisfy the conditions enumerated in Table I simultaneously. So given a dynamics $f(x)$ one must find values of the threshold and initial state satisfying the conditions derived from the truth tables (see Table II).

A representative example is given in Table III, showing the exact solutions for x_{gate} and the threshold x^* that satisfy the conditions in Table II when the dynamical evolution is governed by the logistic equation,

$$f(x) = ax(1-x).$$

Here we choose the nonlinearity parameter $a=4$ and constant $\delta=\frac{1}{4}$ common to all the logic gates.

This is the essence of chaos computing: for the chaotic system of interest, one looks for sets of initial conditions and

TABLE III. One specific set of solutions of the conditions in Table II which yield the logical operations AND, OR, XOR, and NAND with $\delta=\frac{1}{4}$. Note that these theoretical solutions have been fully verified in a discrete electrical circuit implementing a logistic map (Ref. 4).

Operation	AND	OR	XOR	NAND
x_{gate}	0	1/8	1/4	3/8
x^*	3/4	11/16	3/4	11/16

TABLE IV. Necessary and sufficient conditions to be satisfied by a nonlinear system in order to implement the logical operations NAND, AND, NOR, XOR, and OR at different iterations.

Logic	NAND	AND	NOR	XOR	OR
Iteration n	1	2	3	4	5
Input (0,0)	$x_1=f(x_0)>x^*$	$f(x_1)<x^*$	$f(x_2)>x^*$	$f(x_3)<x^*$	$f(x_4)<x^*$
Input (0,1)/(1,0)	$x_1=f(x_0+\delta)>x^*$	$f(x_1)<x^*$	$f(x_2)<x^*$	$f(x_3)>x^*$	$f(x_4)>x^*$
Input (1,1)	$x_1=f(x_0+2\delta)<x^*$	$f(x_1)>x^*$	$f(x_2)<x^*$	$f(x_3)<x^*$	$f(x_4)>x^*$

system parameters that produce a result corresponding to a desired mathematical or logical operation. A given set of parameters may produce a result that is their logical OR of the inputs or the logical AND. These parameters can be stored and used as a ready "look-up table" for future computations using this system. Other combinations of inputs and parameters might lead to results that do not produce a mathematical or logical result. Those initial conditions are discarded. So setting the parameters to be the ones that gave the desired functional temporal patterns constitutes programming of the system to give the right output.

Note that it is possible to implement the concept with any (typical) nonlinear function, and there are proof-of-principle realizations of chaos computing using circuits implementing several different nonlinear maps. For instance, in Ref. 12 two coupled complementary (n-channel and p-channel) junction field-effect transistors efficiently implement a nonlinear function of the form $x_{n+1}=2x_n/(1+x_n^{10})$, which is then used for chaos computing.

Contrast this use of nonlinear elements here with the possible use of linear systems on one hand, and stochastic systems on the other. It is not possible to extract all the *different* logic responses from the *same* element in the case of linear components since the temporal patterns are inherently very limited. So linear elements do not offer much flexibility or versatility. Stochastic elements, on the other hand, have many different temporal sequences. But they are *not deterministic* and so one cannot use them to *design* components. Only nonlinear dynamical systems enjoy both richness of temporal behavior as well as determinism.

Also note that, while *nonlinearity* is absolutely necessary for implementing all the logic gates, chaos may not be always be necessary. In the representative example of the logistic map presented in Table III solutions for all the gates exist only in the limit of $a=4$ when the chaotic attractor covers the entire interval. The degree and form of nonlinearity necessary for obtaining all the desired logic responses will depend on the system at hand and on the specific scheme employed to obtain the desired input-output mapping. It may happen that certain nonlinear systems will allow a wide range of logic responses without actually being chaotic.

Proof-of-principle experiments with electronic circuits have rigorously verified the above concept.[4,5] Also note that the generalization of this concept, described explicitly for two-input gates here, is easily extended to multiple input gates.[14] These morphing multiple input gates would make *chaogate*-based computing machines more power efficient, and would also serve to increase their performance and to widen their range of applications.

Lastly contrast this to a conventional field programmable gate array element,[18] where reconfiguration is achieved through switching between multiple single purpose gates, rather than using a single element to perform many different logic functions. This latter is expected to be faster and less wasteful of space on an integrated circuit.

III. LOGIC FROM NONLINEAR EVOLUTION: DYNAMICAL LOGIC OUTPUTS

Now we describe a method for obtaining a logic output from a nonlinear system using the time evolution of the state of the system. This concept uses the *nonlinear characteristics* of the time dependence of the state of the dynamical system to elicit different responses from the system. The highlight of this method is that a single system can perform complicated logic operations very *efficiently*.

We extend the concept in Sec. II by allowing our system to evolve further in time. In doing so we might find that the system can perform an AND operation on its first iteration, but a NOR operation on the next iteration, etc. Thus we hope to perform a sequence of mathematical or logical operations almost as easily as a single operation.

As before encode the inputs via the initial state: $x\rightarrow x_0+X_1+X_2$, for two-input logic operations, with $X=0$ when logic input $I=0$ and $X=\delta$ (with $\delta>0$) when logic input $I=1$.

TABLE V. Updated state of chaotic logistic map satisfying the conditions in Table IV in order to implement the logical operations NAND, AND, NOR, XOR, and OR during different iterations with $x_0=0.325$, $\delta=\frac{1}{4}$, and $x^*=0.6$.

Operation	NAND	AND	NOR	XOR	OR
Iteration n	1	2	3	4	5
State of the system (x_n)	x_1	x_2	x_3	x_4	x_5
Logic input (0,0) $x_0=0.325$	0.88	0.43	0.98	0.08	0.28
Logic input (0,1)/(1,0) $x_0=0.575$	0.9775	0.088	0.33	0.872	0.45
Logic input (1,1) $x_0=0.825$	0.58	0.98	0.1	0.34	0.9

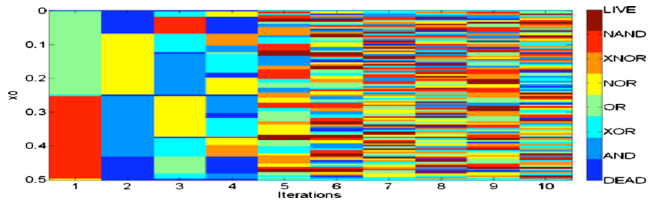

FIG. 1. (Color online) Template showing different logic patterns for the range of x_0 (0–0.5) vs iteration number n ($0<n\leq 10$). Here $x^*=0.75$ for $1\leq n\leq 4$ and $x^*=0.4$ for $n>4$. δ is fixed at 0.25 (Ref. 12).

After evolution over n time steps we determine the output using the condition: if $f_n(x)\leq x^*$ then logic output is 0, and if $f_n(x)>x^*$ then logic output is 1, where x^* is a reference threshold.

The principal difference from the earlier approach is the following: here we vary the time evolution n to obtain different logic relations: i.e., n is "programmed" to yield different logic. Unlike the earlier idea where n was held constant (at $n=1$ specifically) and x_{gate} was tuned to obtain different logic gate behavior, here we tap the state of the system at a longer time n to construct various logic combinations.

So the inputs set up the initial state of the nonlinear system: $x_0+I_1+I_2$. Then the system evolves over n iterative time steps to the final state x_n. The evolved state is compared to a monitoring threshold x^* If the state is greater than the threshold, a logical 1 is the output and, if the state is less than the threshold, a logical 0 is the output (Table IV). This process is repeated for subsequent iterations (see Fig. 1 and Table V for an example).

A. Implementation of half- and full-adder operations

Combinational logic can also be implemented directly by the above method. For instance, the ubiquitous bit-by-bit arithmetic addition (half-adder) involves two logic gate outputs: AND (to obtain the carry) and XOR (to obtain first digit of the sum). Using the scheme above we can obtain this combinational operation in consecutive iterations, with a *single* one-dimensional chaotic element.

Further, the typical full-adder requires two half-adder circuits and an extra OR gate. So in total, the implementation of a full-adder requires five different gates (two XOR gates, two AND gates, and one OR gate). However, using the dynamical evolution of a single logistic map, we require only three iterations to implement the full-adder circuit. So this method allows combinational logic to be obtained very efficiently with fewer computational modules and no cascading.

IV. ASYMMETRIC INPUTS

Now we describe an extension of the above formalism to include asymmetric logic operations. In this method the inputs I_1 and I_2 of a two-input logic operation are encoded as follows:

$$x_0 = x_{\text{gate}} + X_1 + X_2,$$

where x_{gate} is "programmable" to yield different logic function types.

With no loss of generality we follow the scheme:

- $X_1=0$ when $I_1=0$ and $X_1=\delta$ when $I_1=1$.
- $X_2=0$ when $I_2=0$ and $X_2=2\delta$ when $I_2=1$,

where δ is some positive constant.

Consider four distinct situations:

Case 1: Both I_1 and I_2 are 0 (row 1 in Table I), i.e., the initial state of the system is $x_0=x_{\text{gate}}+0+0=x_{\text{gate}}$.

Case 2: $I_1=1$, $I_2=0$ (row 2 in Table I), i.e., the initial state is $x_0=x_{\text{gate}}+\delta+0=x_{\text{gate}}+\delta$.

Case 3: $I_1=0$, $I_2=1$ (row 3 in Table I), i.e., the initial state is $x_0=x_{\text{gate}}+0+2\delta=x_{\text{gate}}+2\delta$.

Case 4: Both I_1 and I_2 are 1 (row 4 in Table I), i.e., the initial state is $x_0=x_{\text{gate}}+\delta+2\delta=x_{\text{gate}}+3\delta$.

This encoding of the inputs is capable of distinguishing between the input sets (0,1) and (1,0) and therefore can treat asymmetric logic functions correctly.

As before, the evolved state $x_1=f(x_0)$ yields the logic output as described before: logic output=0 if $f(x_0)\leq x^*$, logic output=1 if $f(x_0)>x^*$. Here the value of x_0 (equal to $x_{\text{gate}}+I_1+I_2$) is determined by the input set, and varying x_{gate} can select out different logic functions.

The basic logic gates, which are symmetric [with input set (0,1) being equivalent to (1,0)], can be obtained as a specific case of this general formalism. For instance, in order to obtain all the desired input-output responses of the gates displayed in Table I, we need to satisfy the conditions enumerated in Table VI simultaneously. That is, given a dynamics $f(x)$, one must find values of threshold x^*, constant δ, and initial state x_0 that satisfy the conditions derived from the truth table to be implemented.

TABLE VI. Necessary and sufficient conditions to be satisfied by a nonlinear system in order to implement the logical operations AND, OR, XOR, NAND, and NOR consistently on all possible input sets. Considering $a=4$ in the logistic map function, iteration $n=1$, threshold level $x^*=0.95$, and $\delta=0.1$, solutions of x_{gate} satisfying the above conditions simultaneously are 0.1, 0.3, 0.35, 0.4, and 0.6, respectively, for AND, OR, XOR, NAND, and NOR logic operations.

Inputs	AND	OR	XOR	NAND	NOR
(0,0)	$f(x_{\text{gate}})\leq x^*$	$f(x_{\text{gate}})\leq x^*$	$f(x_{\text{gate}})\leq x^*$	$f(x_{\text{gate}})>x^*$	$f(x_{\text{gate}})>x^*$
(0,1)	$f(x_{\text{gate}}+\delta)\leq x^*$	$f(x_{\text{gate}}+\delta)>x^*$	$f(x_{\text{gate}}+\delta)>x^*$	$f(x_{\text{gate}}+\delta)>x^*$	$f(x_{\text{gate}}+\delta)\leq x^*$
(1,0)	$f(x_{\text{gate}}+2\delta)\leq x^*$	$f(x_{\text{gate}}+2\delta)>x^*$	$f(x_{\text{gate}}+2\delta)>x^*$	$f(x_{\text{gate}}+2\delta)>x^*$	$f(x_{\text{gate}}+2\delta)\leq x^*$
(1,1)	$f(x_{\text{gate}}+3\delta)>x^*$	$f(x_{\text{gate}}+3\delta)>x^*$	$f(x_{\text{gate}}+3\delta)\leq x^*$	$f(x_{\text{gate}}+3\delta)\leq x^*$	$f(x_{\text{gate}}+3\delta)\leq x^*$

TABLE VII. Necessary and sufficient conditions to be satisfied by a nonlinear system in order to implement the logical operations AND, XOR, OR, NAND, XNOR, and NOR, with n being the number of evolution steps. For example, when $a=4$, $x^*=0.5$, and $\delta=1$, $x_0=0.005$, 0.05, 0.75, 0.3, 0.425, and 0.5 yield AND, XOR, OR, NAND, XNOR, and NOR logic operations, respectively.

Inputs	n	AND	XOR	OR	NAND	XNOR	NOR
(0,0)	1	$x_1=f(x_0)\le x^*$	$x_1=f(x_0)\le x^*$	$x_1=f(x_0)\le x^*$	$x_1=f(x_0)>x^*$	$x_1=f(x_0)>x^*$	$x_1=f(x_0)>x^*$
(0,1)	2	$x_2=f(x_1)\le x^*$	$x_2=f(x_1)>x^*$	$x_2=f(x_1)>x^*$	$x_2=f(x_1)>x^*$	$x_2=f(x_1)\le x^*$	$x_2=f(x_1)\le x^*$
(1,0)	3	$x_3=f(x_2)\le x^*$	$x_3=f(x_2)>x^*$	$x_3=f(x_2)>x^*$	$x_3=f(x_2)>x^*$	$x_3=f(x_2)\le x^*$	$x_3=f(x_2)\le x^*$
(1,1)	4	$x_4=f(x_3)>x^*$	$x_4=f(x_3)\le x^*$	$x_4=f(x_3)>x^*$	$x_4=f(x_3)\le x^*$	$x_4=f(x_3)>x^*$	$x_4=f(x_3)\le x^*$

Alternately one can use the iteration value n to encode the logic inputs,

$$n \rightarrow n + I_1 + I_2.$$

This again yields four distinct input conditions:

Case 1: Both I_1 and I_2 are 0 (row 1 in Table I), i.e., the evolution time is $n+0+0=n$.

Case 2: $I_1=1$, $I_2=0$ (row 2 in Table I), i.e., the evolution time is $n+\delta+0=n+\delta$.

Case 3: $I_1=0$, $I_2=1$ (row 3 in Table I), i.e., the evolution time is $n+0+2\delta=n+2\delta$.

Case 4: Both I_1 and I_2 are 1 (row 4 in Table I), i.e., the evolution time is $n+\delta+2\delta=n+3\delta$.

After chaotic updates over n steps, the state of the evolved system $f_n(x)$ yields the desired logic output.

In order to obtain all the desired input-output responses of all the basic logic gates displayed in Table I, we need to satisfy all the conditions enumerated in Table VII simultaneously.

As a case study, we will next consider the implementation of bit-by-bit addition and subtraction, thus demonstrating how complex combinational operations can be directly achieved by a single system, without necessitating concatenation. *Significantly, the half-subtractor is an asymmetric logic operation and so it cannot be obtained by the formalism in Sec. II.*

V. IMPLEMENTATION OF BIT-BY-BIT ARITHMETIC FUNCTIONS

We demonstrate here how one can obtain the very widely used bit-by-bit arithmetic function involving two logic gate outputs, in consecutive iterations, using a single one-dimensional chaotic element.

A. Half-adder operation

A half-adder operates on two inputs, I_1 and I_2, and yields two outputs, *carry* and *sum*. Carry is the AND logic output of I_1 and I_2 [i.e., $O_1=\mathrm{AND}(I_1,I_2)$], and sum is the XOR of I_1 and I_2 [i.e., $O_2=\mathrm{XOR}(I_1,I_2)$]. A summary of the necessary and sufficient conditions to be satisfied for the cascaded bit-by-bit addition operation is given in Table VIII. It is required to have all the necessary and sufficient conditions tabulated in Table VIII to be satisfied simultaneously to implement $O_1=\mathrm{AND}(I_1,I_2)$ and $O_2=\mathrm{XOR}(I_1,I_2)$, since the mapping from (I_1,I_2) to (O_1,O_2) must hold for all combinations of (I_1,I_2). Conversely, when these conditions are satisfied, $O_1=\mathrm{AND}(I_1,I_2)$ and $O_2=\mathrm{XOR}(I_1,I_2)$ hold. That is, these conditions are necessary and sufficient for the logic operations implementing half-adder arithmetic operation. For instance, for $x_0=0$, $x^*=0.65$, $\delta=0.1$, and $a=4.0$, outputs O_1 and O_2 are obtained in iterations $n=1$ and 2, respectively, for the logistic map.

B. Half-subtractor operation

A half-subtractor operates on two inputs I_1 and I_2 and yields two outputs, *borrow* and *difference*. Borrow is the result of the AND operation on $\mathrm{NOT}(I_1)$ and I_2 [i.e., $O_1=\mathrm{AND}(\mathrm{NOT}(I_1),I_2)$], and difference is the result of the XOR operation on I_1 and I_2 [i.e., $O_2=\mathrm{XOR}(I_1,I_2)$]. A summary of the necessary and sufficient conditions to be satisfied for the cascaded bit-by-bit subtraction operation is given in Table IX. It is required that all the necessary and sufficient

TABLE VIII. Truth table of the half-adder and the necessary and sufficient conditions that implement the half-adder.

Logic operation	Input set (I_1,I_2)	Logic output	Initial state	Necessary and sufficient condition
Carry	(0,0)	0	x_0	$x_1=f(x_0)\le x^*$
	(0,1)	0	$(x_0+\delta)$	$x_1=f(x_0+\delta)\le x^*$
	(1,0)	0	$(x_0+2\delta)$	$x_1=f(x_0+2\delta)\le x^*$
	(1,1)	1	$(x_0+3\delta)$	$x_1=f(x_0+3\delta)>x^*$
Sum	(0,0)	0	x_0	$x_2=f(x_0)\le x^*$
	(0,1)	1	$(x_0+\delta)$	$x_2=f(x_0+\delta)>x^*$
	(1,0)	1	$(x_0+2\delta)$	$x_2=f(x_0+2\delta)>x^*$
	(1,1)	0	$(x_0+3\delta)$	$x_2=f(x_0+3\delta)\le x^*$

TABLE IX. Truth table of half-subtractor and the necessary and sufficient conditions that implement the half-subtractor.

Logic operation	Input set (I_1,I_2)	Logic output	Initial state	Necessary and sufficient condition
Difference	(0,0)	0	$x_0=0.525$	$x_3=f(x_0)\le x^*$
	(0,1)	1	$(x_0+\delta)=0.625$	$x_3=f(x_0+\delta)>x^*$
	(1,0)	1	$(x_0+2\delta)=0.725$	$x_3=f(x_0+2\delta)>x^*$
	(1,1)	0	$(x_0+3\delta)=0.825$	$x_3=f(x_0+3\delta)\le x^*$
Borrow	(0,0)	0	$x_0=0.525$	$x_4=f(x_0)\le x^*$
	(0,1)	1	$(x_0+\delta)=0.625$	$x_4=f(x_0+\delta)>x^*$
	(1,0)	0	$(x_0+2\delta)=0.725$	$x_4=f(x_0+2\delta)\le x^*$
	(1,1)	0	$(x_0+3\delta)=0.825$	$x_4=f(x_0+3\delta)\le x^*$

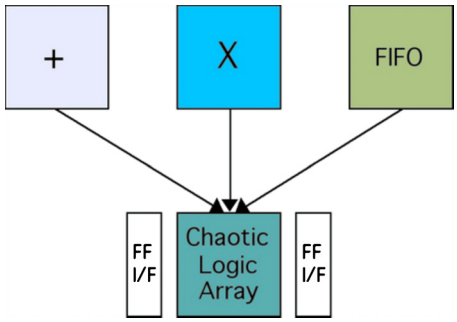

FIG. 2. (Color online) Simplified schematic of the proof of concept VLSI implementation of an ALU which can switch between at least two arithmetic functions and a completely different function such as a small first-in, first-out memory buffer.

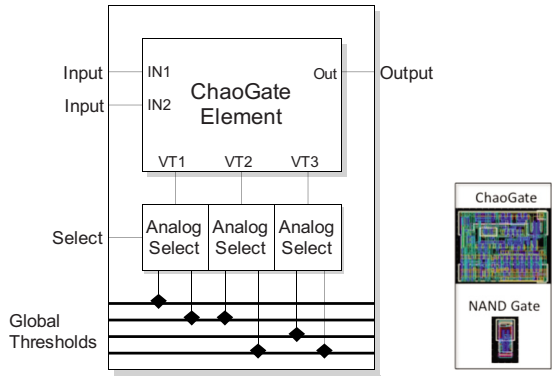

FIG. 3. (Color online) (Left) Schematic of a two-input, one output morphable chaogate. The gate logic functionality (NOR, NAND, and XOR) is controlled (morphed) in the current VLSI design by global thresholds connected to VT1, VT2, and VT3 through analog multiplexing circuitry and (right) a size comparison between the current chaogate circuitry implemented in the ChaoLogix VLSI chaotic computing chip and a typical NAND gate circuit (courtesy of ChaoLogix, Inc.).

conditions tabulated in Table IX be satisfied simultaneously to implement $O_1=\mathrm{AND}(\mathrm{NOT}(I_1),I_2)$ and $O_2=\mathrm{XOR}(I_1,I_2)$, since the mapping from (I_1,I_2) to (O_1,O_2) must hold for all combinations of (I_1,I_2). Conversely, when these conditions are satisfied, $O_1=\mathrm{AND}(\mathrm{NOT}(I_1),I_2)$ and $O_2=\mathrm{XOR}(I_1,I_2)$ hold. That is, these conditions are necessary and sufficient for the logic operations implementing the half-subtractor arithmetic operation. For instance, with $x_0=0.525$, $x^*=0.65$, $\delta=0.1$, and $a=4.0$, outputs O_2 and O_1 are obtained in iterations $n=3$ and $n=4$, respectively, for the logistic map.

So the highlight of this method is the ability of a single nonlinear system to yield complicated logic outputs. The basis of this ability is the range of states a nonlinear system accesses in the course of its evolution. This allows the evolution time to be used as a "knob" to extract different combinational logic responses, without necessitating concatenation of many units.

Thus we have presented a design for a computing device based on the capability of a nonlinear system to implement all the fundamental computing operations. It does this by exploiting the nonlinear responses of the system. The main benefit is its ability to reconfigure a single chaotic element into different logic gates. Contrast this with a conventional field programmable gate array element, where reconfiguration is achieved through switching between multiple single purpose gates. This latter sort of reconfiguration is both slow and wasteful of space on an integrated circuit.

VI. VLSI IMPLEMENTATION OF CHAOTIC COMPUTING ARCHITECTURES—PROOF OF CONCEPT

Recently ChaoLogix, Inc. designed and fabricated a proof of concept chip that demonstrates the feasibility of constructing reconfigurable chaotic logic gates, dubbed chaogates, in standard CMOS-based VLSI (0.18 μm process operating at 30 MHz with a 3.1×3.1 mm die size and a 1.8 V digital core). The basic building block chaogate is shown schematically in Fig. 2.

The demonstration chip (Fig. 3) has a parallel read/write interface to communicate with a microcontroller using standard logic gates. The read/write interface responds to a range of addresses to give access to internal registers, and the internal registers will interface to the demonstration chaotic computing circuits.

Chaogates were then incorporated into a chaogate array in the VLSI chip to demonstrate higher order morphing functionality including the following:

(1) A small ALU that morphs between higher order arithmetic functions (multiplier and adder/accumulator) in *less than one clock cycle*. An ALU is a basic building block of computer architectures.
(2) A communications protocols unit that morphs between two different complex communications protocols in less than one clock cycle: serial peripheral interface (a synchronous serial data link) and an inter-integrated circuit control bus implementation (I2C, a multimaster serial computer bus).

While the design of the chaogates and chaogate arrays in this proof of concept VLSI chip was not optimized for performance, it clearly demonstrates that chaogates can be constructed and organized into reconfigurable chaotic logic gate arrays capable of morphing between higher order computational building blocks. Current efforts are focused upon optimizing the design of a single chaogate to levels where it is comparable to or smaller than a single NAND gate in terms of power and size, yet is capable of morphing between all gate functions in less than a single computer clock cycle. Preliminary designs indicate that this goal is achievable and that all gates currently used to design computers may be replaced with chaogates to provide added flexibility and performance. Practical applications may also take a hybrid approach, combining static hardware with modules of programmable hardware.

One should also add the following caveat: programming chaogates requires the development of new hardware description languages. The absence of these may pose a bottleneck in obtaining the most efficient use of the chaogate. It is conceivable that ideas from evolutionary algorithms may be employed to reach optimal configurations of arrays of chaogates. At this point, this is still a completely open problem.

VII. CONCLUSIONS

In summary, we have demonstrated the direct and flexible implementation of all the basic logic gates utilizing nonlinear dynamics. The richness of the dynamics allows us to select out all the different gate responses from the same processor by simply setting suitable threshold levels. These threshold levels are known exactly from theory and are thus available as a look-up table. Arrays of such logic gates can conceivably be programmed on the run (for instance, with a stream of threshold values being sent in by an external program) to be optimized for the task at hand. For instance, such a morphing device may serve flexibly as an arithmetic processing unit or a unit of memory and can be swapped as the need demands to be one or the other. This capacity for reconfigurability may then enable us to achieve the flexibility of field programmable gate arrays (FPGAs),[18] the optimization and speed of application specific integrated circuits (ASIC),[22] and the general utility of a central processing unit within the same computer chip architecture.

Finally, regarding the applicability of this idea, nonlinear systems are abundant in nature, and so embodiments of this concept are conceivable in many different physical systems, ranging from fluids to electronics to optics. Thus chaos-based computing may be implemented not only on conventional CMOS-based VLSI circuitry (as discussed in Sec. VI) but also on more esoteric platforms. Possible candidates for physical realization of the method include nonlinear electronic circuits,[4] magneto-based circuitry,[23] high speed chaotic photonic integrated circuits operating in the gigahertz frequency range,[24] and single electron tunneling junctions[25] which are natural piecewise linear maps. Therefore the idea of exploiting nonlinear dynamics for computation has considerable potential to be realized in a variety of physical systems, offering an alternate dynamics-based computing device with capacity for reconfigurability.

[1] S. Sinha and W. L. Ditto, Phys. Rev. Lett. **81**, 2156 (1998).

[2] S. Sinha, T. Munakata, and W. L. Ditto, Phys. Rev. E **65**, 036214 (2002); T. Munakata, S. Sinha, and W. L. Ditto, IEEE Trans. Circuits Syst., I: Fundam. Theory Appl. **49**, 1629 (2002); T. Munakata and S. Sinha, Proceedings of COOL Chips VI, Yokohama, 2003, p. 73.

[3] S. Sinha and W. L. Ditto, Phys. Rev. E **60**, 363 (1999); S. Sinha, T. Munakata, and W. L. Ditto, *ibid.* **65**, 036216 (2002); W. L. Ditto, K. Murali, and S. Sinha, Proceedings of IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS06), Singapore, 2006, pp. 1835–1838.

[4] K. Murali, S. Sinha, and W. L. Ditto, Proceedings of the STATPHYS-22 Satellite Conference Perspectives in Nonlinear Dynamics, Special Issue of Pramana, 2005, Vol. 64, p. 433.

[5] K. Murali, S. Sinha, and W. L. Ditto, Int. J. Bifurcation Chaos Appl. Sci. Eng. **13**, 2669 (2003); K. Murali, S. Sinha, and I. R. Mohamed, Phys. Lett. A **339**, 39 (2005).

[6] K. Murali, S. Sinha, and W. L. Ditto, Proceedings of Experimental Chaos Conference (ECC9), Brazil, 2006 [Philos. Trans. R. Soc. London, Ser. A **366**, 653 (2008)]; K. Murali, S. Sinha, and W. L. Ditto, Proceedings of IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS06), Singapore, 2006, pp. 1839–1842.

[7] B. S. Prusha and J. F. Lindner, Phys. Lett. A **263**, 105 (1999).

[8] D. Cafagna and G. Grassi, International Symposium on Signals, Circuits and Systems (ISSCS 2005), 2005, Vol. 2, p. 749.

[9] K. E. Chlouverakis and M. J. Adams, Electron. Lett. **41**, 359 (2005).

[10] M. R. Jahed-Motlagh, B. Kia, W. L. Ditto, and S. Sinha, Int. J. Bifurcation Chaos Appl. Sci. Eng. **17**, 1955 (2007).

[11] K. Murali and S. Sinha, Phys. Rev. E **75**, 025201 (2007).

[12] A. Miliotis, K. Murali, S. Sinha, W. L. Ditto, and M. L. Spano, Chaos, Solitons Fractals **42**, 809 (2009).

[13] A. Miliotis, S. Sinha, and W. L. Ditto, Int. J. Bifurcation Chaos Appl. Sci. Eng. **18**, 1551 (2008); A. Miliotis, S. Sinha, and W. L. Ditto, Proceedings of IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS06), Singapore, 2006, pp. 1843–1846.

[14] W. L. Ditto, K. Murali, and S. Sinha, *Understanding Complex Systems* (Springer, Berlin, 2009), pp. 3–13.

[15] K. Murali, A. Miliotis, W. L. Ditto, and S. Sinha, Phys. Lett. A **373**, 1346 (2009).

[16] K. Murali, S. Sinha, W. L. Ditto, and A. R. Bulsara, Phys. Rev. Lett. **102**, 104101 (2009); S. Sinha, J. M. Cruz, T. Buhse, and P. Parmananda, Europhys. Lett. **86**, 60003 (2009); K. Murali, I. R. Mohamed, S. Sinha, W. L. Ditto, and A. R. Bulsara, Appl. Phys. Lett. **95**, 194102 (2009); D. N. Guerra, A. R. Bulsara, W. L. Ditto, S. Sinha, K. Murali, and P. Mohanty, Nano Lett. **10**, 1168 (2010).

[17] W. Ditto, S. Sinha, and K. Murali, U.S. Patent No. 07,096,347 (22 August 2006); Commercialization of this technology is undertaken by the company ChaoLogix, http://www.chaologix.com.

[18] FPGAs allow the same hardware to be used in many different applications using programmable interconnects which enable the logic gates to be wired differently. See G. Taubes, Science **277**, 1931 (1997).

[19] M. M. Mano, *Computer System Architecture*, 3rd ed. (Prentice-Hall, Englewood Cliffs, NJ, 1993); T. C. Bartee, *Computer Architecture and Logic Design* (McGraw-Hill, New York, 1991).

[20] S. Sinha and D. Biswas, Phys. Rev. Lett. **71**, 2010 (1993); L. Glass and W. Zheng, Int. J. Bifurcation Chaos Appl. Sci. Eng. **4**, 1061 (1994); S. Sinha, Phys. Rev. E **49**, 4832 (1994); Phys. Lett. A **199**, 365 (1995); S. Sinha and W. L. Ditto, Phys. Rev. E **63**, 056209 (2001); S. Sinha, *ibid.* **63**, 036212 (2001); S. Sinha, in *Nonlinear Systems*, edited by R. Sahadevan and M. L. Lakshmanan (Narosa, New Delhi, 2002), pp. 309–328; W. L. Ditto and S. Sinha, Philos. Trans. R. Soc. London, Ser. A **364**, 2483 (2006).

[21] K. Murali and S. Sinha, Phys. Rev. E **68**, 016210 (2003).

[22] ASICs are customized for a particular use rather than intended for general-purpose use. For example, it may be designed to solely run a cell phone. ASICs may involve a large number of logic gates (typically between thousands to millions) and are optimized for specific functions alone.

[23] R. Koch, Sci. Am. **293**, 56 (2005).

[24] M. Yousefi, Y. Barbarin, S. Beri, E. A. J. M. Bente, M. K. Smit, R. Notzel, and D. Lenstra, Phys. Rev. Lett. **98**, 044101 (2007).

[25] T. Yang and L. O. Chua, Int. J. Bifurcation Chaos Appl. Sci. Eng. **10**, 1091 (2000).