Physics Letters A 373 (2009) 1346–1351

Contents lists available at ScienceDirect

Physics Letters A

www.elsevier.com/locate/pla

Logic from nonlinear dynamical evolution

K. Murali [a,b,*], Abraham Miliotis [a], William L. Ditto [a], Sudeshna Sinha [c]

[a] J. Crayton Pruitt Family Department of Biomedical Engineering, University of Florida, Gainesville, FL 32611-6131, USA
[b] Department of Physics, Anna University, Chennai 600 025, India
[c] Institute of Mathematical Sciences, C.I.T. Campus, Chennai 600 113, India

ARTICLE INFO

Article history:
Received 12 November 2008
Received in revised form 3 February 2009
Accepted 8 February 2009
Available online 14 February 2009
Communicated by C.R. Doering

PACS:
05.45.-a
89.70.+c

Keywords:
Chaotic systems
Digital logic
Computing with chaos

ABSTRACT

We propose a direct and flexible implementation of logic operations using the dynamical evolution of a nonlinear system. The concept involves the observation of the state of the system at different times to obtain different logic outputs. We explicitly implement the basic NAND, AND, NOR, OR and XOR logic gates, as well as multiple-input XOR and XNOR logic gates. Further we demonstrate how the single dynamical system can do more complex operations such as bit-by-bit addition in just a few iterations.

1. Introduction

Recently there has been a new theoretical direction in harnessing the richness of nonlinear dynamics, namely the exploitation of chaos to do flexible computations. This so-called *chaos computing* paradigm [1–8] is driven by the motivation to use new concepts of physics to build better computing devices. The general strategy underlying this research activity exploits the determinism of dynamics on one hand, and its richness on the other. The determinism allows one to reverse engineer, so to speak and the richness of dynamical patterns allows flexibility and versatility in accomplishing wide-ranging operations. This novel paradigm forms part of the over-arching attempt to find new ways to exploit physical phenomena that are well understood in the context of physics, to do computations, and in particular to bridge dynamical phenomena and computations [9–13].

The fundamental components of computer architecture today are the logical AND, OR, NOT, and XOR operations, from which we can directly obtain basic operations like bit-by-bit addition and memory [14]. A typical 2-input operation act on two inputs I_1 and I_2 and outputs a signal O. The type of logic is defined by patterns of input-to-output mapping represented by the truth table in Table 1. Now all the above mentioned gates can be constructed by combining the NOR or NAND operations [14]. Clearly though, this conversion process is inefficient in comparison with direct implementation, considering perhaps that such fundamental operations may be performed a large number of times. So the direct and flexible implementation of gates is useful and could prove very cost effective. Our problem then is to design a method that yields the appropriate outputs, for the different fundamental gates, for all possible sets of inputs.

Table 1
The truth table of the basic logic operations.

Input	NAND	AND	NOR	XOR	OR
(0, 0)	1	0	1	0	0
(0, 1)	1	0	0	1	1
(1, 0)	1	0	0	1	1
(1, 1)	0	1	0	0	1

Towards this aim, in this work we first show the direct and flexible implementation of all these logical operations utilizing low dimensional chaos. Importantly the motivation is to use a single nonlinear element to emulate different logic gates and perform different arithmetic tasks, and further have the ability to switch easily between the different operational roles. Such a reconfigurable logic unit may then serve as an ingredient for the construction of general purpose reprogrammable hardware.

Arrays of such morphing logic gates can conceivably be programmed on the run (for instance, by an external program) to be

* Corresponding author at: Department of Physics, Anna University, Chennai 600 025, India. Tel.: +1 352 3928934; fax: +1 352 392 9791.
E-mail addresses: kmurali@annauniv.edu, muralikin@ufl.edu (K. Murali).

0375-9601/$ – see front matter
doi:10.1016/j.physleta.2009.02.026

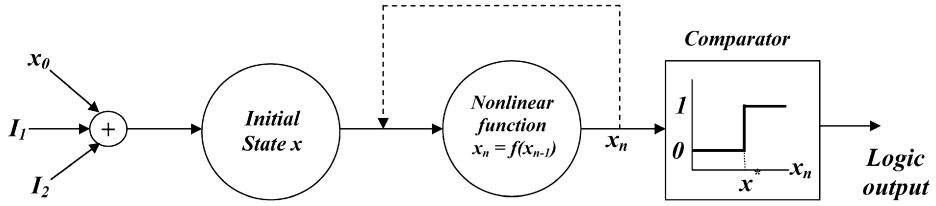

Fig. 1. Schematic diagram of the nonlinear evolution based logic operations. The dotted line denotes successive iteration operation with updated value x_n for $n > 1$. The logic output is recovered from x_n using a comparator with reference threshold value x^*.

optimized for the task at hand. For instance, they may serve flexibly as an arithmetic processing unit or an unit of memory, and can be swapped, as the need demands, to be one or the other. Applications of such reconfigurable hardware includes digital signal processing, software-defined radio, aerospace and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation and a growing range of other areas. Further advantages of reconfigurable hardware include the ability to re-program in the field, to fix bugs, lower non-recurring engineering costs and implement coarse-grained architectural approaches [15].

In the present work, we suggest a method for obtaining logic output from a nonlinear system using the time evolution of the state of the system. Namely, our concept uses the nonlinear characteristics of the time dependence of the state of the dynamical system to extract different responses from the system at different time instances. The highlight of this method is that a *single* nonlinear system is capable of yielding *a time sequence* of logic operations. We explicitly demonstrate the implementation of sequences of fundamental logic gates, as well as the direct implementation of bit-by-bit addition through such a sequence.

The organization of this Letter is as follows: In Section 2, we demonstrate the basic principles of obtaining sequences of different logic operations using the evolution of nonlinear iterative maps. Section 3 presents a specific realization of arithmetic operations using an one-dimensional chaotic map and a sequence of logic operations. Section 4 presents the extension of this approach for implementing multi-input logic gate operations. Finally, Section 5 discusses and summarizes the results.

2. Generation of a sequence of logic operations using iterates of a chaotic map

Now we outline a method for obtaining all basic logic gates using different dynamical iterates of a single nonlinear system. In particular consider a chaotic system whose state is represented by a value x. The state of the system evolves according to some dynamical rule. For instance, the updates of the state of the element from time n to $n+1$ may be well described by a map, i.e., $x_{n+1} = f(x_n)$, where f is a nonlinear function.

Now this element receives inputs before the first iteration (i.e., $n = 0$) and outputs a signal after evolving for a (short) specified time or number of iterations.

In our technique all the basic logic gate operations, NAND, AND, NOR, XOR and OR (see Table 1 for the truth table), involve the following steps:

(1) Inputs (for a 2 inputs operation):

$$x \to x_0 + I_1 + I_2$$

where x_0 is the initial state of the system, and $I = 0$ when logic input is zero, and $I = \delta$ (where δ is some positive constant) when logic input is one.

So we need to consider the following three situations:

Case 1. Both I_1, and I_2 are 0 (row 1 in Table 1) i.e., the initial state of the system is $x_0 + 0 + 0 = x_0$.

Case 2. Either $I_1 = 1, I_2 = 0$ or $I_1 = 0, I_2 = 1$ (row 2 or 3 in Table 1) i.e., the initial state is $x_0 + 0 + \delta = x_0 + \delta + 0 = x_0 + \delta$.

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

(2) Chaotic updates over some prescribed number of steps, i.e., $x \to f_n(x)$, where $f_n(x)$ is the nth iterate of the evolution of the function $f(x)$.

(3) The evolved state $f_n(x)$ yields a logic output at iteration n as follows:

$$\text{Logic output} = 0 \quad \text{if } f(x) \leqslant x^*,$$

$$\text{Logic output} = 1 \quad \text{if } f(x) > x^*,$$

where x^* is a reference threshold value.

Since the system is chaotic, in order to specify the initial x_0 accurately one needs a controlling mechanism. For instance one can employ a threshold controller to set the initial value x_0. The action of threshold control or limit control is to clip the state of a system to some prescribed value. The theory and experimental verification of this efficient control method is given in [16,17]. Note that the state of the system can be reset to x_0 at any time using such a controller, and after that the system can be 're-used' as another logic gate, as the situation demands. For logic recovery, the updated or evolved value of $f(x)$ is compared with x^* value using a comparator action as shown in Fig. 1. This recovered output can be properly rescaled to match with input logic levels in-terms of δ, so that further concatenating these logic gates is possible.[1]

In order to obtain all the desired input-output responses of the different gates, as displayed in Table 1, we need to satisfy the conditions enumerated in Table 2 simultaneously. Note that the symmetry of inputs reduces the four conditions in the truth Table 1 to three distinct conditions, with rows 2 and 3 of Table 1 leading to condition 2 in Table 2.

So given dynamics $f(x)$, one must find values of a reference threshold x^* and initial state x_0 satisfying the conditions derived from the truth table to be implemented. Table 2 shows the exact values of the initial x_0 and reference threshold x^* when

$$f(x) = 4x(1-x). \tag{1}$$

Here $x \in [0, 1]$. The constant δ, common to all logical gates, is fixed as 0.25. The above inequalities have many possible solutions based on the size of δ. For example, by setting $\delta = 0.25$, we can simulate the equation for the different time shifts that each gate requires.

[1] The ease of concatenation of logic gates morphing through time evolution is still an open issue, and further engineering oriented, design specific, work is necessary in order to demonstrate how small and large groups of such logic gates can be integrated into computer architectures.

Thus the inputs setup the initial state $x_0 + I_1 + I_2$. Then the system evolves over n iterative time steps to an updated state x_n. The evolved state is compared to a monitoring threshold x^* (refer to Fig. 1), at every n. If the state at iteration n, is greater than x^* a logical 1 is the output and if the state is less than or equal to x^* a logical 0 is the output. This process is repeated for subsequent iterations. Relating inputs with the obtained outputs provides us the operation that is performed at a specific iteration. For illustrative purposes, the graphical iteration representation of Eq. (1) for various initial values corresponding to different logic inputs is depicted in Fig. 2. The initial values are denoted by labels a, b and c. For clarity, the state of x_n for first 5 iterations ($0 < n < 5$) can be identified from this diagram. It is interesting to note that first 5 iterations satisfy the realization of basic logic gates as indicated in Table 1. In addition, subsequent iterations beyond $n > 5$ continue to yield different logic gate operations including XNOR operation. A more exclusive template of various logic responses being admitted by this system (Eq. (1)) for different iterations n versus range of x_0 values is depicted in Fig. 3. To generate this template, the representative value of δ is fixed as 0.25. The value $x^* = 0.75$ is used for $1 \leqslant n \leqslant 4$ and $x^* = 0.4$ is used for $n > 4$. Fig. 3 shows the logic behavior arising from a system with initial state x_0 evolving over n iterative steps, with $n = 1, 2, \ldots, 10$. It is clear from this figure, that while the system will always yield some logic behavior, the robustness of the response, with respect to initial state specification is lost after n around 5 or so. This is expected from the chaotic nature of the dynamics, and so for large n the response is extremely sensitive to the precision with which x_0 is set. However note that one need not go to iterates beyond 5 or so, as all basic logic outputs can be obtained within the first few iterates, in large robust ranges of initial state x_0. After n around 5 or so, the system can be re-set, for instance by the threshold controller mentioned earlier, and the nonlinear system can be '*re-used*' after this re-initialization.

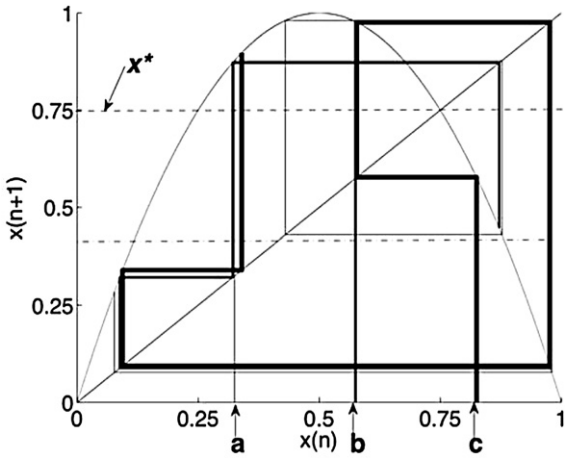

Fig. 2. Graphical iteration representation of the logistic map with three logic initial inputs (**a**) $= x_0$, (**b**) $= x_0 + \delta$ and (**c**) $= x_0 + 2\delta$ corresponding to Table 2. Here $x^* = 0.75$ is used to recover logic operations NAND, AND, NOR and XOR. For OR logic operation $x^* = 0.4$ is utilized.

3. Implementation of bit-by-bit addition

We now demonstrate how one can obtain the ubiquitous *bit-by-bit arithmetic addition*, involving two logic gate outputs, in con-

Table 2
Necessary and sufficient conditions to be satisfied by a chaotic element in order to implement the logical operations NAND, AND, NOR, XOR and OR during different iterations. Here $x_0 = 0.325$ and $\delta = 0.25$. $x^* = 0.75$ is used for NAND, AND, NOR, XOR logic operations and $x^* = 0.4$ is fixed for OR logic operation.

LOGIC	NAND	AND	NOR	XOR	OR
Iteration 'n'	1	2	3	4	5
Condition 1: Logic input $(0, 0)$ $x_0 = 0.325$	$x_1 = f(x_0) > x*$ $x_1 = 0.88$	$f(x_1) < x*$ $x_2 = 0.43$	$f(x_2) > x*$ $x_3 = 0.98$	$f(x_3) < x*$ $x_4 = 0.08$	$f(x_4) < x*$ $x_5 = 0.28$
Condition 2: Logic input $(0, 1)$ or $(1, 0)$ $x_0 = 0.575$	$x_1 = f(x_0 + \delta) > x*$ $x_1 = 0.9775$	$f(x_1) < x*$ $x_2 = 0.088$	$f(x_2) < x*$ $x_3 = 0.33$	$f(x_3) > x*$ $x_4 = 0.872$	$f(x_4) > x*$ $x_5 = 0.45$
Condition 3: Logic input $(1, 1)$ $x_0 = 0.825$	$x_1 = f(x_0 + 2\delta) < x*$ $x_1 = 0.58$	$f(x_1) > x*$ $x_2 = 0.98$	$f(x_2) < x*$ $x_3 = 0.1$	$f(x_3) < x*$ $x_4 = 0.34$	$f(x_4) > x*$ $x_5 = 0.9$

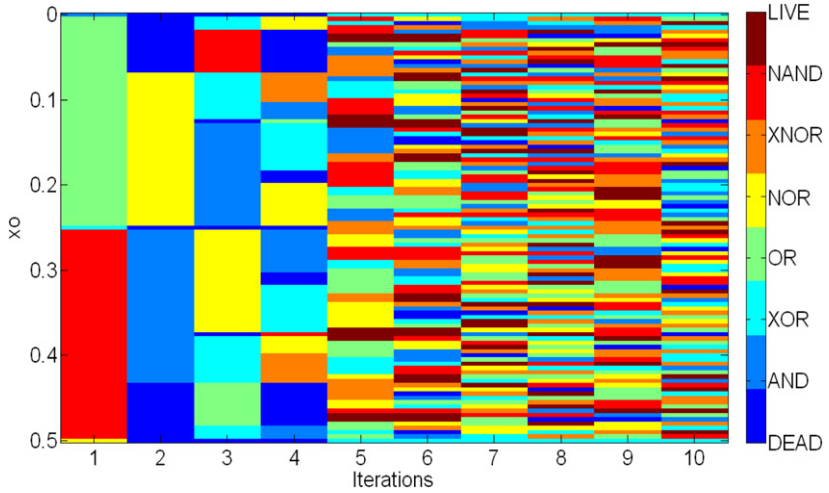

Fig. 3. Template showing different logic patterns for range of x_0 (0 to 0.5) versus iteration n (0 to 10). Here $x^* = 0.75$ for $1 \leqslant n \leqslant 4$ and $x^* = 0.4$ for $n > 4$. δ is fixed as 0.25.

Table 3
The truth table of full adder, necessary and sufficient conditions to be satisfied by the logistic map. State values x_1 (iteration $n = 1$) and x_2 (iteration $n = 2$) are used to obtain C_{out} and S, respectively. Here $x_1^* = 0.8, x_2^* = 0.4, x_0 = 0.0$ and $\delta \approx 0.23$.

Input bit for number (A)	Input bit for number (B)	Carry bit input (C_{in})	C_{out}	S	C_{out}	S
0	0	0	0	0	$x_1 = f(x_0) \leqslant x_1^*$	$x_2 = f(x_1) \leqslant x_2^*$
0	0	1	0	1	$x_1 = f(x_0 + \delta) \leqslant x_1^*$	$x_2 = f(x_1) > x_2^*$
0	1	0	0	1	$x_1 = f(x_0 + \delta) \leqslant x_1^*$	$x_2 = f(x_1) > x_2^*$
0	1	1	1	0	$x_1 = f(x_0 + 2\delta) > x_1^*$	$x_2 = f(x_1) \leqslant x_2^*$
1	0	0	0	1	$x_1 = f(x_0 + \delta) \leqslant x_1^*$	$x_2 = f(x_1) > x_2^*$
1	0	1	1	0	$x_1 = f(x_0 + 2\delta) > x_1^*$	$x_2 = f(x_1) \leqslant x_2^*$
1	1	0	1	0	$x_1 = f(x_0 + 2\delta) > x_1^*$	$x_2 = f(x_1) \leqslant x_2^*$
1	1	1	1	1	$x_1 = f(x_0 + 3\delta) > x_1^*$	$x_2 = f(x_1) > x_2^*$

secutive iterations, with a single one-dimensional chaotic element as obeying Eq. (1). A simple 1-bit binary arithmetic addition requires a full adder logic which adds three individual bits together (two bits being the digit inputs and the third bit assumed to be carry from the addition of the next least-significant bit addition operation, known as 'C_{in}'). A typical full-adder requires two half-adder circuits and an extra OR gate. In total, the implementation of a full-adder requires five different gates (two XOR gates, two AND gates and one OR gate) [14]. However in the present direct implementation by *utilizing the dynamical evolution of a single logistic map, we need only two iterations of a single element to implement a full-adder*.

Now by choosing the $\delta = 0.23$ and $x_0 = 0.0$, the truth table, summary of the necessary and sufficient conditions to be satisfied for the full-adder operation is given in Table 3. The *Carry bit* output C_{out} and the *Sumbit* output S are recovered from first and second iterations of map Eq. (1), respectively. Here thresholds x_1^* and x_2^* for 1st and 2nd iterations are fixed as 0.8 and 0.4, respectively. If $x_1 < x_1^*$ then C_{out} is logic zero or else it is logic one. Also if $x_2 < x_2^*$ then S is logic zero or else it is logic one. Now we will employ three steps to implement the full-adder logical operations:

Step 1. Initialization of the state of the system to x_0 and addition of external inputs,

$$x \to x_0 + I_1 + I_2 + I_3$$

where x_0 is the initial state of the system, and $I = 0$ when logic input is zero, and $I = \delta$ (where δ is some positive constant) when logic input is one. Here I_1, I_2 and I_3 correspond the input number A, input number B and carry input C_{in} respectively of Table 3. So we need to consider the following four situations:

Case 1. If all inputs are 0 (row 1 in Table 3) i.e., the initial state of the system is

$$x_0 + 0 + 0 + 0 = x_0.$$

Case 2. If any one of the input equals 1 (row 2, 3 and 5 in Table 3) i.e., the initial state is

$$x_0 + 0 + 0 + \delta = x_0 + 0 + \delta + 0 = x_0 + \delta + 0 + 0 = x_0 + \delta.$$

Case 3. If any two inputs equal to 1 (row 4, 6 and 7 in Table 3), i.e., the initial state is

$$x_0 + 0 + \delta + \delta = x_0 + \delta + 0 + \delta = x_0 + \delta + \delta + 0 = x_0 + 2\delta.$$

Case 4. If all inputs equal to 1 (row 8 in Table 3), i.e., the initial state is

$$x_0 + \delta + \delta + \delta = x_0 + 3\delta.$$

Table 4
The truth table of the 3-input XOR and XNOR logic operations, necessary and sufficient conditions to be satisfied by the map. State value x_2 (iteration $n = 2$) is used for logic operation recovery. Here $x^* = 0.5$ and $\delta \approx 0.25$.

I_1	I_2	I_3	XOR	XNOR	XOR ($x_0 = 0$)	XNOR ($x_0 = 0.25$)
0	0	0	0	1	$x_2 \leqslant x^*$	$x_2 > x^*$
0	0	1	1	0	$x_2 > x^*$	$x_2 \leqslant x^*$
0	1	0	1	0	$x_2 > x^*$	$x_2 \leqslant x^*$
1	0	0	1	0	$x_2 > x^*$	$x_2 \leqslant x^*$
0	1	1	0	1	$x_2 \leqslant x^*$	$x_2 > x^*$
1	0	1	0	1	$x_2 \leqslant x^*$	$x_2 > x^*$
1	1	0	0	1	$x_2 \leqslant x^*$	$x_2 > x^*$
1	1	1	1	0	$x_2 > x^*$	$x_2 \leqslant x^*$

Step 2. Chaotic evolution for two time steps, of the initial state given above, via Eq. (1).

Step 3. The evolved state $f_n(x)$ yields the logic output as follows:

$$\textit{Logic output} = 0 \quad \text{if } f_n(x) \leqslant x_n^*,$$

$$\textit{Logic output} = 1 \quad \text{if } f_n(x) > x_n^*,$$

where x_n^* is a monitoring threshold, with $n = 1, 2$.

In this representative example of implementing full adder operation, applying step 3 to the first two iterative values ($n = 1$ and 2) of Eq. (1), i.e., $f(x_0) = x_1$ and $f(x_1) = x_2$ yield the two outputs encoding C_{out} and S in Table 3. So basically an element will take the three inputs A, B and C_{in} and produce the carry for the next addition on the very first update. This new carry can of course be immediately supplied to the next element ready to perform the addition of the next bits, while the current sum (S) is calculated on the second update. However, we wish to emphasize that the time delay of occurrence between C_{out} and S (through iteration delay) can be compensated by adapting an interface circuitry(in actual hardware implementation) like sample-and-hold circuits with suitable sampling pulses [18]. Representative timing waveforms for the full-adder implementation are depicted in Fig. 4.

As a final note, consider that we can allow the map to evolve beyond the second iteration ($n > 2$) just as we carried-out for two inputs case referred in Section 2 and obtain different logical operations.

4. Implementation of multi-input logic gates

As in Section 2, consider a single chaotic element to be the logistic map model described by Eq. (1). Now this basic element can be further used to do specific logical operations with three or more logical inputs. The basic modification simply involves adding another input to the conventional 2-input logic gate structure. Three or more input logic gates are advantageous

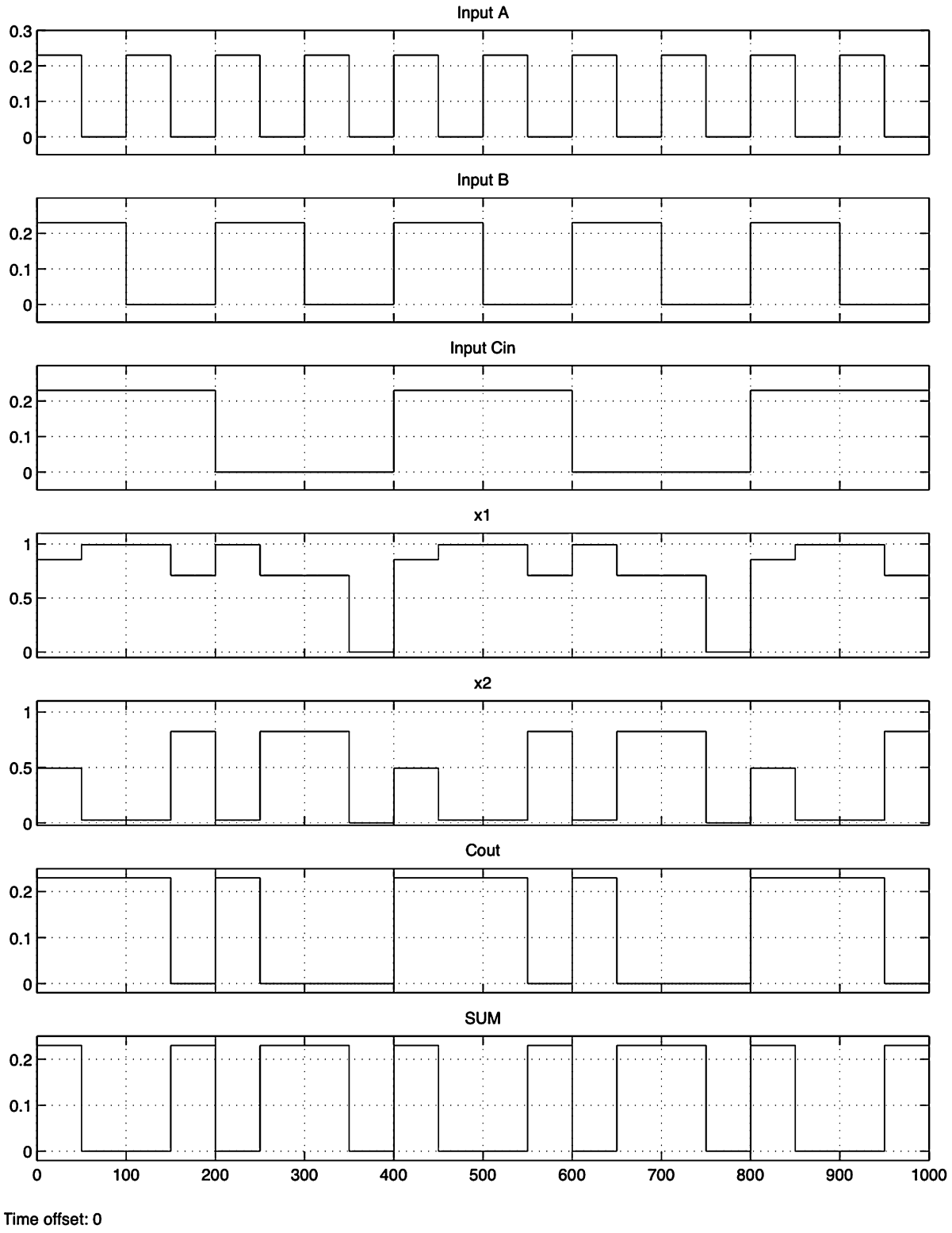

Fig. 4. Timing sequences for full-adder: First input A (panel 1), second input B (panel 2), third input C_{in} (panel 3), first iteration output x_1 (panel 4), second iteration output x_2 (panel 5), carry-out (panel 6) and sum (panel 7). Abscissa corresponds to time increment of each initialization.

because they require less complexity in actual experimental circuit realization than that of coupling conventional 2-input logic gates [14].

We consider the weights (δ) given to each logic input to be the same for the 2-input and 3-input gates, but the threshold value x^* will be different. In a manner exactly like the 2-input gates above, appropriate choices of x_0 and x^* can be found that lead to the realization of the 3-input XOR and XNOR logic operations. The truth table for 3-input XOR and XNOR logic gate operations, the necessary and sufficient conditions to be satisfied by the map is shown in Table 4. In this representative case, the state value x_2 (i.e., at iteration $n = 2$) of the logistic map is used uniformly for logic recovery. The threshold value x^* and δ are fixed as 0.5 and 0.25, respectively. For morphing between XOR and XNOR logic

operations, the initial values are fixed as $x_0 = 0$ and $x_0 = 0.25$, respectively.

5. Discussion

Previous results in chaos computing have shown that a single nonlinear dynamical system can (with proper tuning of parameters and control inputs) become any logic gate. Additionally it has been shown that such nonlinear dynamical systems can be morphed to become any logic gate [1–3]. Our discovery in this work lies in the remarkable result that the varied temporal patterns embedded in the dynamical evolution of nonlinear systems are capable of performing *sequences* of logic operations in time (or iterates) and in contrast with previous methods. Thus minimal control is needed thereby we only invoke control mechanism on initialization, from there on we just monitor the state and the morphing between gates takes place in time evolution, instead of varying the control parameters. So one can set a global parameter and let time evolve the logic, rather than micromanage each morphing step through a separate parameter change. This approach has the potential to lead to enhanced flexibility in the morphing ability of a nonlinear computing device.

The implementation of a sequence of logic functions in time, as described above, is now another mechanism through which computer architectures based upon the chaos computing approach can be optimized for better performance. In particular, we have shown explicitly how multiple sequentially connected nonlinear maps with unidirectional coupling (through state variables) or successive iterations of a single nonlinear map can perform bit-by-bit arithmetic addition through a sequence of logic operations with a small number of elements. With these fundamental ingredients in hand it is conceivable to build simple, fast, cost effective, and general-purpose computing devices, which are more flexible than statically wired hardware. It becomes clear that exploiting not just the pattern formation of nonlinear dynamical systems, but the formation of sequences of such patterns, produced naturally by such systems, may prove to be a key ingredient towards making nonlinear dynamical computational architectures a real alternative to conventional static logic computer architectures.

Acknowledgements

We gratefully acknowledge the support from the Office of Naval Research [grant No. N00014-02-11019] and from Chaologix, Inc.

References

[1] S. Sinha, W.L. Ditto, Phys. Rev. Lett. 81 (1998) 2156;
S. Sinha, W.L. Ditto, Phys. Rev. E 60 (1999) 363.
[2] S. Sinha, T. Munakata, W.L. Ditto, Phys. Rev. E 65 (2002) 036216.
[3] K. Murali, S. Sinha, W.L. Ditto, Int. J. Bifur. Chaos Appl. Sci. Eng. 13 (2003) 2669;
K. Murali, S. Sinha, W.L. Ditto, Phys. Rev. E 68 (2003) 016205;
K. Murali, S. Sinha, I. Raja Mohamed, Phys. Lett. A 339 (2005) 39.
[4] K.E. Chlouverakis, M.J. Adams, Electron. Lett. 41 (2005) 359.
[5] D. Cafagna, G. Grassi, Int. Sym. Signals Circuits Syst. (ISSCS 2005) 2 (2005) 749.
[6] M.R. Jahed-Motlagh, B. Kia, W.L. Ditto, S. Sinha, Int. J. Bifur. Chaos Appl. Sci. Eng. 17 (2007) 1955.
[7] K. Murali, S. Sinha, Phys. Rev. E 75 (2007) 025201(R).
[8] B. Prusha, J. Lindner, Phys. Lett. A 263 (1999) 105.
[9] J.P. Crutchfield, K. Young, Phys. Rev. Lett. 63 (1989) 105;
J.P. Crutchfield, Physica D 75 (1994) 11.
[10] N. Margolus, Physica D 10 (1984) 81;
T. Toffoli, N. Margolus, Cellular Automata Machines: A New Environment for Modelling, MIT Press, 1987;
T. Toffoli, N. Margolus, Physica D 47 (1990) 263.
[11] C. Moore, Phys. Rev. Lett. 64 (1990) 2354.
[12] A.V. Holden, J.V. Tucker, H. Zhang, M.J. Poole, Chaos 2 (1992) 367.
[13] A. Toth, K.J. Showalter, J. Chem. Phys. 103 (1995) 2058.
[14] M.M. Mano, Computer System Architecture, third ed., Prentice Hall, Englewood Cliffs, NJ, 1993;
T.C. Bartee, Computer Architecture and Logic Design, McGraw–Hill, New York, 1991.
[15] G. Taubes, Science 277 (1997) 1935.
[16] S. Sinha, Phys. Rev. E 49 (1994) 4832;
S. Sinha, Phys. Rev. E 63 (2001) 036212;
S. Sinha, W.L. Ditto, Phys. Rev. E 63 (2001) 056209;
S. Sinha, in: R. Sahadevan, M. Lakshmanan (Eds.), Nonlinear Systems, Narosa, 2002, p. 309.
[17] K. Murali, S. Sinha, Phys. Rev. E 68 (2003) 016210.
[18] K. Murali, A. Miliotis, W.L. Ditto, S. Sinha, M.L. Spano, Preprint, unpublished, 2009.