Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 11 (2013) 373 – 378

www.elsevier.com/locate/procedia

2nd International Through-life Engineering Services Conference

Demonstration of a self-recovering ALU using a convergent cellular automata

Richard McWilliam*, Philipp Schiefer, Alan Purvis

School of Engineering and Computing Sciences, Durham University, Science Labs, South Road, Durham, DH1 3LE

* Corresponding author. Tel.: +44 (0)191 3342418; fax: +44(0)191 334 2408. *E-mail address:* r.p.mcwilliam@durham.ac.uk

Abstract

This paper presents work in progress towards the demonstration of a self-restoring arithmetic logic unit (ALU) based on convergent cellular automata (CCA). The need for fault tolerance and self-recovery strategies for electronic circuits is discussed, with particular focus on well-known redundancy and reconfiguration approaches. Our CCA fault tolerant strategy is demonstrated via MATLAB simulation using fault injection. The combined roles of the CCA as coordination layer and restoration agent are discussed. Work in progress towards a hardware demonstration using VHDL description and FPGA hardware is also described.

Selection and peer-review under responsibility of the International Scientific Committee of the "2nd International Through-life Engineering Services Conference" and the Programme Chair – Ashutosh Tiwari

Keywords: Cellular automata; self-repairing electronics; FPGA

1. Introduction

We have previously demonstrated the composition of the convergent cellular automata (CCA) and considerations for its implementation for reliable electronics [1][2]. Here the standard cellular automata (CA) is modified by a set of constraints that initiate convergent behaviour i.e., that cause the CA to recursively reassemble a predetermined pattern. A CCA comprises memory in the form of look up tables (LUTs) for storing the rule set and a state machine to govern the cell behavior.

We consider the CCA as performing a coordination role to arrange functional logic to form the required I/O task. Associated with this are two sets of I/O: one for communicating CCA states between neighbouring cells and a second set for passing data between the functional logic. Once the rule set is programmed, the CCA begins coordination of its internal states and simultaneously the functional logic via unique mappings. This self-organisation occurs continually and is beneficial because the CCA is able to restore the correct logical functionality in the event of non – persistent soft errors without requiring external intervention.

We describe construction of a demonstration unit that uses a convergent rule set to create a robust full adder arithmetic logic unit (ALU) and memory mapped state machine. Progress towards a hardware demonstration unit is also presented.

The future scalability of this approach is an important consideration, particularly with regard to hardware platforms, where limitations of the field programmable gate array (FPGA) platform must be understood. In particular there is need for more efficient mapping of the dynamic functional logic to hardware and for run-time alterations to the rule set. We also discuss the more challenging problem of persistent soft errors and hard errors, where one or more cells do not recover functionality, in which case a reconfiguration scheme is proposed which requires the development of a more advanced CA behaviour.

Selection and peer-review under responsibility of the International Scientific Committee of the "2nd International Through-life Engineering Services Conference" and the Programme Chair – Ashutosh Tiwari
doi:10.1016/j.procir.2013.07.026

2. Fault tolerance and self-recovery in electronics

In this section we introduce the somewhat broad categories of redundancy and reconfiguration applied to self-recovering electronics. The scope of this work is currently limited to non-persistent soft errors typically resulting from single event upsets (SEUs), for which there are many responsible mechanisms. External triggers include high energy cosmic rays [3], thermal fluctuations and electromagnetic interference. Internally induced SEUs arise from noise present in semiconductor junctions and design flaws. Special constraints such as minimal voltage, a common requirement for modern ICs, tend to exacerbate susceptibility to external and internal SEUs [4]. The general trend to smaller feature size is also directly linked to SEU susceptibility and is of major concern for next-generation nanoscale electronics [5].

2.1. Fault tolerance via redundancy

Fault tolerant methods are often based on the concept of adding more elements to the system than strictly necessary in such a way that the system acquires the ability to 'absorb' fault events. Here we consider fixed elements introduced at the point of fabrication. Hence it is usually possible to calculate with high accuracy the predicted response of the resulting redundant circuit in the event of a restricted set of fault events. Sometimes statistical simulation must be used to form an accurate model of the circuit robustness. The majority of such mechanisms can be related to the early work of Von Neumann [6] involving unreliable components. Von Neumann argued that, in many situations, a reliable system can be constructed from components that are not 100% reliable, but whose individual reliabilities are reasonably high. This lead to the development of many practical configurations, including modular redundancy, signal interweaving and quadded logic [7].

2.2. Fault tolerance via reconfiguration

An alternative approach distinct from redundancy approaches is that of active reconfiguration, in which the circuit is altered in the event of external events such as an SEU. This requires additional memory and switching elements repeated throughout the design. Reconfiguration is arguably closest in principle to the concept of *self-repair* since it enables proactive steps to be taken in response to SEU or other malfunction events in an effort to re-establish operation. Of course this may occur under the auspices of external control from a global agent or high system level, in which case the qualifier *self* no longer applies.

Although desirable in terms of the flexibility offered by re-routing or re-programming logic, this approach tends to require high complexity due to the dynamic range of possible configurations. A centralised control entity is usually employed to control the overall configuration. A common example is the FPGA, which is configurable via programming bitstream. The bitstream defines how the FPGA's structured logic and embedded static random access memory (SRAM) are arranged in order to implement combinatorial and sequential logic operations, including LUTs (often defined as register transfer level (RTL) design). While this platform offers re-configurability, it is not generally possible to alter the configuration at run-time, except for limited cases and high end FPGA chips. Nevertheless, we use an FGPA here as a convenient programmable platform with which to demonstrate our CCA implementation.

Alternative platforms exist, which aim to exploit regular architectures based on cellular-inspired concepts and local neighbor interactions (for example, see [8]). The associated hardware is similar to an FPGA however the design is able to provide more sophisticated run-time localised reconfiguration.

3. CCA Design and simulation

Our goal is to demonstrate a self-recovering electronic circuit using a data protection scheme encoded within a CA, which then controls a functional layer of logic gates also arranged in cellular fashion.

3.1. Cellular automata as a data protection scheme

The CA is composed of a regular array of identical electronic cells. Each cell behaves essentially as a simple state machine driven by two inputs, a rule LUT and two outputs. The state transitions are illustrated in Fig. 1, where the idle state is interrupted by either Reset or Clk transition. A further LUT is present for output mapping and an output logic whose state is determined by the current cell state.

The CCA is formed by applying key restrictions to the CA:

- boundary cells are placed along the top and left hand sides of the CA,
- inter-cellular data flow is restricted along a single diagonal,
- the rule set is derived analytically and guarantees convergent behaviour.

3.2. Rule derivation and simulation

The method used to calculate the rule set is described in [1]. The required pattern expected to be generated by the CCA is fed into the rule generation algorithm, resulting in a set of rules and corresponding state mapping table. A MATLAB script is then used to check CCA pattern convergence, an example of which is shown in Fig. 2.

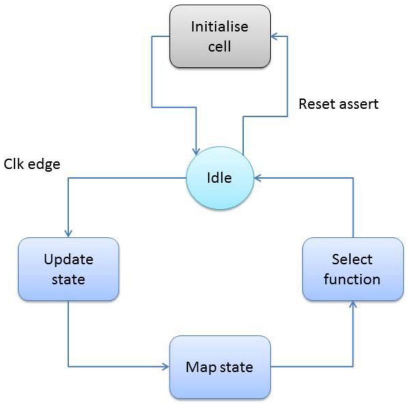
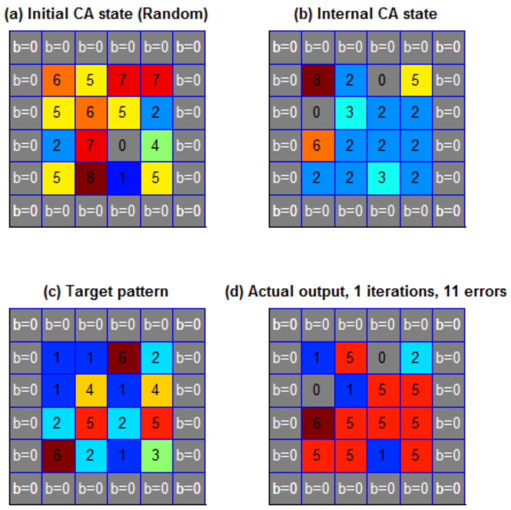


Fig. 1. State transition diagram for a CCA cell, showing Clk and Reset actions. The states shown refer to: *Idle* – wait for next trigger; *Initialise* cell – clear the cell state and functional logic to a predetermined initial state; *Update state* – change the cell internal state depending on the input values; *Map state* – update the mapped output state of the cell; S*elect function* – program the functional logic layer depending on the mapped cell output.



(a) Initial CA state (Random)

b=0	b=0	b=0	b=0	b=0	b=0
b=0	6	5	7	7	b=0
b=0	5	6	5	2	b=0
b=0	2	7	0	4	b=0
b=0	5	8	1	5	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(b) Internal CA state

b=0	b=0	b=0	b=0	b=0	b=0
b=0	8	2	0	5	b=0
b=0	0	3	2	2	b=0
b=0	6	2	2	2	b=0
b=0	2	2	3	2	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(c) Target pattern

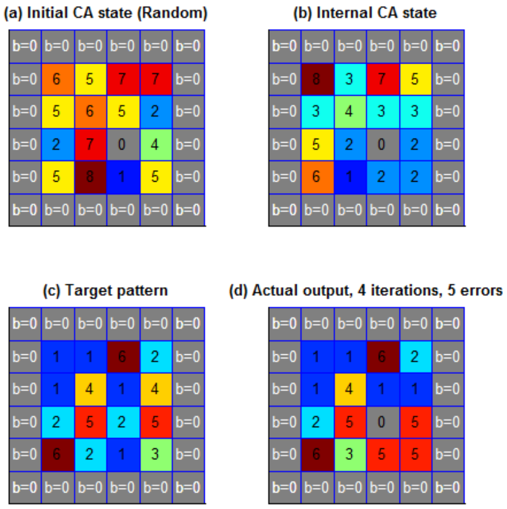
b=0	b=0	b=0	b=0	b=0	b=0
b=0	1	1	6	2	b=0
b=0	1	4	1	4	b=0
b=0	2	5	2	5	b=0
b=0	6	2	1	3	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(d) Actual output, 1 iterations, 11 errors

b=0	b=0	b=0	b=0	b=0	b=0
b=0	1	5	0	2	b=0
b=0	0	1	5	5	b=0
b=0	6	5	5	5	b=0
b=0	5	5	1	5	b=0
b=0	b=0	b=0	b=0	b=0	b=0

Fig. 2. Example of CCA simulation after first iteration. (a) initial randomised state of CCA (boundary cells denoted by “b=0” labels); (b) internal cell states after first iteration; (c) Target pattern used to derive rule set; (d) CCA output after mapping of output states according to state mapping table showing number of mismatches in comparison to target pattern.

After the first iteration, only 5 cells match the target pattern (cf. Fig. 2 (c) & (d)). After successive iterations, the correct pattern emerges as shown in Figs. 3(d) and Fig. 4(d). The CCA pattern converges upon a full refresh of all diagonals from top left to bottom right, hence the number of iterations required is r(c-1) where r and c are the number of rows and columns in the CCA respectively.



(a) Initial CA state (Random)

b=0	b=0	b=0	b=0	b=0	b=0
b=0	6	5	7	7	b=0
b=0	5	6	5	2	b=0
b=0	2	7	0	4	b=0
b=0	5	8	1	5	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(b) Internal CA state

b=0	b=0	b=0	b=0	b=0	b=0
b=0	8	3	7	5	b=0
b=0	3	4	3	3	b=0
b=0	5	2	0	2	b=0
b=0	6	1	2	2	b=0
b=0	b=0	b=0	b=0	b=0	b=0

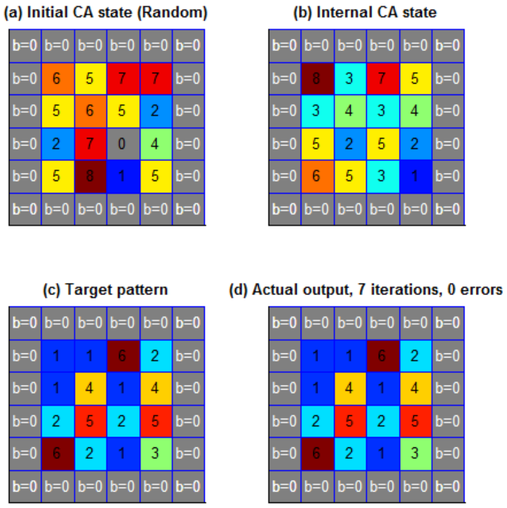
(c) Target pattern

b=0	b=0	b=0	b=0	b=0	b=0
b=0	1	1	6	2	b=0
b=0	1	4	1	4	b=0
b=0	2	5	2	5	b=0
b=0	6	2	1	3	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(d) Actual output, 4 iterations, 5 errors

b=0	b=0	b=0	b=0	b=0	b=0
b=0	1	1	6	2	b=0
b=0	1	4	1	1	b=0
b=0	2	5	0	5	b=0
b=0	6	3	5	5	b=0
b=0	b=0	b=0	b=0	b=0	b=0

Fig. 3. CCA state after 4 iterations, showing reduction of errors. See Fig. 2 for key.



(a) Initial CA state (Random)

b=0	b=0	b=0	b=0	b=0	b=0
b=0	6	5	7	7	b=0
b=0	5	6	5	2	b=0
b=0	2	7	0	4	b=0
b=0	5	8	1	5	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(b) Internal CA state

b=0	b=0	b=0	b=0	b=0	b=0
b=0	8	3	7	5	b=0
b=0	3	4	3	4	b=0
b=0	5	2	5	2	b=0
b=0	6	5	3	1	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(c) Target pattern

b=0	b=0	b=0	b=0	b=0	b=0
b=0	1	1	6	2	b=0
b=0	1	4	1	4	b=0
b=0	2	5	2	5	b=0
b=0	6	2	1	3	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(d) Actual output, 7 iterations, 0 errors

b=0	b=0	b=0	b=0	b=0	b=0
b=0	1	1	6	2	b=0
b=0	1	4	1	4	b=0
b=0	2	5	2	5	b=0
b=0	6	2	1	3	b=0
b=0	b=0	b=0	b=0	b=0	b=0

Fig. 4. CCA state after 7 iterations, showing convergence to correct pattern. See Fig. 2 for key.

3.3. Adder example

The example shown in Figs 2-4 was intended to coordinate an adder ALU, hence the target pattern can be mapped to a functional layer that implements the adder. This mapping is illustrated in Fig. 5. The CCA coordinating layer requires 20 rules and 8 state mappings (which comprises 6 original states and 2 additional states) which represents a modest rule set size. The cells having state ‘6’ are in fact unallocated and may be used for fault-tolerant reconfiguration in future implementations.

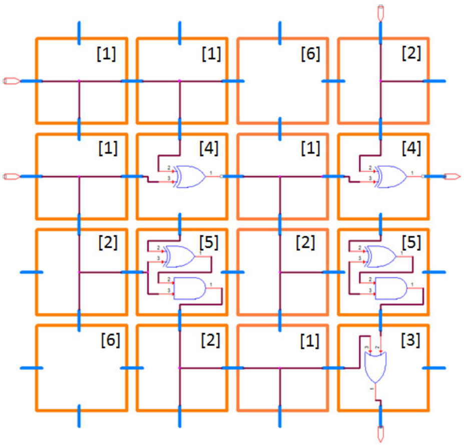


Fig. 5. Functional logic mapping for adder ALU (corresponding CCA states shown in square brackets).

The protection of the adder ALU is secured when the CCA successfully updates the correct coordination pattern.

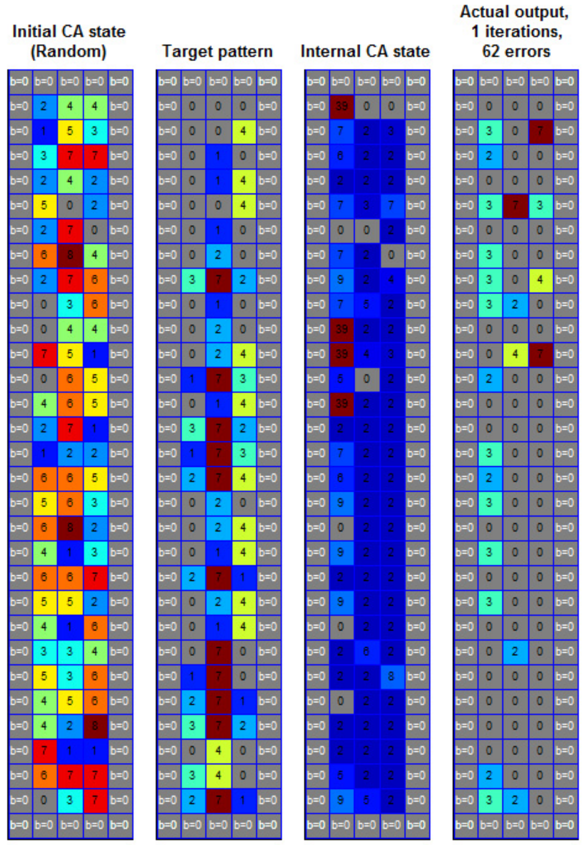
3.4. LUT example

We illustrate a more complex example by encoding a LUT memory map for a real state machine. In this example, the state machine behaviour is converted to a LUT so that the data becomes easier to protect. We are then able to apply the benefits of CCA pattern reconstruction to protect the LUT. In this case each 8-bit binary LUT entry is converted into segmented entries of the form: [pair, tuple, tuple], which are then converted into integer form. This form of segmentation is useful when the LUT entry is naturally split into different data field (such as pointer address, input comparison bits, output assert bits), each of which is protected by a CCA cell. There is no functional logic mapping in this example; the CCA output directly corresponds to the LUT entries. The resulting target pattern can be seen in Fig. 6, along with the initial iteration result. Fig. 7 shows convergence towards the target pattern after 31 iterations. This example requires 102 rules and 39 mapped states, illustrating the increasing memory overhead incurred for more complex examples.

3.5. Fault injection

The initial random CCA state represents a scrambling of every cell internal state, which may occur at power on (before Reset) or after a significant non-persistent SEU capable of affecting every cell. We can further illustrate the effect of single cell upsets and recovery by initiating fault injection. In Fig 8. a fault is injected into cell (2,2) such that the cell state increments from ‘4’ to ‘5’, This could be as a result of a logic ‘stuck at’ error. The pattern is reconstructed once the fault clears (Fig. 9). Since the cells refresh in a fixed diagonal direction, an erroneous cell near the upper left will cause significant damage to the CCA pattern until the fault clears.

Any number of faults can be simulated via fault masking process implemented in MATLAB.



Initial CA state (Random)

b=0	b=0	b=0	b=0	b=0
b=0	2	4	4	b=0
b=0	1	5	3	b=0
b=0	3	7	7	b=0
b=0	2	4	2	b=0
b=0	5	0	2	b=0
b=0	2	7	0	b=0
b=0	6	8	4	b=0
b=0	2	7	6	b=0
b=0	0	3	6	b=0
b=0	0	4	4	b=0
b=0	7	5	1	b=0
b=0	0	6	5	b=0
b=0	4	6	5	b=0
b=0	2	7	1	b=0
b=0	1	2	2	b=0
b=0	6	6	5	b=0
b=0	5	6	3	b=0
b=0	6	8	2	b=0
b=0	4	1	3	b=0
b=0	6	6	7	b=0
b=0	5	5	2	b=0
b=0	4	1	6	b=0
b=0	3	3	4	b=0
b=0	5	3	6	b=0
b=0	4	5	6	b=0
b=0	4	2	8	b=0
b=0	7	1	1	b=0
b=0	6	7	7	b=0
b=0	0	3	7	b=0
b=0	b=0	b=0	b=0	b=0

Target pattern

b=0	b=0	b=0	b=0	b=0
b=0	0	0	0	b=0
b=0	0	0	4	b=0
b=0	0	1	0	b=0
b=0	0	1	4	b=0
b=0	0	0	4	b=0
b=0	0	1	0	b=0
b=0	0	2	0	b=0
b=0	3	7	2	b=0
b=0	0	1	0	b=0
b=0	0	2	0	b=0
b=0	0	2	4	b=0
b=0	1	7	3	b=0
b=0	0	1	4	b=0
b=0	3	7	2	b=0
b=0	1	7	3	b=0
b=0	2	7	4	b=0
b=0	0	2	0	b=0
b=0	0	2	4	b=0
b=0	0	1	4	b=0
b=0	2	7	1	b=0
b=0	0	2	4	b=0
b=0	0	1	4	b=0
b=0	0	7	0	b=0
b=0	1	7	0	b=0
b=0	2	7	1	b=0
b=0	3	7	2	b=0
b=0	0	4	0	b=0
b=0	3	4	0	b=0
b=0	2	7	1	b=0
b=0	b=0	b=0	b=0	b=0

Internal CA state

b=0	b=0	b=0	b=0	b=0
b=0	39	0	0	b=0
b=0	7	2	3	b=0
b=0	6	2	2	b=0
b=0	2	2	2	b=0
b=0	7	3	7	b=0
b=0	0	0	2	b=0
b=0	7	2	0	b=0
b=0	9	2	4	b=0
b=0	7	5	2	b=0
b=0	39	2	2	b=0
b=0	39	4	3	b=0
b=0	5	0	2	b=0
b=0	39	2	2	b=0
b=0	2	2	2	b=0
b=0	7	2	2	b=0
b=0	6	2	2	b=0
b=0	9	2	2	b=0
b=0	0	2	2	b=0
b=0	9	2	2	b=0
b=0	2	2	2	b=0
b=0	9	2	2	b=0
b=0	0	2	2	b=0
b=0	2	6	2	b=0
b=0	2	2	8	b=0
b=0	0	2	2	b=0
b=0	2	2	2	b=0
b=0	2	2	2	b=0
b=0	5	2	2	b=0
b=0	9	5	2	b=0
b=0	b=0	b=0	b=0	b=0

Actual output, 1 iterations, 62 errors

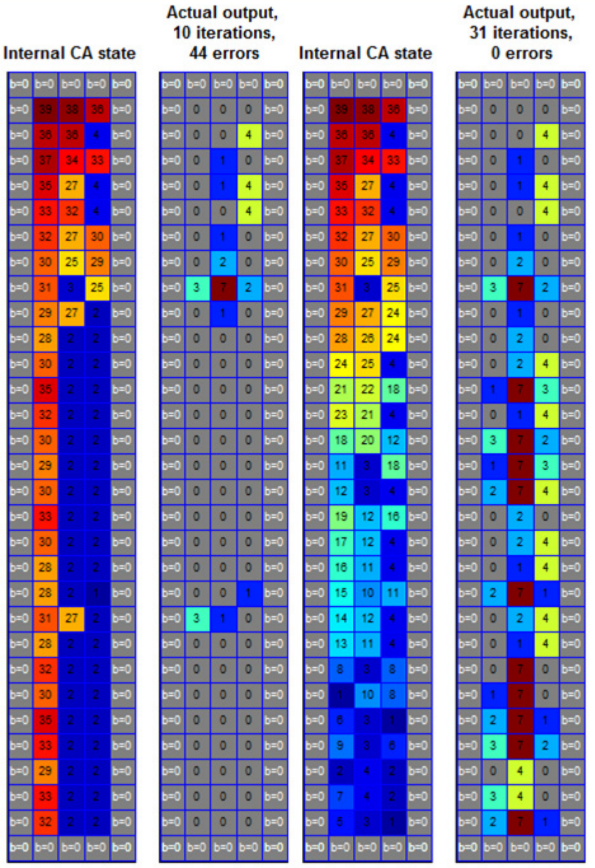
b=0	b=0	b=0	b=0	b=0
b=0	0	0	0	b=0
b=0	3	0	7	b=0
b=0	2	0	0	b=0
b=0	0	0	0	b=0
b=0	3	7	3	b=0
b=0	0	0	0	b=0
b=0	3	0	0	b=0
b=0	3	0	4	b=0
b=0	3	2	0	b=0
b=0	0	0	0	b=0
b=0	0	4	7	b=0
b=0	2	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	3	0	0	b=0
b=0	2	0	0	b=0
b=0	3	0	0	b=0
b=0	0	0	0	b=0
b=0	3	0	0	b=0
b=0	0	0	0	b=0
b=0	3	0	0	b=0
b=0	0	0	0	b=0
b=0	0	2	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	2	0	0	b=0
b=0	3	2	0	b=0
b=0	b=0	b=0	b=0	b=0

Fig. 6. CCA implementation of 29x3 LUT memory map showing (left to right) Initial random CCA configuration, LUT memory map, internal CCA state after 1 iteration, corresponding CCA mapped output.

4. FPGA Implementation

In order to produce a functional FPGA demonstrator each CCA cell was modelled using VDHL and schematically laid out using Xilinx IDE. The CCA cell comprises a rule LUT and a functional LUT and I/O logic. This can usefully be described as a VHDL entity that describes the behaviour of the cell. A VHDL test bench is used to test a 16 cell array and various rule sets and to check the functional logic behaviour. The top level schematic is shown in Fig. 10, where the arrangement of 16 cells can be seen. Boundary values are supplied by direct integer definitions at schematic level.

A touch screen is under development to provide a convenient method of interacting with the CCA such that faults may be injected to disturb the correct states of cells. When faults are induced the CA then automatically reassembles the correct global state. Key design decisions can be evaluated, in particular the suitability of Commercial off the shelf (COTS) reconfigurable hardware, logic design and recovery mechanism.



Internal CA state

b=0	b=0	b=0	b=0	b=0
b=0	39	38	36	b=0
b=0	36	36	4	b=0
b=0	37	34	33	b=0
b=0	35	27	4	b=0
b=0	33	32	4	b=0
b=0	32	27	30	b=0
b=0	30	25	29	b=0
b=0	31	3	25	b=0
b=0	29	27	2	b=0
b=0	28	2	2	b=0
b=0	30	2	2	b=0
b=0	35	2	2	b=0
b=0	32	2	2	b=0
b=0	30	2	2	b=0
b=0	29	2	2	b=0
b=0	30	2	2	b=0
b=0	33	2	2	b=0
b=0	30	2	2	b=0
b=0	28	2	2	b=0
b=0	28	2	1	b=0
b=0	31	27	2	b=0
b=0	28	2	2	b=0
b=0	32	2	2	b=0
b=0	30	2	2	b=0
b=0	35	2	2	b=0
b=0	33	2	2	b=0
b=0	29	2	2	b=0
b=0	33	2	2	b=0
b=0	32	2	2	b=0
b=0	b=0	b=0	b=0	b=0

Actual output, 10 iterations, 44 errors

b=0	b=0	b=0	b=0	b=0
b=0	0	0	0	b=0
b=0	0	0	4	b=0
b=0	0	1	0	b=0
b=0	0	1	4	b=0
b=0	0	0	4	b=0
b=0	0	1	0	b=0
b=0	0	2	0	b=0
b=0	3	7	2	b=0
b=0	0	1	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	1	b=0
b=0	3	1	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	0	0	0	b=0
b=0	b=0	b=0	b=0	b=0

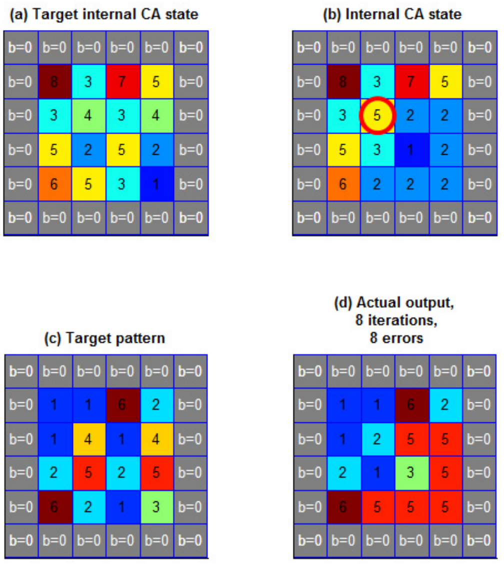
Internal CA state

b=0	b=0	b=0	b=0	b=0
b=0	39	38	36	b=0
b=0	36	36	4	b=0
b=0	37	34	33	b=0
b=0	35	27	4	b=0
b=0	33	32	4	b=0
b=0	32	27	30	b=0
b=0	30	25	29	b=0
b=0	31	3	25	b=0
b=0	29	27	24	b=0
b=0	28	26	24	b=0
b=0	24	25	4	b=0
b=0	21	22	18	b=0
b=0	23	21	4	b=0
b=0	18	20	12	b=0
b=0	11	3	18	b=0
b=0	12	3	4	b=0
b=0	19	12	16	b=0
b=0	17	12	4	b=0
b=0	16	11	4	b=0
b=0	15	10	11	b=0
b=0	14	12	4	b=0
b=0	13	11	4	b=0
b=0	8	3	8	b=0
b=0	1	10	8	b=0
b=0	6	3	1	b=0
b=0	9	3	6	b=0
b=0	2	4	2	b=0
b=0	7	4	2	b=0
b=0	5	3	1	b=0
b=0	b=0	b=0	b=0	b=0

Actual output, 31 iterations, 0 errors

b=0	b=0	b=0	b=0	b=0
b=0	0	0	0	b=0
b=0	0	0	4	b=0
b=0	0	1	0	b=0
b=0	0	1	4	b=0
b=0	0	0	4	b=0
b=0	0	1	0	b=0
b=0	0	2	0	b=0
b=0	3	7	2	b=0
b=0	0	1	0	b=0
b=0	0	2	0	b=0
b=0	0	2	4	b=0
b=0	1	7	3	b=0
b=0	0	1	4	b=0
b=0	3	7	2	b=0
b=0	1	7	3	b=0
b=0	2	7	4	b=0
b=0	0	2	0	b=0
b=0	0	2	4	b=0
b=0	0	1	4	b=0
b=0	2	7	1	b=0
b=0	0	2	4	b=0
b=0	0	1	4	b=0
b=0	0	7	0	b=0
b=0	1	7	0	b=0
b=0	2	7	1	b=0
b=0	3	7	2	b=0
b=0	0	4	0	b=0
b=0	3	4	0	b=0
b=0	2	7	1	b=0
b=0	b=0	b=0	b=0	b=0

Fig. 7. CCA convergence to LUT memory map after 10 and 31 iterations.



(a) Target internal CA state

b=0	b=0	b=0	b=0	b=0	b=0
b=0	8	3	7	5	b=0
b=0	3	4	3	4	b=0
b=0	5	2	5	2	b=0
b=0	6	5	3	1	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(b) Internal CA state

b=0	b=0	b=0	b=0	b=0	b=0
b=0	8	3	7	5	b=0
b=0	3	5	2	2	b=0
b=0	5	3	1	2	b=0
b=0	6	2	2	2	b=0
b=0	b=0	b=0	b=0	b=0	b=0

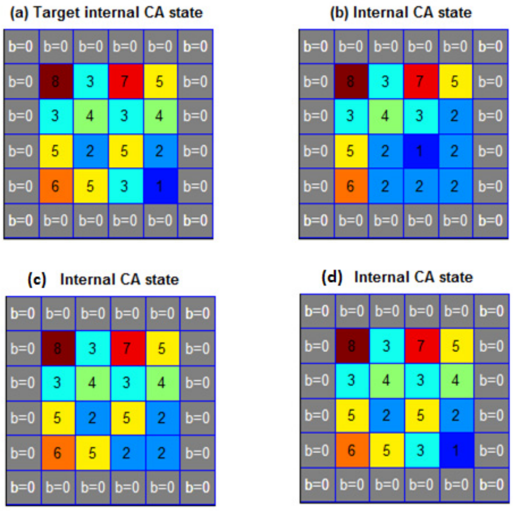
(c) Target pattern

b=0	b=0	b=0	b=0	b=0	b=0
b=0	1	1	6	2	b=0
b=0	1	4	1	4	b=0
b=0	2	5	2	5	b=0
b=0	6	2	1	3	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(d) Actual output, 8 iterations, 8 errors

b=0	b=0	b=0	b=0	b=0	b=0
b=0	1	1	6	2	b=0
b=0	1	2	5	5	b=0
b=0	2	1	3	5	b=0
b=0	6	5	5	5	b=0
b=0	b=0	b=0	b=0	b=0	b=0

Fig. 8. Result of fault injection for CCA adder. (a) intended internal CCA pattern; (b) resulting internal pattern with fault (circled); (c) target pattern; (d) actual output pattern.



(a) Target internal CA state

b=0	b=0	b=0	b=0	b=0	b=0
b=0	8	3	7	5	b=0
b=0	3	4	3	4	b=0
b=0	5	2	5	2	b=0
b=0	6	5	3	1	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(b) Internal CA state

b=0	b=0	b=0	b=0	b=0	b=0
b=0	8	3	7	5	b=0
b=0	3	4	3	2	b=0
b=0	5	2	1	2	b=0
b=0	6	2	2	2	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(c) Internal CA state

b=0	b=0	b=0	b=0	b=0	b=0
b=0	8	3	7	5	b=0
b=0	3	4	3	4	b=0
b=0	5	2	5	2	b=0
b=0	6	5	2	2	b=0
b=0	b=0	b=0	b=0	b=0	b=0

(d) Internal CA state

b=0	b=0	b=0	b=0	b=0	b=0
b=0	8	3	7	5	b=0
b=0	3	4	3	4	b=0
b=0	5	2	5	2	b=0
b=0	6	5	3	1	b=0
b=0	b=0	b=0	b=0	b=0	b=0

Fig. 9. Recovery of CCA internal state. (a) target pattern; (b-d) successive recovery of target pattern.

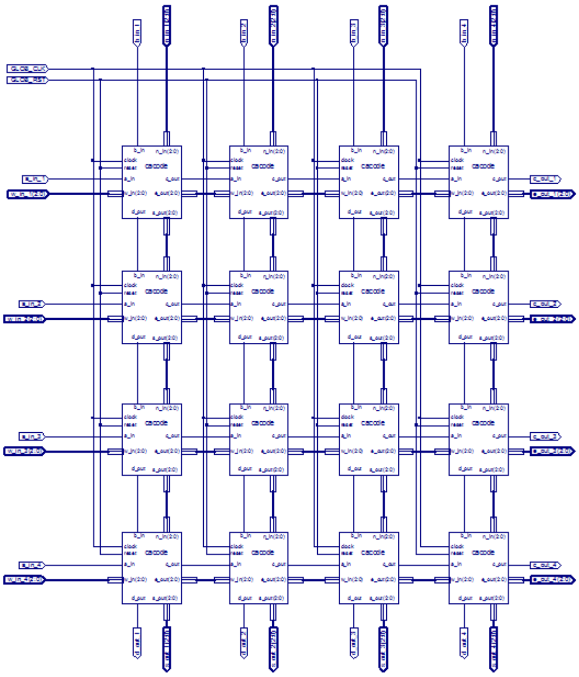


Fig. 10. Schematic layout for FPGA design. Each block contains VHDL description of a cell.

5. Discussion and conclusions

The current scope of this work is limited to non-persistent soft errors; however we are investigating how dynamic CCA rules or boundary vectors enable reconfiguration in the event of persistent soft errors or even hard fault situations. For example, the redundant cells seen in Fig. 5 are available for assignment but the CCA rule set cannot yet perform reconfiguration steps in an autonomous fashion. This problem has been investigated by others for multi-core processors [9], however the rule sets used were not convergent. This means that, whilst the dynamic CA was able to generate new configurations in the event of a processor halt event (including inter-processor dependencies), it does not offer continuous self-recovery for temporary errors in the same way. This is due to the vastly more complex nature of processors in comparison to simple functional logic meaning that the processor state cannot be mapped to a simple functional LUT.

Further FPGA testing will include hardware fault injection and removal of the Clk signal to investigate asynchronous operation, where the precise updating of inter-cell signals will depend upon interconnection delays and therefore the unique RTL synthesis.

We observed that soft errors occurring within the CCA are mitigated by inter-cellular signals, which reconstruct the correct pattern when the fault cell recovers. However, significant damage occurs when the faulty cell is near the top left corner. We are investigating alternative strategies that vary the direction of cell update in order to locate the position of the fault cell and potentially quarantine its state until it recovers normal operation, thus limiting the extent of damage.

Acknowledgements

This work was carried out with the support of the EPSRC Innovative Centre for Through-life Engineering Services.

References

[1] D. Jones, R. McWilliam, A. Purvis, “Designing convergent cellular automata.,” *Biosystems*, vol. 96, no. 1, pp. 80–85, 2008.

[2] R. McWilliam, A. Purvis, D. Jones, P. Schiefer, R. Frei, A. Tiwari, M. Zhu, “Self–repairing Electronic Logic Units Based on Convergent Cellular Automata,” in *1st International Conference on Through–life Engineering Services*, Shrivenham, England, pp. 353–360, 2012.

[3] M. Nicolaidis, *Soft Errors in Modern Electronic Systems*. Springer, 2010.

[4] A. R. Alameldeen, Z. Chishti, C. Wilkerson, W. Wu, and S.-L. Lu, “Adaptive Cache Design to Enable Reliable Low-Voltage Operation,” *Ieee Trans. Comput.*, vol. 60, no. 1, pp. 50–63, 2011.

[5] J. Han, J. Gao, P. Jonker, Y. Qi, and J. A. B. Fortes, “Toward hardware-redundant, fault-tolerant logic for nanoelectronics,” *Ieee Des. Test Comput.*, vol. 22, no. 4, pp. 328 – 339, 2005.

[6] J. Von Neumann, “Probabilistic logics and the synthesis of reliable organisms from unreliable components,” *Autom. Stud.*, vol. 34, pp. 43–98, 1956.

[7] P. A. Jensen, “Quadded NOR Logic,” *Ieee Trans. Reliab.*, vol. R-12, no. 3, pp. 22 –31, 1963.

[8] A. J. Greensted and A. M. Tyrrell, “RISA: A Hardware Platform for Evolutionary Design,” in *IEEE Workshop on Evolvable and Adaptive Hardware, 2007. WEAH 2007*, pp. 1 –7, 2007

[9] M. Kawanaka, M. Tsunoyama, and S. Naito, “A fault-tolerant parallel processor modeled by a two-dimensional linear cellular automaton,” *Syst. Comput. Jpn.*, vol. 25, no. 6, pp. 1–11, 2007.