ARITHMETIC OPERATIONS ON SELF-REPLICATING CELLULAR AUTOMATA

Enrico Petraglio, Jean-Marc Henry, Gianluca Tempesti
Logic Systems Laboratory
Swiss Federal Institute of Technology
Lausanne, Switzerland
E-Mail: enrico.petraglio@epfl.ch

Abstract

In this paper, we present a possible implementation of arithmetic functions (notably, addition and multiplication) using self-replicating cellular automata. The operations are performed by storing a dedicated program (sequence of states) on self-replicating loops, and letting the loops retrieve the operands, exchange data among themselves, and perform the calculations according to a set of rules. To determine the rules required for addition and multiplication, we exploited an existing algorithm for computation in the cellular automata environment and adapted it to exploit the features of self-replicating loops. This approach allowed us to study a variety of issues (synchronization, data exchange, etc.) related to the use of self-replicating machines for complex operations.

Introduction

The history of self-replicating cellular automata (CAs) has been marked by two major events. The first is von Neumann's development of his *universal constructor* [1], an automaton capable at the same time of universal construction, that is, of constructing any other automaton given its description (and hence a copy of itself given its own description), and of universal computation, that is, of executing any given application. This automaton, unfortunately handicapped by its great complexity, was the starting point for much of the further research in the field [2,3,4]. The second major event in the study of self-replicating CAs is Christopher Langton's development of the automaton known as *Langton's loop* [5], an automaton where the features of universal construction and universal computation were sacrificed for the sake of simplicity. The result is a small automaton capable exclusively of self-replication, extensively used and ameliorated by Langton's successors [6,7].
The motivations behind the study of self-replication in the environment of cellular automata is not immediately obvious, since this environment presents many features (e.g., the unbalance between the size of the memory required to store the transition rules and the functionality of a single cell) which render it somewhat cumbersome for most practical applications. Nevertheless, CAs do provide a rigid mathematical framework which can be very useful to systematically develop new approaches to the problem of self-replication, approaches which can then be transferred to more "conventional" and "practical" environments. This is, in fact, the motivation of our own research into the field of self-replicating automata. In particular, we have attempted to re-introduce computation to self-replicating automata in order to develop a mechanism allowing for self-replication in very large scale integrated circuits [8,9,10,11].

In the course of our research, we have developed a set of computationally-useful self-replicating automata, notably by adding a Turing machine to Langton's loop [12], and, more importantly as far as this article is concerned, by developing a "programmable" automaton (Fig. 1), that is, a self-replicating automaton capable of storing and executing a user-specified program [10,13]. The versatility of this automaton, which we used extensively in the development of our hardware systems, was illustrated through a simple self-contained example (that is, a program with no external inputs) which, while useful as a demonstration tool, was nevertheless not very interesting from a computational point of view. In this article, we wish to show that it is indeed possible to perform computationally useful tasks using self-replicating automata by using our programmable automaton to execute some arithmetical operations, notably addition and multiplication, on binary numbers.

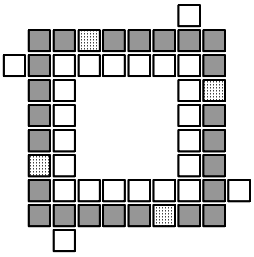


Figure 1 –Our self-replicating automaton

In order to implement these features, we used the *particle model* described by Steiglitz et al. [14], briefly introduced in the next section. We will then describe the implementation of this model in our automaton and its application to the operations of addition and multiplication, as well as to a combination of both. We will conclude with a few observations and remarks.

Theoretical Notions

To implement a binary addition function and a binary multiplication function on our automaton, we decided to exploit the model described by Steiglitz et al., since it is a model designed to operate within the CA environment and it can easily be adapted to self-replicating automata.
Obviously, the details of the operation of Steiglitz's algorithm had to be modified to fit the automaton, but we essentially maintained untouched the overall approach to the execution of addition and multiplication.

Binary Addition

To explain the mechanism used to add binary numbers, we will start with a simple example of a sum of two one-bit numbers. This example is shown in Figure 2.

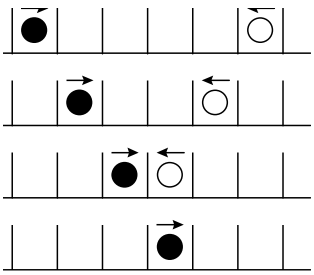


Figure 2 – Sum of two one-bit numbers.

To effect the sum, the two bits are stored in two cells which are moving towards each other. When the two cells collide, the right one is destroyed and the left one is transformed into a new right-moving cell which contains the result of the collision. The carry remains in place in the cell where the collision took place. In our example the left cell represent a logic 1 and the right one a logic 0. On the collision the sum is made and the new right-moving cell represent a logic 1, the result of the computation.
For the sum of binary numbers coded on more than one bit, the left and the right addends are represented by a sequence of cells, each cell representing a bit (one or zero). The two sequences move towards each other. A *processor cell* is placed between the two sequences of cells (Fig. 3). In each number, the least-significant bit lead the sequence, so that when the two numbers collide head-on at the processor cell, this last can add the bits in order of increasing significance.

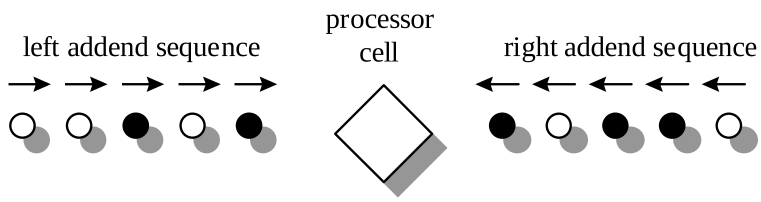


Figure 3 - Two data stream collide on one processor cell

After a collision the two incoming cells are destroyed, the processor cell computes the result and generates a new left-moving cell, which encodes the result of the first addition. After the creation of the "answer" cell, the processor stores the value of carry bit, which it will use to compute the result of the next collision between the bits of the two operands.

Binary Multiplication

For binary addition, we have seen that a single processor cell was sufficient for the computation. In the case of binary multiplication, we need a stream of processor cells, and more precisely double the number of bits of the multiplicands. Figure 4 shows the starting configuration of the multiplication.

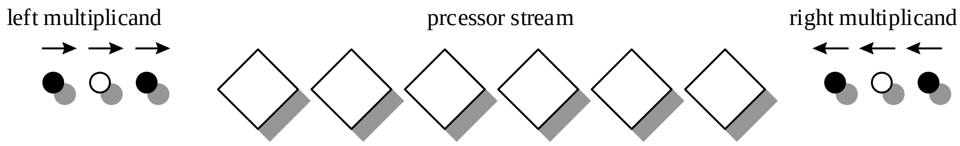


Figure 4 - Two data sequences collide in a processor stream

In this figure the left- and right-moving sequences of cells represent the two multiplicands and the processor stream is placed between the two sequences. To make the multiplication, the two multiplicands travel across all the processor cells. When two cells collide in a processor cell, this last computes the result according to the rules shown in Table 1. The two data cells then continue to travel across the processor stream.
When all the cells have traveled through the entire processor stream, the result of the multiplication is represented by the states of the processor stream's cells. Figure 5 shows an example of the multiplication of two 2-bit numbers.

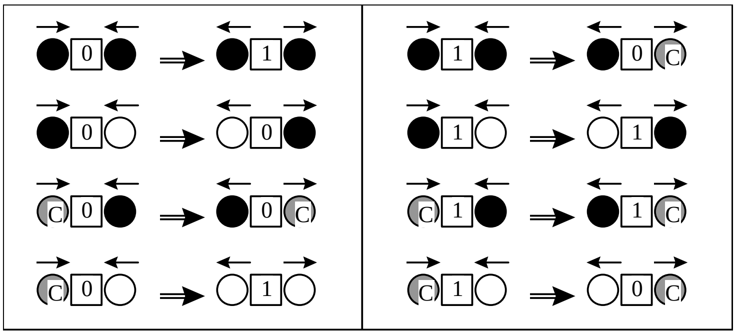


Table 1 – Rules for the processor cells

In figure 5, each row represent the state of the multiplication at the time *t*. In this example, the processor cells can have three different states. At the beginning (t=0), the cells' state is empty, while after the first collision the cells' state can be "1" or "0". At the end of the computation all of the processor cells are set to "1" or "0", and we can read the result on the processor stream: 11 x 11 = 1001, that is, in decimal notation, 3 x 3 = = 9.

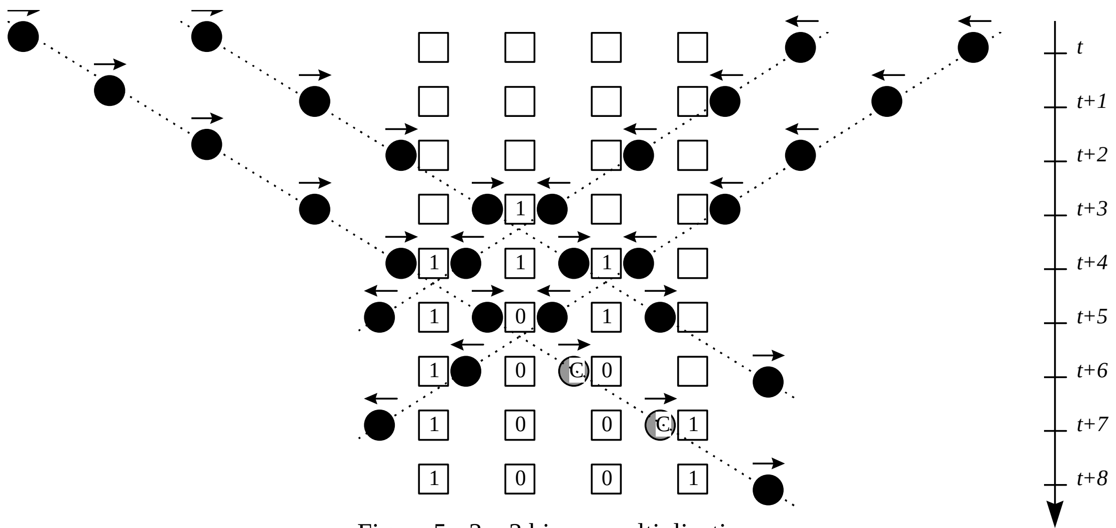


Figure 5 - 3 x 3 binary multiplication

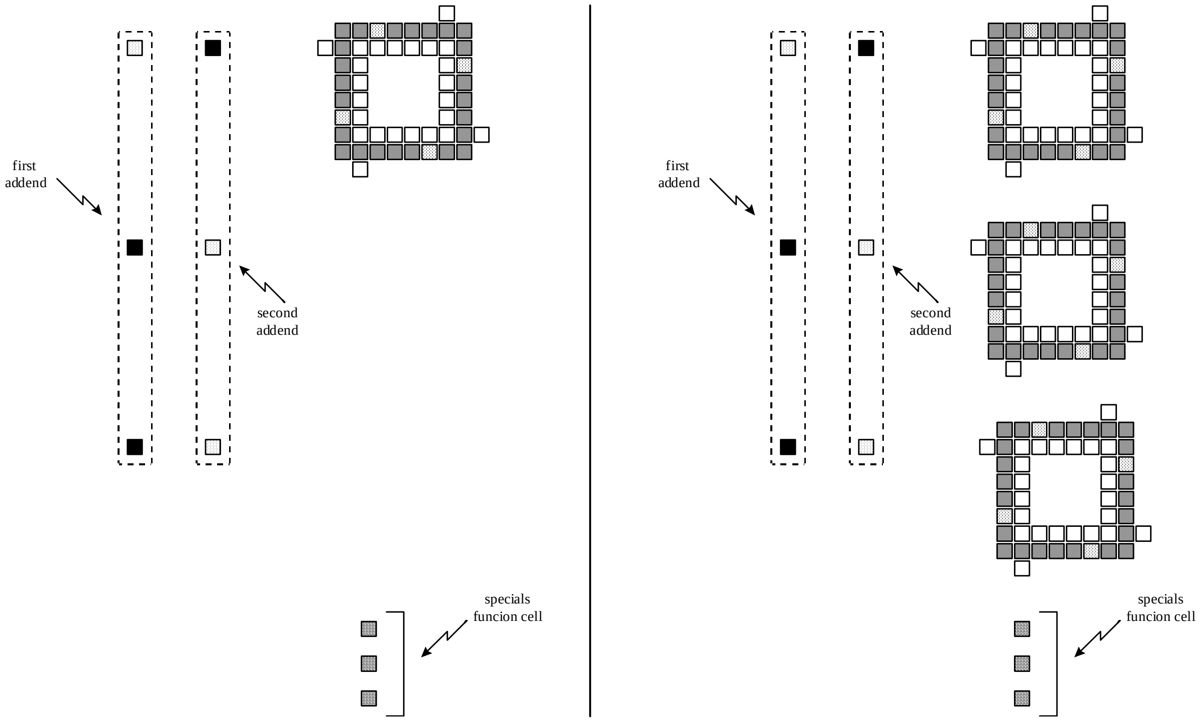
Implementation on Self-Replicating Loops

The programmable automaton we mentioned in the introduction (Fig. 1) consists of two concentric square loops: an inert internal loop (the *sheath*) and an active external *program loop*, containing the program to be executed along with the information used to direct the self-replication process. To duplicate itself, the automaton sends out four *constructing arms*, which build four new sheaths in the cardinal directions. When the sheath is complete, the automaton sends out the information contained in the program loop (the external loop of the original automaton). Finally, the constructing arms retract, completing the self-replication process and letting the four new automata attempt their own self-replication on the four cardinal directions. When an arm finds an obstacle (the border of the cellular array or an existing automaton), it retracts, abandoning the replication attempt. The self-replication will thus end only when all the available space (the cellular array) has been filled.

As we will see in this section, both the operation of our loop and that of Steiglitz's model had to be slightly modified to allow them to be merged, but the modifications were fairly minor and the basic concepts were not in the least altered.

Addition

To execute this function, one automaton is charged with computing the result of a single collision between two data cells, unlike the original algorithm, in which a single processor cell computed the entire result. The initial configuration of the adder (Fig. 6a) consists of a single loop, containing the program which implements the sum. This first loop is a slightly modified version of the original loop, in that it replicates in one direction only (downwards). As time progresses, a column of loops will be created. The replication process ends when the last automaton finds, in the place where it should replicate, a special cell (Fig. 6b). Upon finding this special cell, the bottom automaton generates a START signal which propagates upwards to the first automaton to tell it to begin the operation.



(a) Initial Configuration (b)End of self-replication

Figure 6 – Stream of automata

Once the first automaton has received the START signal, it looks to its left to find the bits it needs to add. It extends its constructing arm (Fig. 7a), retrieves the first bit it finds (least significant bit of the first number) and adds it to the second bit it finds (least significant bit of the second number). The arm then leaves in place the result of the computation and brings the carry bit back to the loop (Fig. 7b), which will propagate it to the next automaton (Fig. 7c). The process continues until the bottom loop is reached, signaling the end of the sum.

Once the operation is complete, the bottom loop will extend an arm downwards (as if to propagate the carry bit). The arm will meet one of three kinds of cells: a new START cell, which will activate a new sum, an END cell, which halts the operation of the automata, or an ACTIVATE cell, whose functionality will be explained below.

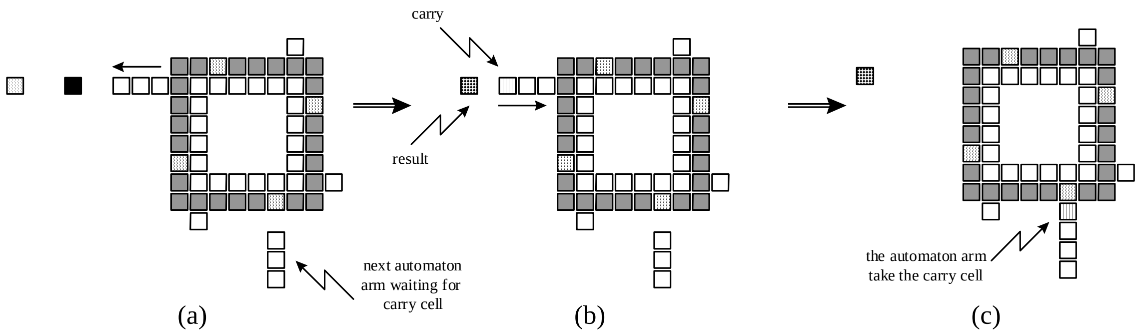


Figure 7 – Computation of a collision

Multiplication

As for the sum, the multiplication starts with a single loop, which replicates towards the right (unlike the sum) to create a stream of 2N automata (where N is the number of bits of the multiplicands).
The multiplication algorithm requires that the first collision between the data cells occur at a specific automaton, notably the Nth automaton from the right. This introduces some synchronization problems which complicate the execution considerably. The first complication is that a sequence of temporization signals, in the form of N-1 *shifting cells*, needs to be added in front of the left operand (Fig. 8).

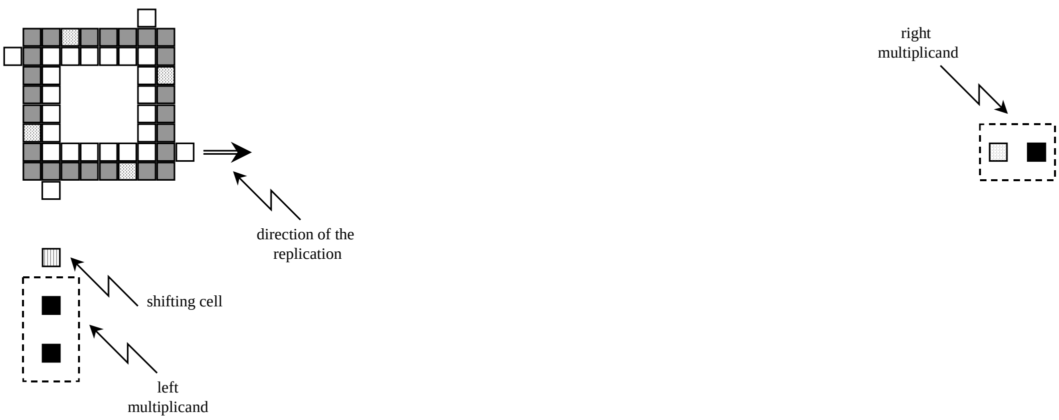


Figure 8 – Starting point of multiplication

The operation begins when the self-replication process has ended (i.e., when the replicating automata have filled all the available space) and the leftmost automaton has received a START signal. At this point, the leftmost automaton (which we will call Loop 1) starts retrieving the data cells of the left operand and propagating them to the right. Throughout the multiplication, Loop 1 will keep retrieving and propagating the data cells at a frequency of one data cell every three time steps (where one *time step* is the time required for an automaton to extend and retract its constructing arm).

The first shifting cell (the first cell of the left operand to be retrieved) propagates then to the right until it reaches the rightmost automaton (which we will call Loop 2N). Upon receiving the shifting cell, Loop 2N retrieves the first cell of the right operand and stores it. Each of the shifting cells traversing the automata will cause the right operand data cells to be shifted from the loop they are on to the loop to its left and a new data cell to be retrieved by Loop 2N.
After N-1 shifting cells have gone through, each of the bits of the right operand (except for the last one) are thus distributed on the N-1 rightmost loops. When the first left operand data cell arrives (behind the shifting cells) on Loop N+1 (the Nth automaton from the right), the first collision occurs (Fig. 9).
The collision process occurs between a data cell A on one loop and a data cell B on the loop to its right, according to the rules shown in Table 1. At the end of the collision process, the result of the collision and data cell B are stored on the left loop, along with a possible carry bit (which will taken into account when computing the next collision), while data cell A has been propagated to the right loop, where it will be used for the next collision. Each left operand data cell will thus collide with each right operand data cell, and the right operand will be shifted by one automaton to the right after being traversed by each left operand data cell. At the end of the multiplication, the right operand, stored on the N rightmost loops of the automaton, will be deleted by a special CLEAR cell, and the result will be stored on each of the loops.

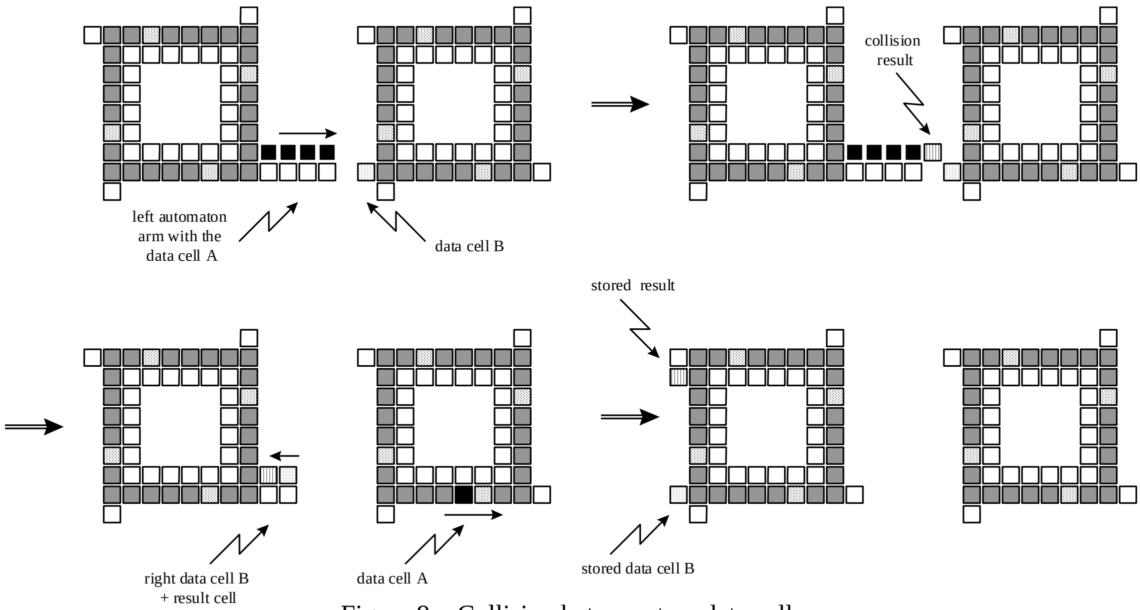


Figure 9 – Collision between two data cells

Combinations of Multiplication and Addition

In order to render its operation more "useful", our automaton was conceived so as to be able to realize combinations of operations. In particular, it can compute the multiplication of two results of sums. That is, it can compute any function of the form:

$$(A + B + \ldots) * (a + b + \ldots)$$

In order to compute this kind of function, we need a starting configuration similar to Fig. 10, which expands to the machine shown in Fig. 11 after self-replication.

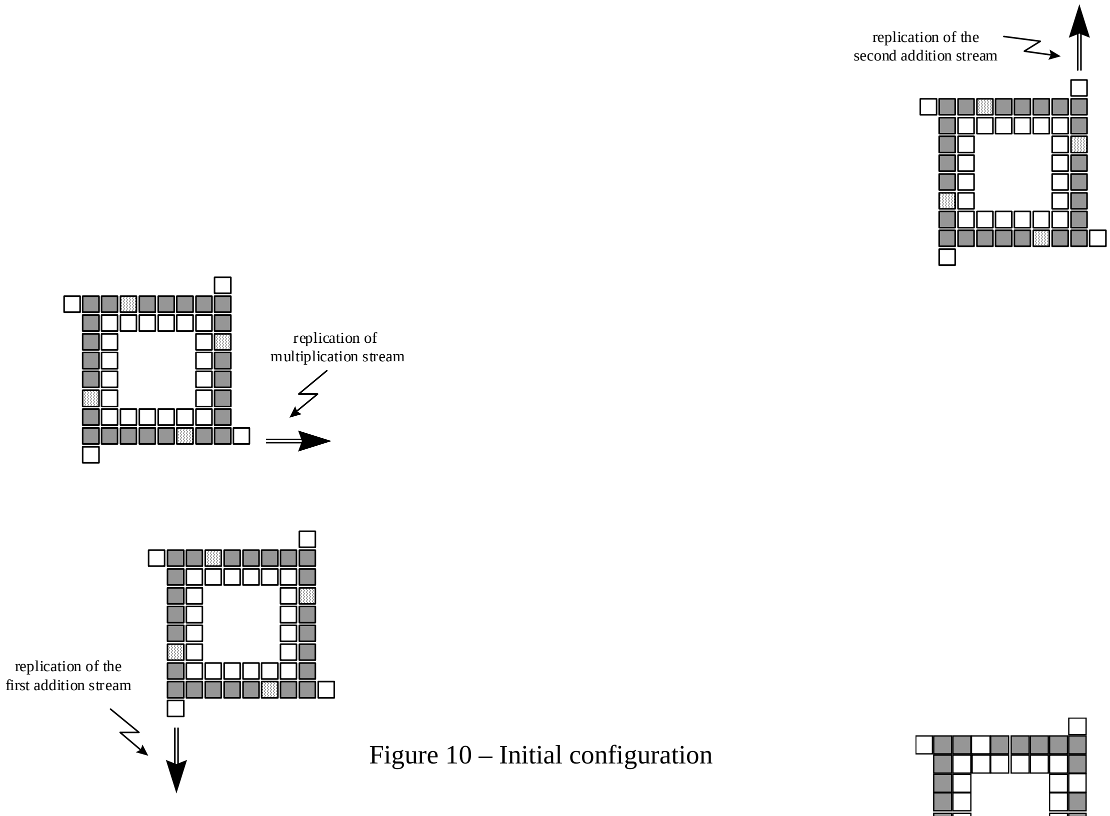


Figure 10 – Initial configuration

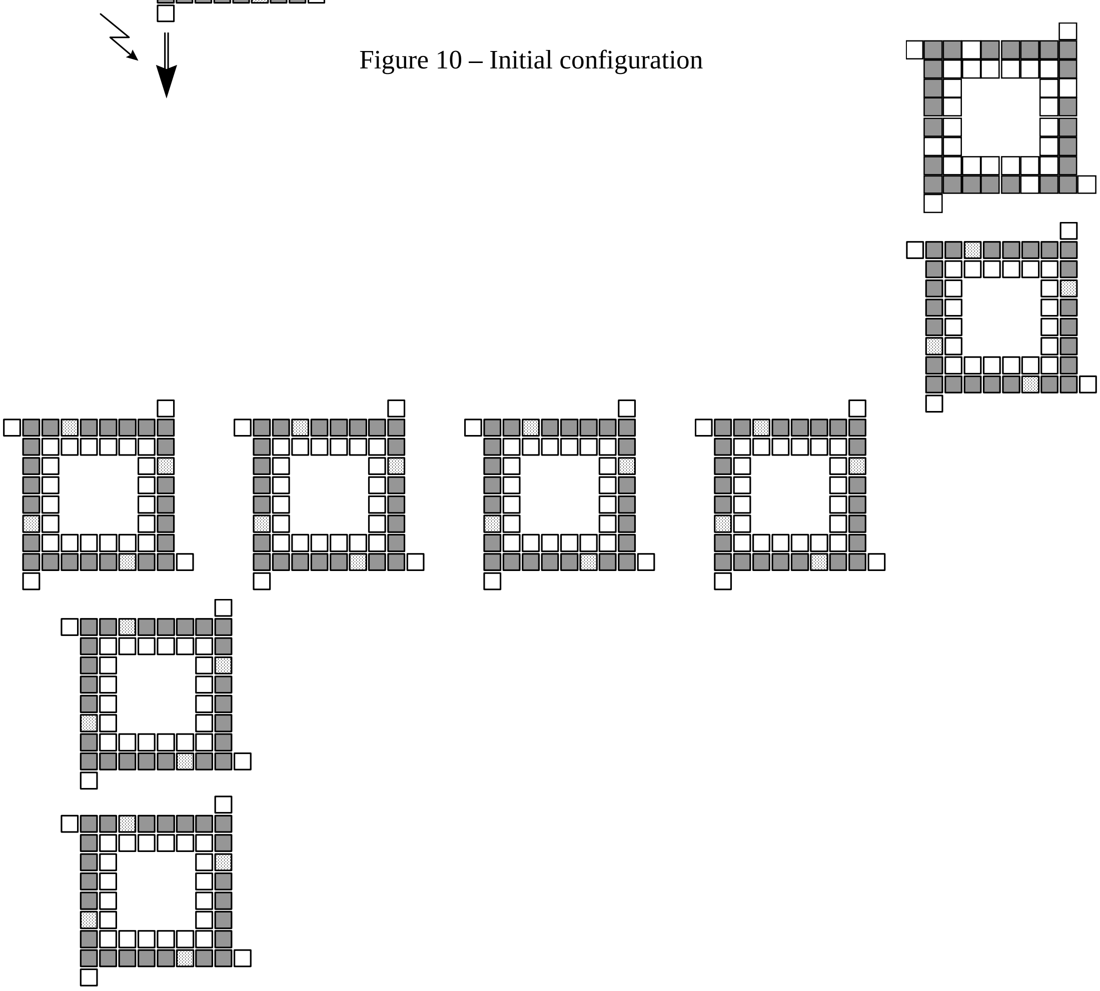


Figure 11 – End of self-reproduction

When the left and the right automata have completed their sums, they leave a special ACTIVATE cell (mentioned above) for the multiplier to retrieve. The latter will interpret this cell as a START signal, and execute the multiplication.
The two operations can thus be chained without difficulty, the only new feature being a *carriage cell* which will "reformat" the data generated by the adders into a form which the multiplier can use as an input (Fig. 12).

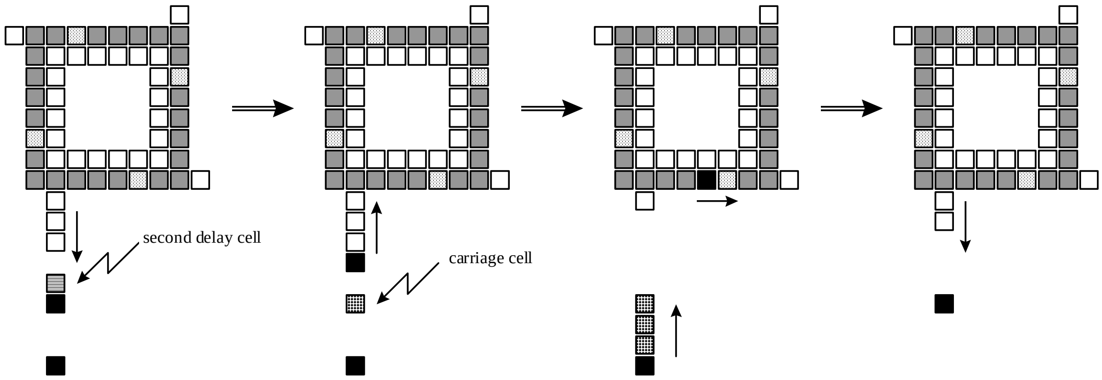


Figure 12 – Operation of the carriage cells.

Conclusion

The goal of the work presented in this article was to show that is possible, and indeed not exceedingly difficult, to exploit the capabilities of self-replicating automata (and notably our self-replicating programmable loops) to perform complex mathematical operations. To demonstrate this, we implemented the arithmetic operations of addition and multiplication using the algorithm described by Steiglitz et al. The resulting machines, while relatively complex (the final number of states required for combined sum and multiplication exceeds 30, including the states used only for self-replication), are nevertheless simple enough to be entirely simulated, and the use of the support provided by the programmable loops considerably simplified the finding of the relevant transition rules.
It should be noted that, while the automaton we designed is simple enough for simulation, it is extremely unlikely that such a system would ever be actually used for real-world computing. As we have mentioned, in fact, cellular automata are a useful environment for theoretical research but its real-world applications are few and not usually concerned with complex mathematical operations. Moreover, "pure" cellular automata do not contemplate the existence of external inputs, i.e., of data, such as mathematical operands, which is not present in the cellular space at time 0 (for example, in our system, the operands should clearly be inserted as needed, which would simplify considerably the operation of the automata).
Our aim, however, was not to develop a cellular automaton to be used in real-world applications. As mentioned in the introduction, our goal in studying this kind of structures is to determine what the advantages and constraints are in the use of self-replicating machines for complex operations, so as to be able to transfer these observations to the design of self-replicating integrated circuits. From this perspective, the work we presented is indeed interesting, in that it allows a number of observations:

1 Self-replication can be advantageously exploited to realize application-specific parallel systems by associating a self-replication mechanism and an execution unit.

2 The execution units need not be very powerful, as complex operations can be performed by many small identical units (the fundamental principle of parallelism).
3 Self-replication allows the systems to adapt their architecture to the problem (for example, by producing the correct number of execution units to exactly fit a given problem).
4 The problem of synchronizing the operation of all the units of the system is a major issue, as is the communication between the units.

This kind of information has been, and will be, extremely useful in the development of self-replicating machines and in our attempt to realize von Neumann's dream.

References

[1] J. von Neumann. *The Theory of Self-Reproducing Automata*. A. W. Burks, ed. University of Illinois Press, Urbana, IL, 1966.

[2] E.R. Banks. "Universality in Cellular Automata". In *Proc. IEEE 11th Annual Symposium on Switching and Automata Theory*, Santa Monica, CA, October 1970, pp. 194-215.

[3] E.F. Codd. *Cellular Automata*. Academic Press, New York, 1968.

[4] C. Lee. "Synthesis of a Cellular Computer". In *Applied Automata Theory*, Academic Press, London, 1968, pp 217-234.

[5] C. G. Langton. "Self-Reproduction in Cellular Automata". *Physica 10D*, 1984, pp. 135-144.

[6] J. Byl. "Self-Reproduction in Small Cellular Automata". *Physica 34D*, pp.295-299, 1989.

[7] J.A. Reggia, S.A. Armentrout, H.-H. Chou, Y. Peng. "Simple Systems That Exhibit Self-Directed Replication". *Scienc*e, Vol.259, 26 February 1993, pp. 1282-1287.

[8] D. Mange, M. Tomassini, eds. *Bio-inspired Computing Machines: Towards Novel Computational Architectures*. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[9] D. Mange, M. Goeke, D. Madon, A. Stauffer, G. Tempesti, S. Durand. "Embryonics: A New Family of Coarse-Grained Field-Programmable Gate Array with Self-Repair and Self-Reproducing Properties". In E. Sanchez, M. Tomassini, eds., *Towards Evolvable Hardware*, Lecture Notes in Computer Science, Springer, Berlin, 1996, pp. 197-220.

[10] G. Tempesti. *A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes*. Ph.D. Thesis No. 1827, EPFL, Lausanne, 1998.

[11] G. Tempesti, D. Mange, A. Stauffer. "Self-Replicating and Self-Repairing Multicellular Automata". *Artificial Life*, 4(3), 1998, pp. 259-282.

[12] J.-Y. Perrier, M. Sipper, J. Zahnd. "Toward a Viable, Self-Reproducing Universal Computer". *Physica 97D*, pp. 335-352, 1996.

[13] G. Tempesti. "A New Self-Reproducing Cellular Automaton Capable of Construction and Computation". *Proc. 3rd European Conference on Artificial Life*, Lecture Notes in Artificial Intelligence, 929, Springer Verlag, Berlin, 1995, pp. 555-563.

[14] K. Steiglitz, R. K. Squier, M. H. Jakubow. "Programmable Parallel Arithmetic in Cellular Automata using a Particle Model". *Complex Systems* 8, 1994, pp. 311-323.