1 Arithmetic Operations with Self-Replicating Loops

Enrico Petraglio, Gianluca Tempesti and Jean-Marc Henry

We present a possible collision-based implementation of arithmetic functions using a self-replicating cellular automaton capable of construction and computation. Our automaton makes use of some of the concepts developed by Langton for his self-replicating automaton, but provides the added advantage of being able to perform independent constructional and computational tasks along with self-replication. Our automaton is capable, like Langton's automaton and with comparable complexity, of simple self-replication, but it also provides (at the cost, naturally, of increased complexity) the option of attaching to the automaton an executable program which will be duplicated and executed in each of the copies of the automaton. The arithmetic functions that we have implemented 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 collision-based 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.

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 imbalance 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, cellular automata (CA) 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 within the framework of the Embryonics (embryonic electronics) project [4,5,10]. In particular, we have attempted to develop novel automata capable not only of self-replication, but also of performing useful computational tasks, and this in order to analyze the mechanisms involved in the replication process and to find a way to adapt these mechanisms to the development of very large-scale integrated circuits.

The history of self-replicating cellular automata basically begins with John von Neumann's research in the field of complex self-replicating machines. Advised by the mathematician Stanisław Ulam, he applied his concepts in the framework of a "cellular space", a two-dimensional grid of identical elements where each element (cell) is a finite state automaton whose next state is a function of its present state and of the present state of its four neighboring cells. Within this framework, von Neumann was able to conceive a self-replicating automaton endowed with the properties of both computational and constructional universality [11]. Unfortunately, the automaton was of such complexity that, further simplifications notwithstanding, even today's state-of-the-art computers lack the power to simulate it in its entirety.

The next significant event in the history of self-replicating automata was the development of the automaton commonly referred to as "Langton's loop" [3]. By dropping the requirements of computational and constructional universality, Langton created an automaton capable of non-trivial self-replication, that is an automaton where the replication is actively directed by the automaton itself, rather than being a mere consequence of the transition rules.

The automaton we introduce seeks to go beyond Langton's loop, which is capable exclusively of duplicating itself, by adding computational and constructional capabilities to self-replication. In fact, while our automaton is based on the utilization of a "loop" similar to that of Langton's automaton, we have modified the self-replicating mechanism so that it requires only a fraction of the data circulating in the loop to perform its task, thus making the remaining data available for other purposes. We will illustrate this novel capability using first a simple example of a program embedded into our self-replicating automaton, and then a much more complex application in which Steiglitz's collision-based computing model [8] is used in conjunction with a slightly modified version of our automaton to execute binary additions and multiplications.

This chapter is structured in two main parts, reflecting the two papers from which it is derived [6,9]. In the first part, we will begin (Sect. 1.1) with an overview of the cellular automata mentioned above (Von Neumann's universal constructor and Langton's loop), and a description of the main differences between these automata and our novel automaton. We will then (Sect. 1.2) describe in detail the operation of our self-replicating automaton, and provide an example of its constructional capabilities. Then, in the second part of the chapter, after introducing the collision-based computing model we adopted (Sect. 1.3), we will show how our automaton can be adapted to exploit this model in order to execute simple computational functions (Sect. 1.4).

1.1 Self-Replicating Cellular Automata

1.1.1 Von Neumann's Automaton

Von Neumann's self-replicating cellular automaton was a result of his interest in complex machines and their behavior [11]. His research led to the conclusion that the following characteristics should be present in a self-replicating machine:

- Computational universality, that is the ability to operate as a universal Turing machine, and thus to execute any computational task.
- Constructional universality, that is the ability to construct any kind of configuration in the cellular space starting from a given description; self-replication is then a particular case of universal construction.

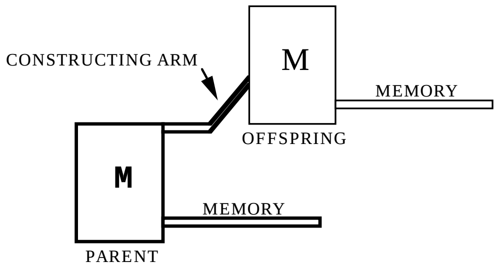


Fig. 1.1. Von Neumann's self-replicating automaton.

To implement these properties in a cellular automaton, von Neumann set out to design a universal constructor, i.e. an automaton capable of constructing, through the use of a "constructing arm", any configuration whose description can be stored on its input tape (Fig. 1.1). This universal constructor, therefore, is able, given its own description, to construct a copy of itself, thus achieving self-replication.

The automaton developed by von Neumann used tens of thousands of 29-state cells and a 5-cell neighborhood (the cell itself plus its four cardinal neighbors). Codd [2] and others managed to reduce the complexity of von Neumann's machine, but the automaton retains a level of complexity too high for simulation. In fact, while parts of the machine have been successfully simulated, the task of simulating the whole automaton is only now becoming barely feasible, given current technology.

Realizing von Neumann's dream of a self-replicating universal computing machine in actual hardware is one of the main goals of our Embryonics project [5].

1.1.2 Langton's Loop

Langton's automaton [3] is based on one of the components of Codd's universal constructor, namely the "periodic emitter" [2]. The automaton (Fig. 1.2) is essentially a square loop, with internal and external sheaths, where the data necessary for the construction of a duplicate loop circulate counterclockwise. Duplication is achieved by extending a constructing arm which will be forced to turn 90 degrees to the left at regular intervals corresponding to the size of one side of the loop. After three such turns the arm will have folded upon itself. When the new loop is closed, the constructing arm will retract and the new loop will be active, that is will be able to replicate itself as the original loop did. The original loop will then repeat the process by creating a second copy of itself in another direction, and finally "die" by losing the information within the loop. Given sufficient time, the automaton will replicate itself to fill the available space.

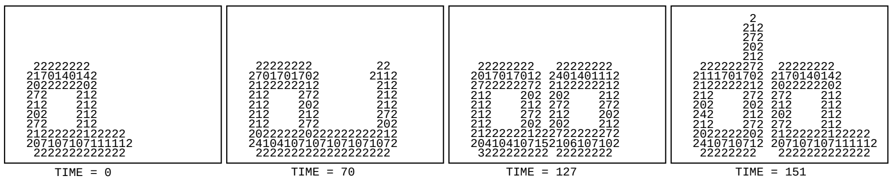


Fig. 1.2. Langton's Loop.

Langton's loop uses 8 states for each of the 86 non-quiescent cells making up its initial configuration, a 5-cell neighborhood, and a few hundred transition rules (the exact number depends on whether default rules are used and whether rotated rules are included in the count). Further simplifications to the automaton were introduced by Byl [1], who eliminated the internal sheath and reduced the number of states per cell, the number of transition rules, and the number of non-quiescent cells in the initial configuration. Reggia et al. [7] managed to remove also the external sheath. Given their low complexity, at least relative to von Neumann's automaton, all of the mentioned automata have been thoroughly simulated.

The main drawback of Langton's loop from our point of view is, of course, its inability to perform any task beyond self-replication, a drawback that we had to address in the development of our novel self-replicating automaton.

1.1.3 The New Automaton

Our automaton uses some of the concepts found in Langton's loop. In particular, we retain the concept of loop, which Langton himself derived from Codd's periodic emitter, to store the data dynamically. However, there are some substantial differences between our loop and Langton's automaton (Fig. 1.3):

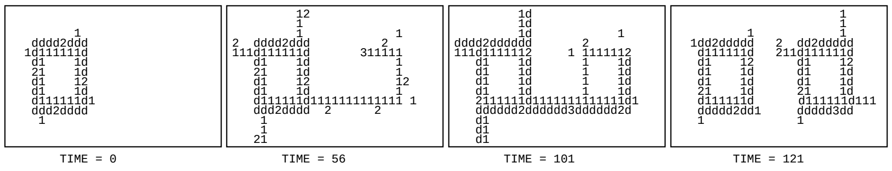


Fig. 1.3. Our Loop.

- We use a 9-cell neighborhood (the cell itself plus its 8 neighbors).
- As in Byl's version of Langton's loop, we use only one sheath, but contrary to Byl, we retain the internal sheath and eliminate the external one. This allows us to let the data in the loop circulate without the need for leading or trailing states (the 0s in Langton's loop). In addition to the internal sheath, we have four "gate cells" (in the same state as the sheath) outside the loop at the four corners of the automaton. These cells are initially in the "open" position, and will shift to the "closed" position once the copy is accomplished.
- We extend four constructing arms in the four cardinal directions at the same time, and thus create four copies of the original automaton in the four directions in parallel. When the arm meets an automaton already in place where the copy should be (which happens for all but the original automaton), it simply retracts and puts the corresponding gate cell in the closed position.
- Rather than being directed to advance, our constructing arm advances by default. As a consequence, it is necessary only to direct it to turn at the appropriate moment. This is done by sending periodic "messengers" to the tip of the constructing arm, which advanced at a slower pace with respect to the messengers.
- The arm does not immediately construct the entire loop. Rather, it constructs a sheath of the same size as the original. Once the sheath is ready, the data circulating in the loop is duplicated and the copy is sent along the constructing arm to wrap around the new sheath. When the new loop is completed, the constructing arm retracts and shifts the corresponding gate cell to the closed position.
- As a consequence of the above, rather than using all of the data in the loop to direct the constructing arm, we use only four of the cells circulating in the loop to generate the messengers. Since the only operation performed on the remaining data cells is duplication, they do not have to be in any particular state. In particular, they can be used as a "program", i.e., a set of states with their own transition rules which will then be applied alongside the self-replication to execute some function.
- Unlike Langton's loop, our loop does not "die" once duplication is complete, as the circulating data remains untouched by the self-replication process. Therefore, any program stored in the loop will be able to continue to execute. Also, it is possible to force the loop to try and duplicate

again in any of the four directions simply by shifting the corresponding gate cell back to the open position.
- When the duplicated loops arrive next to the border of the array, the constructing arm detects the border and retracts without attempting to duplicate the data. Thus, our automaton, unlike Langton's, does not crash when the duplication process reaches the edge of the cellular space.
- Because the replication process occurs in the four directions at the same time, the growth of the colony follows a symmetric pattern (Fig. 1.4), unlike the spiraling pattern of Langton's automaton.

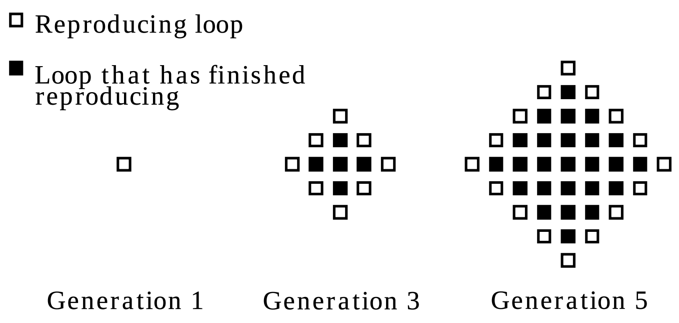


Fig. 1.4. Growth pattern.

As should be obvious from the above, while our loop owes to von Neumann the concept of constructing arm and to Langton (and/or Codd) the basic loop structure, it is in fact a very different automaton, endowed with some of the properties of both. This observation is valid both for the functionality of the automaton, which is capable of embarking on complex programs (unlike Langton's loop) but not physically separate machines (unlike von Neumann's constructor), and for its complexity.

In fact, as far as the complexity of the automaton is concerned, its estimation is more difficult than for Langton's loop, as it depends on the data circulating in the loop. The number of non-quiescent cells making up the initial configuration depends directly on the size of the circulating program. The more complex (i.e. the longer) the program, the larger the automaton. It should be noted, however, that the complexity of the self-replication process does not depend on the size of the loop. The number of states also depends on the complexity of the program. To the five "basic states" used for self-replication (see description below) must be added the "data states" (at least one) used in the program, which must be disjoint from the basic states. The number of transition rules is obviously a function of the number of data states: in the basic configuration, i.e., one data state, the automaton needs 692 rules (173 rules rotated in the four directions). By default, all cells remain in the same state.

The complexity of the basic configuration is therefore in the same order as that of Langton's and Byl's loops, with the proviso that it is likely to increase drastically if the data in the loop is used for some purpose. In fact, the number of rules in the automaton we have described grows as D^4, where D is the number of data states. A different version of the automaton limits the growth to D^3 (at the expense of some versatility), but the increase remains substantial.

In the next section we will describe in some detail the operation of the automaton in a small, basic configuration, and illustrate an example of a loop where a program has been included in the loop to demonstrate the construction capabilities of our automaton.

1.2 Description of the Automaton

1.2.1 Cellular Space and Initial Configuration

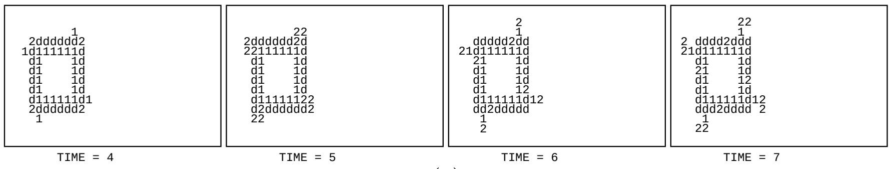
As for von Neumann's and Langton's automata, the ideal cellular space for our automaton is an infinite two-dimensional grid. Since we realize that a practical implementation of such a cellular space might prove difficult, we added some transition rules to handle the collision between the constructing arm and the border of the array. On meeting the border, the arm will retract without attempting to make a copy of the parent loop.

The cells of the array require five basic states and at least one data state (see Fig. 1.4 at time 0). State 0 is the "quiescent state" and is represented by a blank space in the figures. State 1 is the "sheath state", that is the state of the cells making up the sheath and the four gates. State 2 is the "activation state". The four cells in the loop directing the replication are in state 2, as are the messengers which will be used to command the constructing arm and the tip of the constructing arm itself for the first phase of construction, after which the tip of the arm will pass to state 3, the "construction state". State 3 will construct the sheath that will receive the copy of the loop, signal the parent loop that the sheath is ready, and lead the duplicated data to the new loop. State 4, the "destruction state", will destroy the constructing arm once the copy is ready. In addition to these states, we have labeled 'd' the data state, with the understanding that this one symbol might in fact represent any set of states not including states 0 to 4.

The initial configuration is in the form of a square loop wrapped around a sheath. The size of the loop is a variable that for our example has been set to 8×8. The loop is a sequence of data states in which four cells in the activation state are placed at a distance from each other equal to the side of the loop. Near the four corners of the loop we have placed four cells in the sheath state. These are the gate cells, and the position they occupy signifies that the gates are open (that is, that the automaton should attempt to duplicate itself in all four directions).

1.2.2 Operation

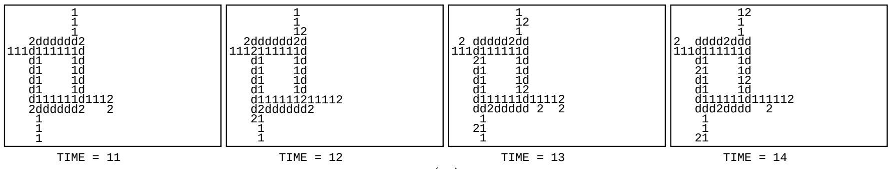
Once the cellular space starts operating, the data starts turning around the loop. Nothing happens until the first 2 reaches a corner, where it finds the gate open. Since the gate is open, the 2 splits into two identical cells. One cell continues turning around the loop, while the second starts extending the arm (Fig. 1.5a). The arm advances by one cell each two time periods. Once the arm has started extending, each 2 that arrives to a corner will again split and one of the copies (the "messenger") will start running along the arm, advancing by one cell per cycle (Fig. 1.5b). Since the arm is extending at half the speed of these messengers and the messengers are spaced 8 cells apart (the length of one side of the loop), the messengers will reach the tip of the arm at regular intervals corresponding to the length of one side of the loop.



```
         1                 22                  2                  22
   2dddddd2          2dddddd2d                 1                   1
  1d111111d          22111111d           ddddd2dd           2 dddd2ddd
   d1    1d           d1    1d          21d111111d          21d111111d
   d1    1d           d1    1d            21    1d            d1    1d
   d1    1d           d1    1d            d1    1d            21    1d
   d1    1d           d1    1d            d1    1d            d1    12
   d111111d1          d11111122           d1    12            d1    1d
   2dddddd2           d2dddddd2           d111111d12          d111111d12
    1                 22                  dd2ddddd            ddd2dddd 2
                                           1                   1
                                           2                  22

   TIME = 4           TIME = 5            TIME = 6            TIME = 7
```

(a)



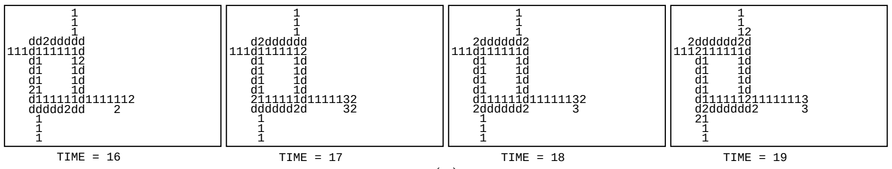
```
TIME = 11:
         1
         1
         1
   2dddddd2
111d111111d
   d1    1d
   d1    1d
   d1    1d
   d1    1d
   d111111d1112
   2dddddd2   2
    1
    1
    1

TIME = 12:
         1
         1
         12
  2dddddd2d
1112111111d
   d1    1d
   d1    1d
   d1    1d
   d1    1d
   d111111211112
   d2dddddd2
   21
    1
    1

TIME = 13:
         1
         12
         1
 2 ddddd2dd
111d111111d
   21    1d
   d1    1d
   d1    1d
   d1    12
   d111111d11112
   dd2ddddd 2  2
    1
   21
    1

TIME = 14:
         12
         1
         1
2  dddd2ddd
111d111111d
   d1    1d
   21    1d
   d1    12
   d1    1d
   d111111d111112
   ddd2dddd  2
    1
    1
   21
```

(b)



```
TIME = 16:
         1
         1
         1
   dd2ddddd
111d111111d
   d1    12
   d1    1d
   d1    1d
   21    1d
   d111111d1111112
   ddddd2dd    2
    1
    1
    1

TIME = 17:
         1
         1
         1
   d2dddddd
111d1111112
   d1    1d
   d1    1d
   d1    1d
   d1    1d
   2111111d1111132
   dddddd2d     32
    1
    1
    1

TIME = 18:
         1
         1
         1
   2dddddd2
111d111111d
   d1    1d
   d1    1d
   d1    1d
   d1    1d
   d111111d11111132
   2dddddd2      3
    1
    1
    1

TIME = 19:
         1
         1
         12
  2dddddd2d
1112111111d
   d1    1d
   d1    1d
   d1    1d
   d1    1d
   d111111211111113
   d2dddddd2      3
   21
    1
    1
```

(c)

Fig. 1.5. Cellular duplication. (a) The constructing arm starts extending. (b) The first messenger leaves the loop. (c) The first messenger reaches the tip of the constructing arm.

When the first messenger reaches the tip of the arm, the tip, which was until then in state 2, passes to state 3 and continues to advance at the same speed (Fig. 1.5c). This transformation tells the arm that it has reached the location of the offspring loop and to start constructing the new sheath.

The next two messengers will force the tip of the arm to turn left (Fig. 1.6), while the fourth will reach the tip as the arm is closing upon

```
         1
         1
         1
   d2dddddd
111d1111112
   d1    1d
   d1    1d
   d1    1d
   d1    1d
   2111111d11111111111113
   dddddd2d     2      23
    1
    1
    1
```
TIME = 31

```
         1
         1
         1
   2dddddd2
111d111111d
   d1    1d
   d1    1d
   d1    1d
   d1    1d
   d111111d111111111111133
   2dddddd2      2     23
    1
    1
    1
```
TIME = 32

```
         1
         1
         12
  2dddddd2d
1112111111d
   d1    1d
   d1    1d
   d1    1d
   d1    1d
   d1111112111111111111123
   d2dddddd2      2    23
   21
    1
    1
```
TIME = 33

```
         12
         1
         1
2  dddd2ddd
111d111111d
   d1    1d
   21    1d
   d1    12
   d1    1d            33
   d111111d1111111111111 1
   ddd2dddd  2      2
    1
    1
   21
```
TIME = 35

```
         1
         1
         1
   ddd2dddd
111d111111d
   d1    1d
   d1    12
   21    1d            3
   d1    1d            1
   d111111d1111111111111 1
   dddd2ddd   2      2
    1
    1
    1
```
TIME = 36

```
         1
         12
         1
 2 ddddd2dd
111d111111d
   21    1d
   d1    1d
   d1    1d
   d1    12             3
   d111111d111111111111331
   dd2ddddd 2      2   23
    1
   21
    1
```
TIME = 34

Fig. 1.6. The second messenger forces the arm to turn to the left.

```
         1
         1
         1             1
   ddd2dddd         2
111d111111d     1 111111
   d1    1d       1    1
   d1    12       1    12
   21    1d      21    1
   d1    1d       1    1
   d111111d1111111111111 1
   dddd2ddd   2      2
    1
    1
    1
```
TIME = 71

```
         1
         1
         1             1
   dd2ddddd        2
111d111111d     1 111111
   d1    12       1    12
   d1    1d       1    1
   d1    1d       1    1
   21    1d      33    1
   d111111d1111111111111 1
   ddddd2dd    2      2
    1
    1
    1
```
TIME = 72

```
         1
         1
         1             1
   d2dddddd       2
111d1111112     1 1111112
   d1    1d       1    1
   d1    1d       1    1
   d1    1d       1    1
   d1    1d       1    1
   2111111d1111113311111 1
   dddddd2d     2      2
    1
    1
    1
```
TIME = 73

```
         1
         1
         1             1
   2dddddd2      2      2
111d111111d     1 111111
   d1    1d       1    1
   d1    1d       1    1
   d1    1d       1    1
   d1    1d       1    1
   d111111d1111111311111 1
   2dddddd2      3      2
    1
    1
    1
```
TIME = 74

```
         1
         1
         12            1
  2dddddd2d            2
1112111111d     12111111
   d1    1d       1    1
   d1    1d       1    1
   d1    1d       1    1
   d1    1d       1    1
   d1111112111111111111121
   d2dddddd2    3 2
   21
    1
    1
```
TIME = 75

```
         1
         12
         1             1
 2 ddddd2dd           2
111d111111d     1 111111
   21    1d      21    1
   d1    1d       1    1
   d1    1d       1    1
   d1    12       1    12
   d111111d1111111111111 1
   dd2ddddd 2  3   2
    1
   21
    1
```
TIME = 76

Fig. 1.7. The arm closes the new loop.

itself (Fig. 1.7). It causes the sheath to close and then runs back along the arm to signal to the original loop that the new sheath is ready.

Once the return signal arrives at the corner of the original loop, it waits for the next 2 to arrive (Fig. 1.8). When the 2 sees the 3 waiting by the gate, again it splits, one copy staying around the loop, the other running along the arm. This time, however, rather than running along the arm in isolation as a messenger, it carries behind him a copy of the data in the loop.

Always followed by the data, the messenger runs around the sheath until it has reached the junction where the arm folded upon itself (Fig. 1.9). On reaching that spot, it closes the loop and sends a destruction signal (the 4)

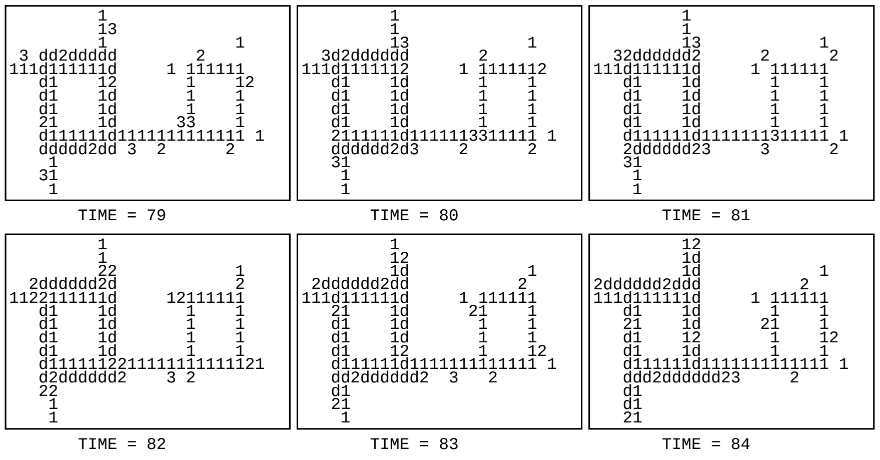


Fig. 1.8. The return signal starts the copy of the data.

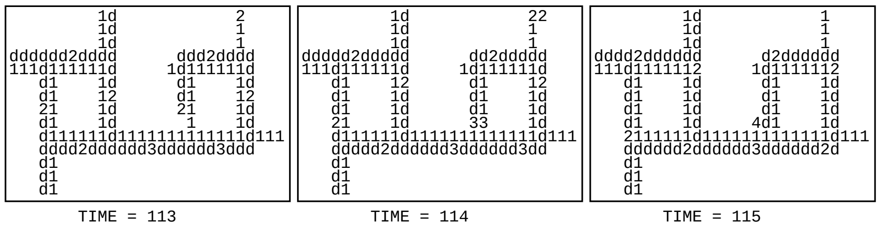


Fig. 1.9. The data warps around the new loop and the arm is destroyed.

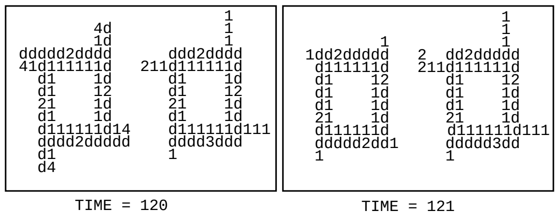


Fig. 1.10. The original cell becomes inactive.

back along the arm. The signal will destroy the arm until it reaches the corner of the original loop, where it closes the gate (Fig. 1.10).

Meanwhile, the new loop is already starting to replicate itself in three of the four directions. One direction (down in the figures) is not necessary since another of the new loops will always get there first, and therefore its corresponding gate will be set to the closed position.

After 121 time periods the gates of the original automaton will be closed and it will enter an inactive state, with the understanding that it will be ready to replicate itself again should the gates be opened.

1.2.3 Example

In Fig. 1.11, we illustrate an example of how the data states can be used to carry out operations alongside self-replication. The operation in question is the construction of three letters, LSL (the acronym of Logic Systems Laboratory), in the empty space inside the loop. Obviously this is not a very useful operation from a practical point of view, but it is a non-trivial case of construction that should demonstrate some of the capabilities of the automaton.

For this example, we have used 5 data states, which have brought the number of transition rules to 35202 (note that this figure is reached through an automatic generation of multiple rules, covering every possible combination of the 5 data states: far fewer rules are likely to be actually required). Of these, 326 are new rules which control the behavior of the program, and do not concern self-replication. The loop size is 20×20, and a full replication of a loop requires 321 time periods.

The operation of the program is fairly straightforward. When a certain "initiation sequence" within the loop arrives at the top left corner of the loop, a "door" is opened in the internal sheath. The rest of the program, as it passes by the door in its rotation around the loop, is duplicated and one of the copies enters the interior of the loop, where it is treated as a sequence of instructions which direct the construction of the three letters. The construction mechanism is somewhat similar to the method Langton used in his own loop, with single-cells instructions such as "turn left", "advance", etc. The construction ends when a "termination sequence" arrives at the door. At that stage, the door is closed and a flag is set in the sheath to warn that the program has already executed.

During the process of replication, the program is simply copied (as opposed to interpreted as in the interior of the cell) and arrives intact in the new loop, where it will execute again exactly as it did in the parent loop.

This is a simple demonstration of one way in which the data in the loop could be used as an executable program. However, it should be noted that, in the above example, the automaton executes a program devoid of any external inputs, and thus relatively uninteresting from a computational point of view. In the next sections we will illustrate another, more complex but also much more useful, example of how the automaton's data states can be used to store and execute programs capable of computing arithmetic functions.

1.3 Collision-Based Computing: Theoretical Notions

To show that it is indeed possible to perform computationally-useful tasks using self-replicating automata, we wished to adapt our programmable automaton to execute some arithmetical operations, notably addition and multiplication, on binary numbers. In order to implement these features, we opted for a slightly modified version of the particle model developed by Steiglitz et al. [8], briefly described in this section. We selected this particular model because it is a model designed to operate within the cellular automaton 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.

1.3.1 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 Fig 1.12.

To effect the sum, the two bits are stored in two cells, that imitate signals 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 represents the value 1 and the right one the value 0. Following the collision the sum is made and the new right-moving cell represent the value 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 signal representing a bit (one or zero). The two cells move towards each other. A processor cell is placed between the two sequences of signals (Fig. 1.13). In each number, the least-significant bit leads the sequence, so that when the two numbers collide head-on at the processor cell, this can add the bits in order of increasing significance.

After a collision, the two incoming cells are destroyed and 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 "result" cell, the processor stores the value of the carry bit, which it will use to compute the result of the next collision between the bits of the two operands.

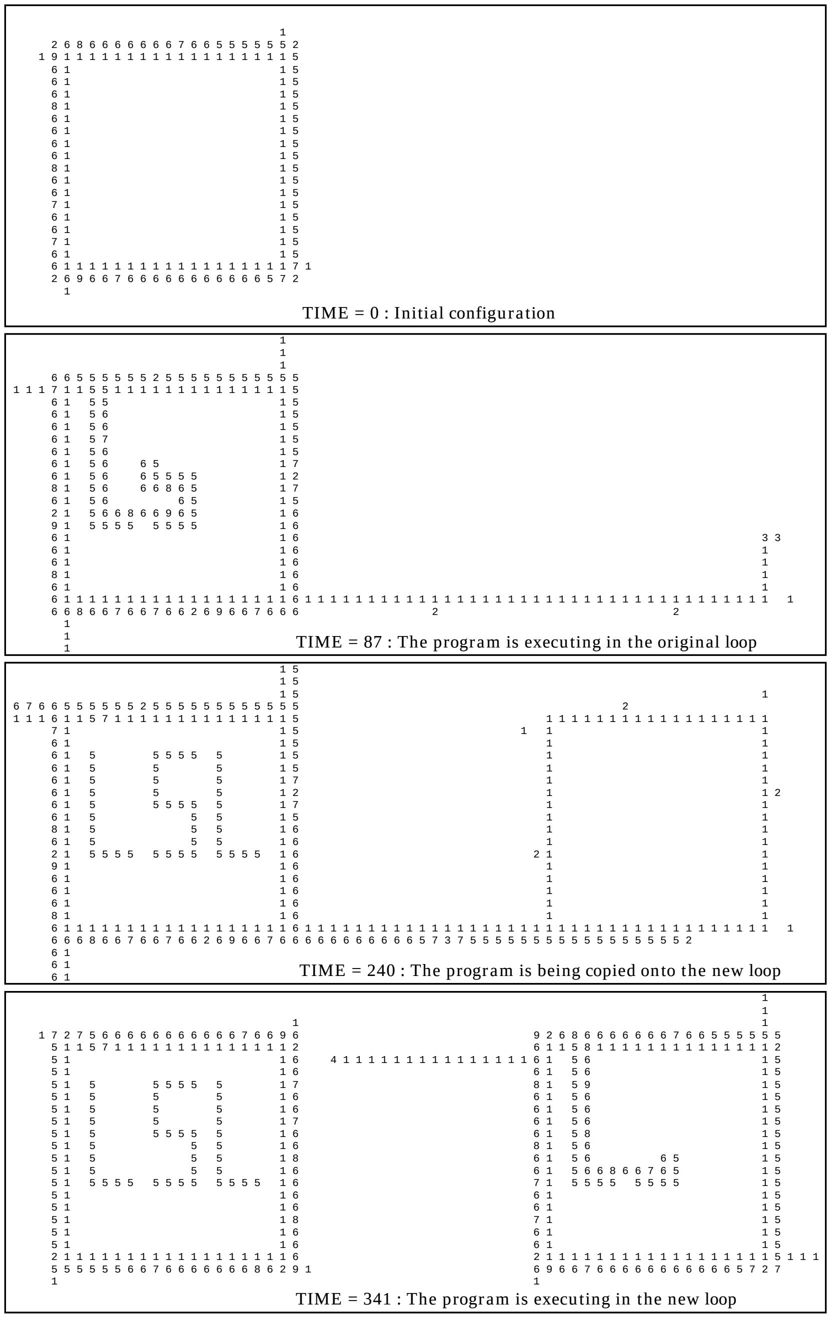

TIME = 0 : Initial configuration
TIME = 87 : The program is executing in the original loop
TIME = 240 : The program is being copied onto the new loop
TIME = 341 : The program is executing in the new loop

Fig. 1.11. An example of the capabilities of our automaton.

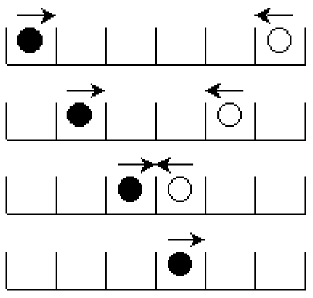


Fig. 1.12. Sum of two one-bit numbers.

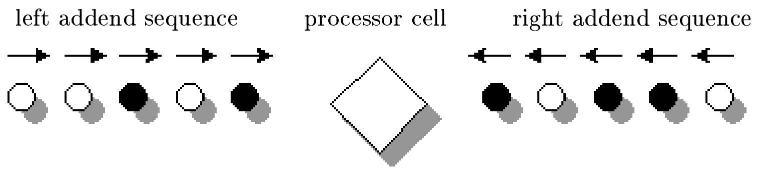


Fig. 1.13. Two data streams collide in one processor cell.

1.3.2 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 twice the number of bits of the multiplicands. Figure 1.14 shows the starting configuration of the multiplication.

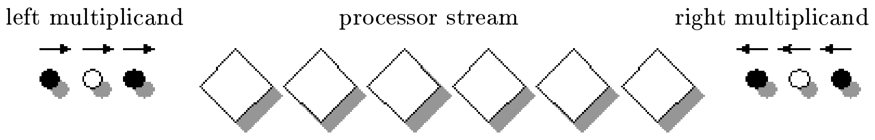
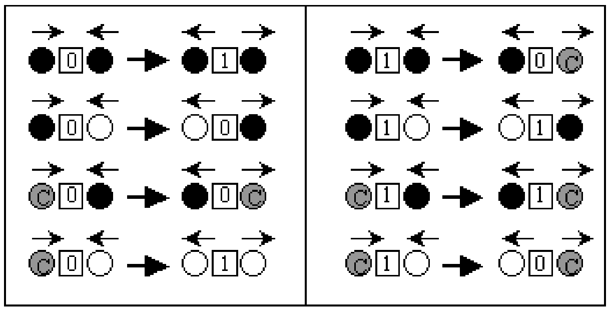


Fig. 1.14. Two data sequences collide in a processor stream.

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 effect 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 1.1. 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 1.15 shows an example of the multiplication of two 2-bit numbers.

In Fig. 1.15, each row represents the state of the multiplication at the time t. In this example, the processor cells can have three different states. At

Table 1.1. Rules for the processor cells.



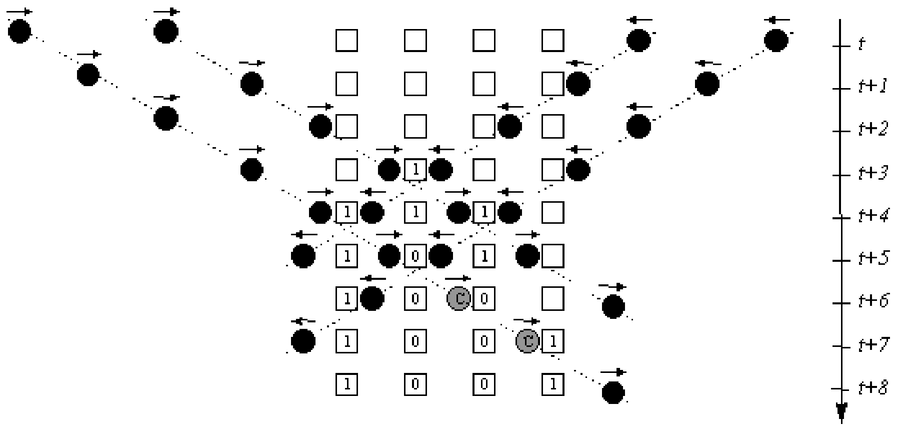


Fig. 1.15. 3×3 binary multiplication.

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 × 11 = 1001, that is, in decimal notation, 3 × 3 = 9.

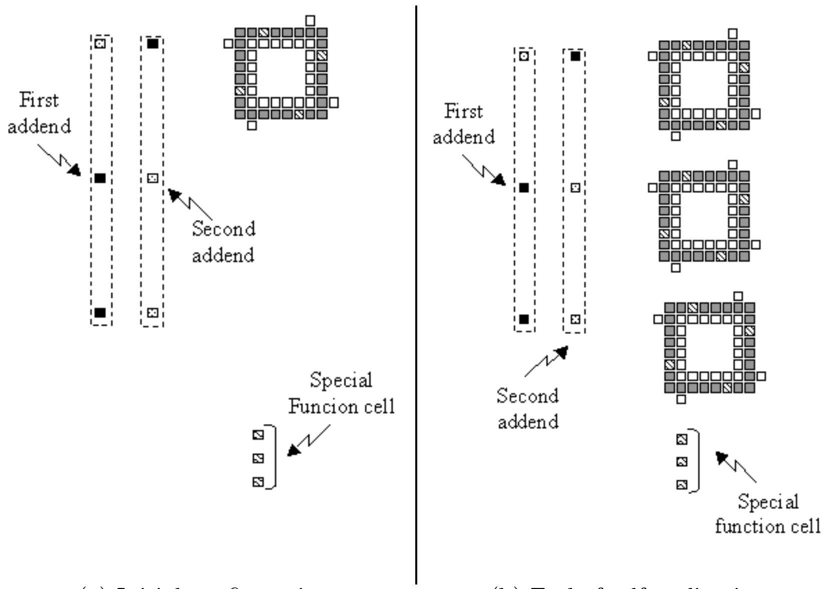
1.4 Implementation on Self-Replicating Loops

In this section, we will show how Steiglitz's model was implemented using the support of our self-replicating loops. As we will see, 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.

1.4.1 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. 1.16a) 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. 1.16b). 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

Fig. 1.16. 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. 1.17a), 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. 1.17b), which will propagate it to the next automaton (Fig. 1.17c).

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.

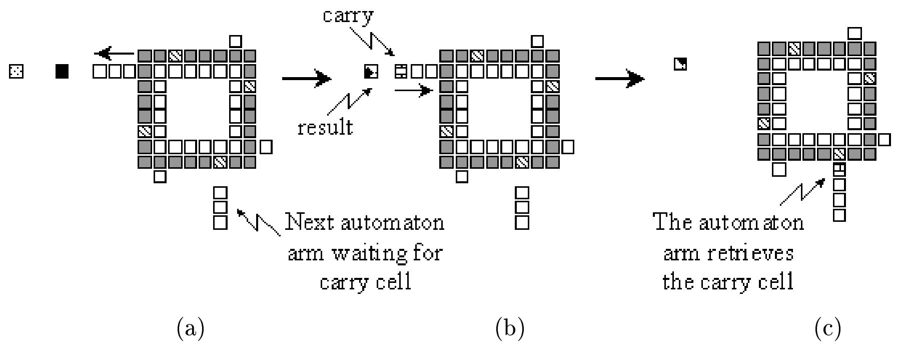

carry
result
Next automaton arm waiting for carry cell
The automaton arm retrieves the carry cell
(a)
(b)
(c)

Fig. 1.17. Computation of a collision.

1.4.2 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. 1.18).

The multiplication 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

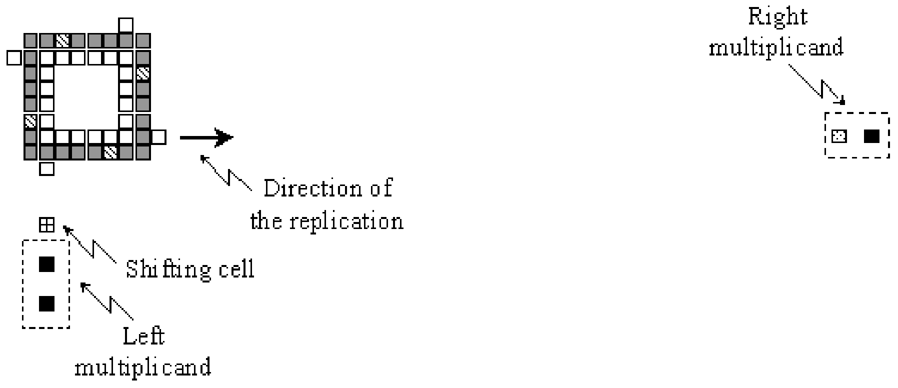


Fig. 1.18. Starting point of multiplication.

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. 1.19).

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 Tab. 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 be 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.

1.4.3 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:

$$(A + B + ...) * (a + b + ...) \tag{1.1}$$

In order to compute this kind of function, we need a starting configuration similar to Fig. 1.20, which expands to the machine shown in Fig. 1.21

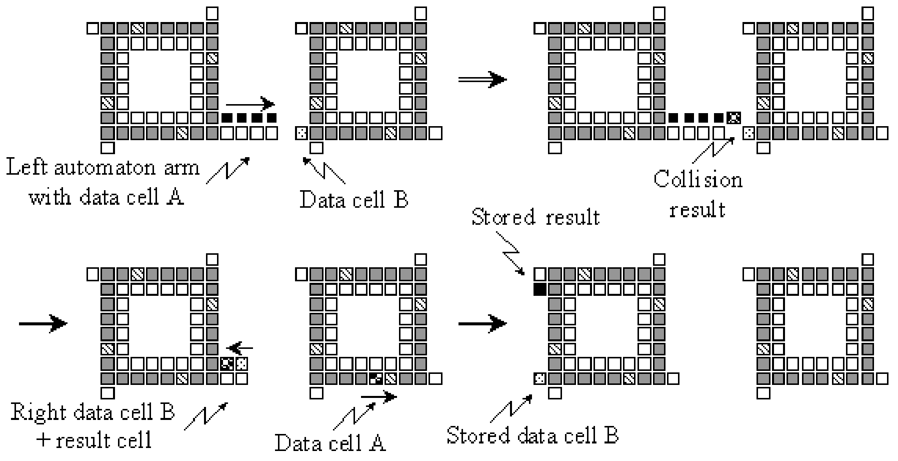


Fig. 1.19. Collision between two data cells.

after self-replication. 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. 1.22).

The use of self-replicating loops as a support for this kind of computation simplified considerably the design of our machines, and the same approach could be applied to the development of further mathematical functions and further combinations of such functions.

This development, however, is not as simple as it should be: the cellular automata environment, even with the support of the self-replicating loops, sorely lacks the tools that would be necessary to automate the design of these kind of complex machines.

1.5 Conclusion

The goal of the work presented in this chapter was to show that it 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. [8]. 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

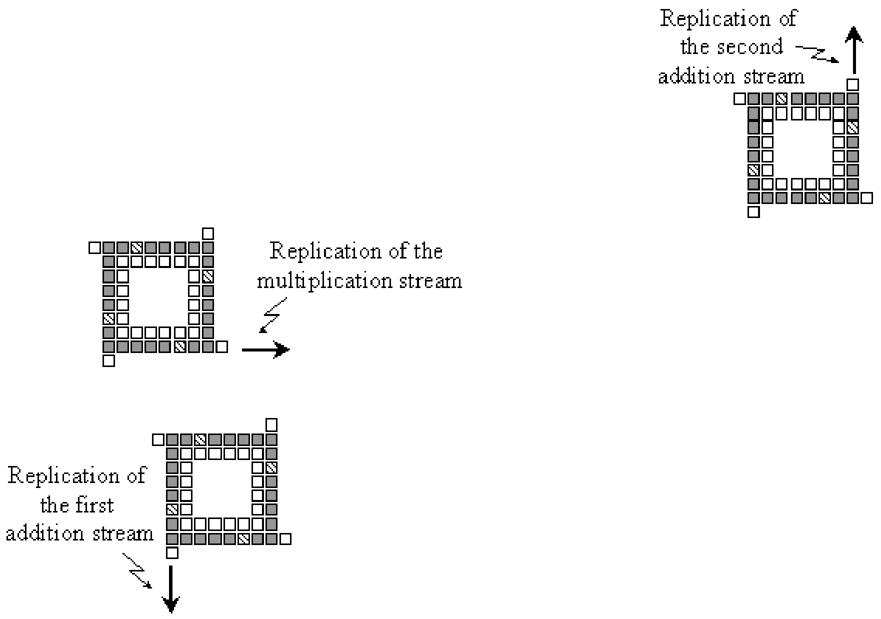

Replication of the second addition stream
Replication of the multiplication stream
Replication of the first addition stream

Fig. 1.20. Initial configuration.

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 these 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:

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

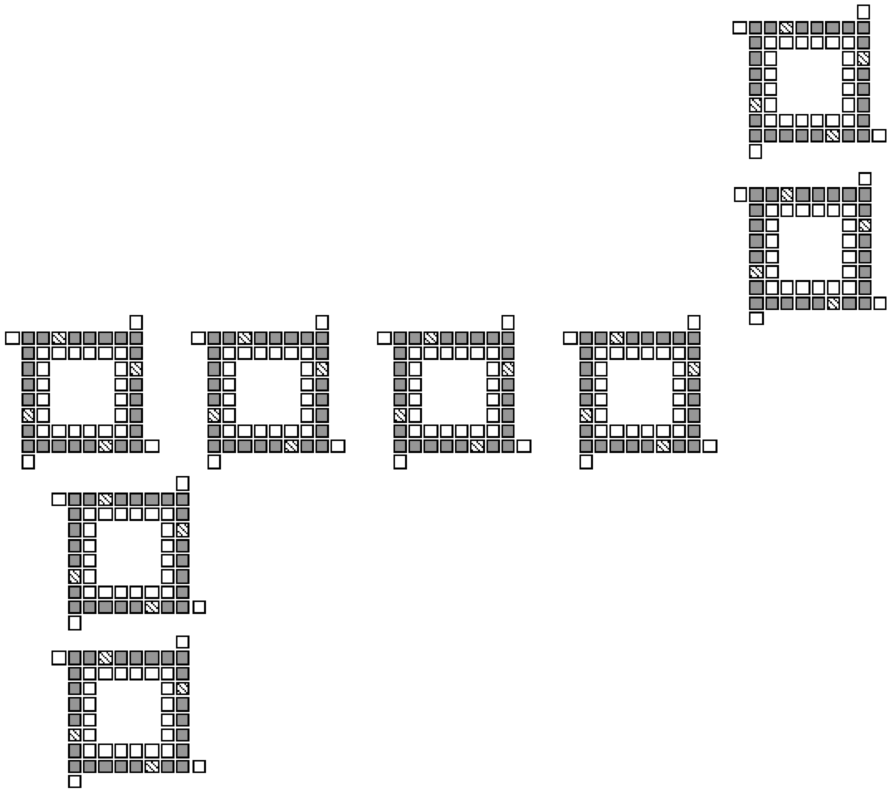


Fig. 1.21. End of self-replication.

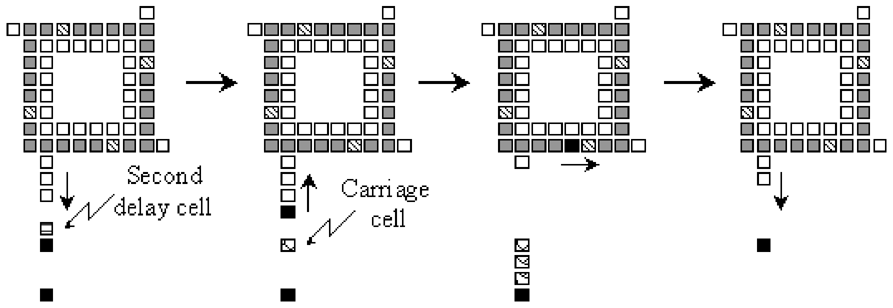


Fig. 1.22. Operation of the carriage cells.

- The execution units need not be very powerful, as complex operations can be performed by many small identical units (the fundamental principle of parallelism).

- 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).
- 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.

Acknowledgments

This work was partially supported by the Swiss National Foundation under grant 21-54113.98 and by the Leenards Foundation (Switzerland).

References

1. Byl J. Self-reproduction in small cellular automata *Physica D* **34** (1989) 295–299.
2. Codd E.F. *Cellular Automata* (Academic Press, New York, 1968).
3. Langton C.G. Self-reproduction in cellular automata. *Physica D* **10** (1984) 135–144.
4. Mange D., Sipper M., Stauffer A. and Tempesti G. Toward robust integrated circuits: the embryonics approach *Proc. IEEE* **88** (2000) 516–541.
5. Mange D. and Tomassini M., Editors *Bio-inspired Computing Machines: Towards Novel Computational Architectures* (Presses Polytechniques et Universitaires Romandes, Lausanne, 1998).
6. Petraglio E., Henry J.-M. and Tempesti G. Arithmetic operations on self-replicating cellular automata *Lecture Notes in Artificial Intelligence* **1674** 447–456.
7. Reggia J.A., Armentrout S.L., Chou H.H. and Peng Y. Simple systems that exhibit self-directed replication *Science* **259** (1993) 1282–1287.
8. Steiglitz K., Squier R.K. and Jakubowski M.H. Programmable parallel arithmetic in cellular automata using a particle model *Complex Systems* **8** (1994) 311–323.
9. Tempesti G. A new self-reproducing cellular automaton capable of construction and computation *Lecture Notes in Artificial Intelligence* **929** (1995) 555–563.
10. Tempesti G., Mange D. and Stauffer A. Self-replicating and self-repairing multicellular automata *Artificial Life* **4** (1998) 259–282.
11. von Neumann J. *The Theory of Self-Reproducing Automata* (University of Illinois Press, Urbana, 1966).