Silicon or Molecules?
What's the Best for Splicing?

Gheorghe Ştefan
Technical Univ. of Bucharest, Dept. of Electronics
stefan@agni.arh.pub.ro

Abstract

We present in this paper the main ideas concerning the implementation in the solid state circuits of some *molecular mechanism*: the *splicing* operation and *insert/delete* operation. The physical support for these operations is based on the *Connex Memory* concept first introduced in [Ştefan '85(91)]. We promote this solution because a pure biological process is very hard to be interfaced with machines in nowadays technologies. In the same time we believe that the mechanisms emphasized in the molecular process of computation are very good suggestions for silicon based machines devoted to perform a fine grain parallelism. Using a Connex Memory the splicing operation or the insert/delete operation is performed in linear time related to the length of the rules; the time does not depend on the length of the processed strings. In order to perform in **parallel** all possible applications of a rule in a set of strings, the function of the Connex Memory is extended over a *cellular automaton*, thus defining the *Eco-Chip*.

1 Introduction

The molecular computing is a very exciting theoretical approach from the '70s. M. Conrad is a pioneer in this domain [Conrad '85]. T. Head emphasizes a basic operation named *splicing* [Head '87,'92] that can be used as model for some aspects of molecular processes involved in computation. At the end of 1994, L. M. Adleman performed his successful experiment of solving the Hamiltonian problem in a graph by manipulating DNA sequences [Adleman '94]. All scientists believe that these approaches will generate maybe in short time "biological computing machines".

In the same time these results *suggest* us two new directions in computer science research.

- The first is to investigate the possibility of using the main molecular operations as basic generative rules in the formal language theory and in the theory of computation. In the last few years, the formal language theory enriches by new formal systems devoted for modeling the typical mechanisms of molecular computing.

- The second is to implement efficiently these operations using an appropriate hardware support. The degree of parallelism is very high in a system that uses these new strange operations. We believe that an adequate "smart memory" device can give full value of using these strange "molecular rules" in computation.

Starting from the first direction, opened by papers like [Păun '95b, '96, '97], our approach is related with the second by the aim of putting together the operations suggested by the molecular computing and the Connex Memory (CM) concept or the Eco-Chip (EC) model.

The *first step* is to prove that a system based upon the CM circuit performs these operations in a time related with the size of the rules instead of the standard system with RAM that performs the same operations in a time related to the storage space used for all the strings involved.

The *second step* uses the CM concept to build the EC as a cellular automaton. On this new circuit the insert and delete functions of the CM can be performed in many points in each clock cycle, rather than on the CM circuit that allows only one insert or delete function per clock cycle. Using the EC circuit, the similarity, between molecular computing and the computation performed on this silicon machine, becomes maximal because many operations can be parallel performed.

2 DNA Based Computing Mechanisms

Gh. Păun emphasized [Păun '97] three distinct basic operations on DNA (and RNA) that can be used for molecular computation:

- the *matching*, the basic operation in **P-systems**, introduced in [Kari '96a] starting from the famous Adleman's experiment

- the *splicing* operation, defined by Tom Head, is the main mechanism in **H-systems**

- the *insertion/deletion* operation [Kari '96b] can be used to define generative mechanisms in **I-systems**.

All the three operations lead toward universal computability models, which are equivalent with the Turing Machine. For our purposes, it is enough to present only the second and the third operations because, from the point of view of the CM concept applications, the first two operations are very similar.

2.1 The Splicing Mechanism and H-Systems

A formal definition of the splicing operation is to be found in [Păun '96]. Following this definition, we specify one of the basic mechanism from DNA computing that can be borrowed for a silicon based machines.

Definition 1 *Let be the finite alphabet V and two special symbols $\#$ and $\$$ not in V. A* splicing rule *over V is specified by $(x, y) \vdash_r z$ where,*
$r = u_1\#u_2\$u_3\#u_4$ with $x, y, z, u_i \in V^*$ and acts as follows:

$$\begin{array}{ll} \textit{if} & x = x_1u_1u_2x_2,\ y = y_1u_3u_4y_2 \\ \textit{then} & z = x_1u_1u_4y_2 \end{array}$$

for some $x_i, y_i \in V^*$. $\diamond$

Starting from two strings x and y a new string z is obtained. The sites of the splicing are the places defined by the substrings u_1u_2 and u_3u_4. A machine must find the sites of the splicing, cut the two strings x and y in the sites of the splicing and concatenate the left part of x with the right part of y. The rests of x and y are added to V^* and will be considered again in the next splicing.

Definition 2 *A* H-system *is a pair $\sigma = (V, R)$ where V is an alphabet and $R \subseteq V^*\#V^*\$V^*\#V^*$ is a set of splicing rules.*$\diamond$

For each rule r a solid-state machine can be imagined. In the same time each H-system σ has an associated machine.

Definition 3 *The* splicing machine *$SM = (V, a_r)$ has a* random accessed memory (RAM) *containing $S \subset V^*$ and a* finite automaton *that performs the rule* r *on S.* $\diamond$

Definition 4 *A* H machine *is defined by $HM = (V, A_R)$ where $S \subset V^*$ is stored in a* memory *and A_R is a set of automata each performing $r_i \in R$ on the same S.*$\diamond$

Definition 5 *The* Universal Splicing Machine *is*
$USM = (V, a)$ in which the memory *contains both $S \subset V^*$ and R, the automaton* a *having the role of applying the rules from R on S.*$\diamond$

Our line of thought follows the conclusion of Păun & Salomaa: "The splicing operation, essentially different from other language-theoretic operations, turns out to be surprisingly powerful. Easy characterizations of recursively enumerable languages (exhibiting Turing machine competence) are obtained in this framework. This, on the one hand, proves again the complexity of the DNA structure and the power of the mechanisms manipulating it, on the other hand, suggests that universal "computers" can be constructed on this basis."

2.2 The Insert/Delete Mechanism and I-Systems

Another main molecular mechanism that has a simple mathematical model refers to the local *mutations* occurred in a DNA or RNA string. A mutation can be assimilated to an insertion or a deletion. In same places defined by specific substrings, a new substring can be inserted or a substring can be deleted. Two distinct set of rules, one for *insert* and another for *delete*, characterize a class of systems called *I-systems* [Păun '97].

Definition 6 *An* I-System *is a construct:*

$$\gamma = (V, T, A, I, D)$$

where: V is a finite alphabet, $T \subseteq V$ is the terminal alphabet, $A \subset V^*$ is the finite set of the axioms, $I, D \subset V^* \times V^* \times V^*$ are finite subsets of the insertion *rules and of the* deletion *rules having the form (u, z, v) and acting as follows:*

1. *$x = x_1uvx_2$ becomes $y = x_1uzvx_2$, for $x_1, x_2 \in V^*$ and $(u, z, v) \in I$, thus performing an* insertion
2. *$x = x_1uzvx_2$ becomes $y = x_1uvx_2$, for $x_1, x_2 \in V^*$ and $(u, z, v) \in D$, thus performing a* deletion*,*

so as for $x, y \in V^*$ we can write $x \Rightarrow y$. $\diamond$

Let be $\Rightarrow^*$ the reflexive and transitive closure of $\Rightarrow$.

Definition 7 *The language generated by the* I-system γ *is*

$$L(\gamma) = \{w \in T | x \Rightarrow^* w, for x \in A\}. \diamond$$

The basic action in a *I-system* is to find all the places where the substrings uv or uxv are located. After that, the substring x is inserted between the substrings u and v or is deleted from among the substrings u and v.

3 The Connex Memory

3.1 The Definition

The CM is a physical support for a string in which we can find any substring, identifying, in such a manner, any place for *reading*, *inserting* or *deleting* a symbol or a substring. The CM is *a sort of CAM* (Content Addressable Memory), structured as a *bi-directional shift register* in which a significant point is *marked*, as a consequence of an *associative sequential mechanism* used to find a *name* in a number of steps equal to the length of the name.

Definition 8 *The* connex memory *CM is a physical support of a string of variables (see Figure 1) having* values *from a finite set of symbols and two* states*:* non-marked *or* marked*, over which we can apply the following set of functions (CM's commands):*
CMCOM = {RESET s, FIND s, CFIND s, INSERT s, READup, READdown, READ, DELETE} *where:*

- *RESET s : all the variables take the value s*
- *FIND s : all the variables that follow a variable having the value s switch to the marked state and the rest switch to the non-marked state*

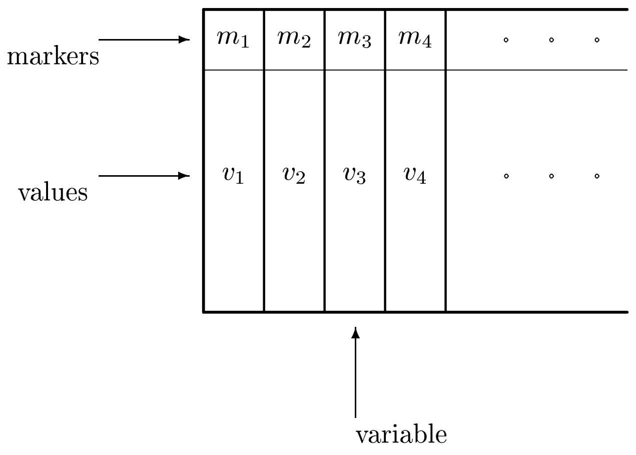

Figure 1: The content of the Connex Memory. Each cell contains the value v_i of a variable and the state bit, m_i, named marker.

- *$CFIND\,s$: (conditioned find) all the variables that follow a variable having the value s and being in the marked state switch in the marked state and the rest switch in the non-marked state*
- *$INSERT\,s$: the value s is inserted before the first marked variable*
- *$READ\,up \mid down \mid -$: the output has the value of the first marked variable and the marker moves one position to right (*up*) or to left (*down*) or remains unchanged (-)*
- *$DELETE$: the value stored in the first marked position is deleted, the position remains marked (the output has the value of the first marked variable) and the symbols from the right are moved one position left*

All these functions are executed in time $O(1)$ (one clock cycle). $\diamond$

3.2 How Does Each Function Work?

We shall answer to the question: *how does each function work?* giving a set of examples in which: $S(t)$ is the string stored in the memory in the current clock cycle when a certain function is applied, $OUTPUT(t)$ is the value of the output of the memory in the current cycle, $S(t+1)$ is the content of the memory as a result of the function applied in the previous clock cycle. The marked variables are bolded.

RESET p
S(t) = roivndkgotrun...
S(t+1) = ppppp....p...

FIND b
S(t) = ...(bubu(big brother's gun))...
S(t+1) = ...(b**u**b**u**(b**i**g b**r**other's gun))...

CFIND u
S(t) = ...(b**u**b**u** (b**i**g b**r**other's gun))...
S(t+1) = ...(bu**bu(**big brother's gun))...

INSERT c
S(t) = ...(bu**bu(**big brother's gun))...
S(t+1) = ...(buc**bu(**big brothe's gun))...

READ
S(t) = ...(bu**bu(**big brother's gun))...
OUTPUT(t) = b
S(t+1) = ...(bu**bu(**big brother's gun))...

READ up
S(t) = ...(bu**bu(**big brother's gun))...
OUTPUT(t) = b
S(t+1) = ...(bub**u(**big brother's gun))...

READ down
S(t) = ...(bu**bu(**big brother's gun))...
OUTPUT(t) = b
S(t+1) = ...(b**u**bu**(**big brother's gun))...

DELETE
S(t) = ...(bu**bu(**big brother's gun))...
OUTPUT(t) = b
S(t) = ...(bu**u(**big brother's gun))...

3.3 The Application Domains of the Connex Memory

The main domain of applications for CM based architecture is the *string oriented symbolic processing.* Some of them were presented in other papers and some represent working in progres.

1. The paper [Ştefan '96] is an exercise of using CM for implementing a *Lisp Oriented Machine.*

2. In [Ştefan '97] and in the present paper is offered a solution for a silicon-based machine that performs efficiently the *splicing mechanism* emphasized in molecular computing.

3. In the present paper we describe a very efficient system for the *insert/delete* mechanism, also characteristic for molecular computing.

4. The work [Ştefan '95] presents an application of the CM concept in implementing *eco (grammar) systems* [Csuhaj-Varú '93].

5. Another domain is the implementation of the unification mechanism in the Prolog language.

6. Expanding the CM functions over the cells of a cellular automaton the Lindenmayer grammars [Lindenmayer '68] gain a very good physical support for their parallel-executed substitutions.

7. We have in progress a work in which we will present applications of CM in implementing *Markov rewriting systems* [Markov '54].

4 Molecular Mechanisms on Connex Memory

In order to explain how CM works in finding strings of symbols having different length it is useful to expand the function $FIND\ s$ to the macro-function $SFIND\ S$ (string find), where: $S = s_1 s_2 \dots s_n$ is a string of symbols. The sequence of commands that emphasizes the end of all occurrences of the S string, marking the variables that immediately follow the last symbol s_n, is:

$$SFIND\ S = FIND\ s_1, CFIND\ s_2, CFIND\ s_3, \dots, CFIND\ s_n.$$

Thus, all occurrences of the string S in CM are found and marked in time $T_{SFIND\ S} \in O(n)$, i.e., *the searching space dimension does not matter.* In order to find the string S, having the length $l(S) = n$, *it is enough to waste only the time nedeed to utter it.*

4.1 Implementing Splicing with CM

For explaining the efficiency of the CM in performing the splicing operations we use a SM only. The expansion towards HM or USM is obvious and the efficiency in performing the splicing operation is not affected.

If the splicing rule is:

$$r = u_1 \# u_2 \# u_3 \# u_4$$

and the initial content of the CM is:

$$\dots \$ x_1 u_1 u_2 x_2 \$ \dots \$ y_1 u_3 u_4 y_2 \$ \dots$$

(\$ is used for delimiting the strings in the memory), then an intermediate form is:

$$\dots \$ x_1 u_1 \& u_2 x_2 \$ \dots \$ y_1 u_3 u_4 y_2 \$ \dots$$

after identifying the first cutting point (emphasized with the special symbol & inserted between the substrings u_1 and u_2) and the final content of the CM is:

$$\ldots \$ x_1 u_1 u_4 y_2 \$ u_2 x_2 \$ \ldots \$ y_1 u_3 \$ \ldots$$

The next procedure describes this mechanism.

```
Prprocedure SPLICING
   if the string $u_1u_2$ is found [step 1]
      then back before $u_2$ [step 2]
           insert & before $u_2$ [step 3]
           if the string $u_3u_4$ is found [step 4]
              then back before $u_4$ [step 5]
                   move $u_4y_2$ before & [step 6]
                   substitute & with $ [step 7]
              else delete &
           endif
   endif
end SPLICING
```

Definition 9 *A splicing machine with CM is $SMCM = (V, a)$, see Figure 2, where CM contains a set of strings $S \subset V^*$, FA is a finite automaton that executes the procedure* **SPLICING**. *On the* Input tape *is stored the splicing rule. The current symbols accessed on the input tape and/or from the CM, together with the current state of the automaton generate, in each clock cycle, the next state of the automaton. In each state the automaton can move, if needed, left or right the input tape head, or commands CM using one of their functions.* $\diamond$

Indeed, executing the macro-function *SFIND S* the first marked variable in the CM is the variable after the first occurrence of the string S. The time for finding the string S in CM is in $O(l(S))$. More precisely, $l(S)$ is the number of clock cycles needed to find S. Using this "macro" the behavior of the finite automaton associated with the procedure SPLICING becomes clear and the execution time for each main step is:

step 1 : $T \in O(l(u_1) + l(u_2))$

step 2 : $T \in O(l(u_2))$

step 3 : $T \in O(1)$

step 4 : $T \in O(l(u_3) + l(u_4))$

step 5 : $T \in O(l(u_2))$

step 6 : $T \in O(l(u_4) + l(y_2))$

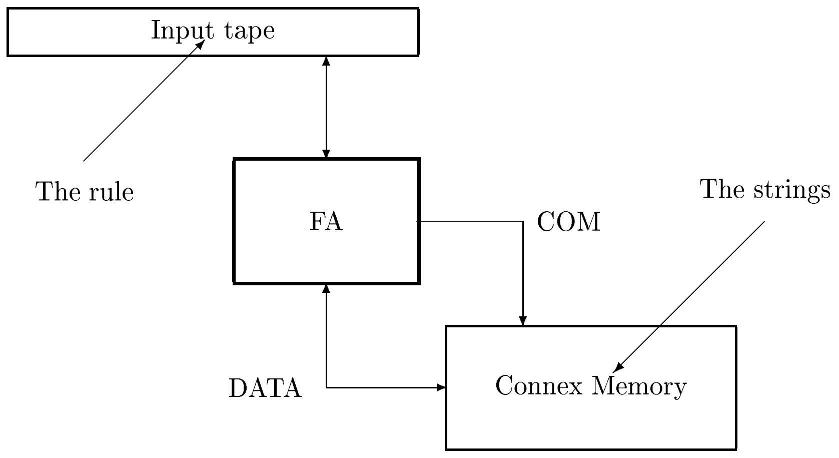

Figure 2: The splicing machine with CM

step 7 : $T \in O(1)$

Comparing SM and the $SMCM$ we observe that the *size* for both is in the same order: $O(\Sigma l(S_i))$ where $\Sigma l(S_i)$ is the total length of the strings $S_i \in S$. But, there is a significant difference between the execution time for the splicing operations in the two approaches. In the first machine, which uses a standard RAM memory, we have $T_{SM} \in O(f(\Sigma l(S_i)))$ but the second solution has a better performance: $T_{SMCM} \in O(max(l(S_i)))$.

Although the actual size of CM is bigger than the size of the RAM memory (around 20 times), the gain in the speed justifies using CM because the time is proportional only with the biggest string S_i stored in the memory instead of the sum of the length of all strings from the memory.

4.2 Implementing Insert/Delete Operations with CM

Starting from Definition 6, the *Insert/Delete* operations are described by the following procedures.

```
Prprocedure INSERT
  if the string $uv$ is found (applying $SFIND\ uv$) [step 1]
    then back before $v$ [step 2]
         insert the string $z$ [step 3]
  endif
end INSERT
```

```
Prprocedure DELETE
   if the string $uzv$ is found (applying $SFIND\ uzv$) [step 1]
      then back before $z$ [step 2]
            delete the string $z$ [step 3]
   endif
end DELETE
```

The execution time for both procedures in a system build with CM is

$$T_{I/D} \in O(l(uzv))$$

because:

step 1 is executed in $T_{SFIND\ uv/uzv} \in O(l(uv))/O(l(uzv))$

step 2 is executed in $T_{BACK\ v/zv} \in O(l(v))/O(l(zv))$ or in $O(1)$ if a simple trick is used (inserting a special symbol after the string u)

step 3 is executed in $T_{i/d} \in O(l(z))$.

It is obvious that the machine that executes *Insert/Delete* operations has the same structure as the machine for the splicing mechanism (see Figure 2).

4.3 Limits to Be Removed

In the previous two approaches, for the *splicing* operation and for *insert/delete* operations, the rule is performed with a partial efficiency only, because the operation can be completed in only one place. Indeed, the places where the rule must be applied are all identified in parallel, but the effective concatenation or effective insert/delete is performed in only one place because the structure of CM that consists in a sort of the shift register. For example, an insert in two distinct points in the string stored in the CM implies that a part of the internal register of CM is shifted one position and another part is shifted with two positions.

5 A Cellular Automaton as Support for Connex Memory Functions

5.1 Definition

In order to add the possibility **to access all marked points** of the stored string, we adopt a two-dimensional support for CM. Instead of the one-dimensional structure of a shift register we use a *two-dimensional CA*. The string length can be now modified synchronously in many points by insert or delete. This new feature is enabled by the liberty of adding more new symbols in any places on the two-dimensional area of a CA.

Each symbol of the string is stored in the state of a cell and the link is done by the adjacency in the CA area. Because in our CA each cell has eight neighbors, it is very easy to add a new adjacency in the string.

The CA consists of *active cells* and of *inactive cells*. The first contain the string and are linked by three-bit pointers in the eight cells neighborhood. In each cell two processes are performed:

- the process of executing operations implied by the CM function:
  - the subset that does not modify the length of the string, only in the active cells
  - the subset that modifies the length of the string
- the self-organizing process of cells toward the state of each active cell to be surrounded by a maximum number (i.e., 6 if possible) of inactive cells.

The self-organizing process generates the conditions in which all the time we have enough space to insert in many places synchronously the same symbol. The same process removes, step by step in a sequential way, the inactivated cells.

5.2 An Algorithm for the Multiple Access CM

Specific for the Multiple Access CM (MACM) are three types of actions:

1. the insert function from the function set of CM
2. the delete function from the function set of CM
3. the self-organizing process offering space for new insert actions.

In this paper, only the case of dispersed marked points is studied.

5.2.1 $INSERT\, s$ in MACM

There are two typical situations in which a new symbol is inserted in a stored string. In Figure 3a, the marked cell (containing the arrow) has as the next cell the pointed cell (containing a circle). The second situation is presented in Figure 3b. These two configurations can be rotated three times for obtaining all the cases.

In the first case the next marked positions can be the cells labeled by a, b, c and d. In Table 1 is presented the algorithm for selecting the new active cell.

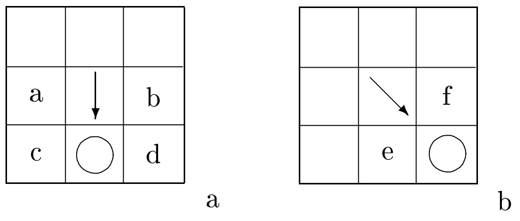

Figure 3: Insert configurations

a b c d	The cell used for expanding the string
0 0 0	randomly *a* or *b*
0 0 0 1	*a*
0 0 1 0	*b*
0 0 1 1	**if** *c* and *d* do not point to the marked symbol (the arrow) **then** randomly *a* or *b* *deadlock*; **if** *c* points the marked symbol **then** *b*; **if** *d* points the marked symbol **then** *a*; **if** *c* and *d* point the marked symbol **then** *deadlock*
0 1 0 0	*a*
0 1 0 1	*a*
0 1 1 0	**if** *c* does not point the marked symbol **then** *a*, **else** *d*
0 1 1 1	**if** *c* does not point the marked symbol **then** *a*, **else** *deadlock*
1 0 0 0	*b*
1 0 0 1	**if** *d* does not point the marked symbol **then** *b*, **else** *c*
1 0 1 0	*b*
1 0 1 1	**if** *d* does not point the marked symbol **then** *b*, **else** *deadlock*
1 1 0 0	**if** *a* and *b* are not connected with the target of the marked symbol (the circle), **then** randomly *c* or *d*; **if** *a* and *b* are connected with "circle", **then** *deadlock*; **if** *a* is not connected with "circle", **then** *c*; **if** *b* is not connected with "circle", **then** *d*
1 1 0 1	**if** *a* is not connected with "circle", **then** *c*, **else** *deadlock*
1 1 1 0	**if** *b* is not connected with "circle", **then** *d*, **else** *deadlock*
1 1 1 1	*deadlock*

Table 1. The algorithm for finding the new active cell when the marked cell points horizontal or vertical.

In the second case the next marked position can be selected between the positions e and f. The corresponding algorithm is presented in the next table.

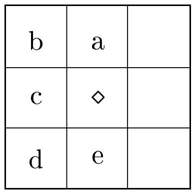

Figure 4: The reference cells for DELETE

e f	Action
0 0	random *e* or *f*
0 1	*e*
1 0	*f*
1 1	*deadlock*

Table 2. The algorithm for finding the new active cell when the marked cell points diagonal.

Deadlock situations are solved as a consequence of the self-organizing process below described.

5.2.2 *DELETE* in MACM

The CM's function DELETE is performed in two steps:

1. inactivate the content of the marked cell but maintains the cell connected in the string as an *empty cell*
2. eliminate the empty cell by shifting them:
   - until the string form allows to *point over*, inactivating the empty cell
   - or until the end of the string is reached.

The *point over algorithm* is the main problem. In Figure 4 we present the typical situations (all the rest is reducible to rotating this representation). The symbol (◇) represents the empty cell.

1. $a \to \diamond \to d, e$, no points over
2. $b \to \diamond \to d, e$, no points over
3. $c \to \diamond \to d \Rightarrow c \to d$, points over
4. $c \to \diamond \to e \Rightarrow c \to e$, points over

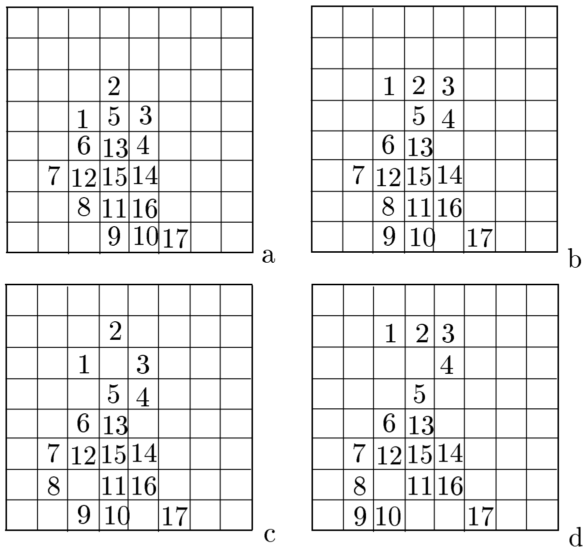

Figure 5: Self-Organizing process

5.2.3 Self-Organizing in MACM

Many inserts in the same marked place generate *deadlocks* that can be solved only by reorganizing the string over the CA's cells. Our aim is to emphasize a set of *local* rules that solve this *global* problem. An example of deadlock is presented in Figure 5a, where in the cell 14 is stored the marked symbol. Any insert is impossible in this configuration. The string must be expanded on the CA's surface, so allowing new insertions.

The self-organizing algorithm consists in applying two rules:

1. moving the active cells in new positions so as to minimize the number of neighbors.

2. generating "fluctuations" of some cells in equivalent positions.

The effects of the moving cells are presented in Figure 5b-d and Figure 6. In Figure 6h the process of spacing the string can not be continued without applying the second rule, generating a "fluctuation" of the 12th cell in a new position in which the number of neighbors is the same. But in this new configuration the first rule can be used in two steps again (Figure 7j and Figure 7k). New "fluctuations" are needed in Figure 7l and Figure 8m. The dispersing process ends when each cell has no more than two neighbors.

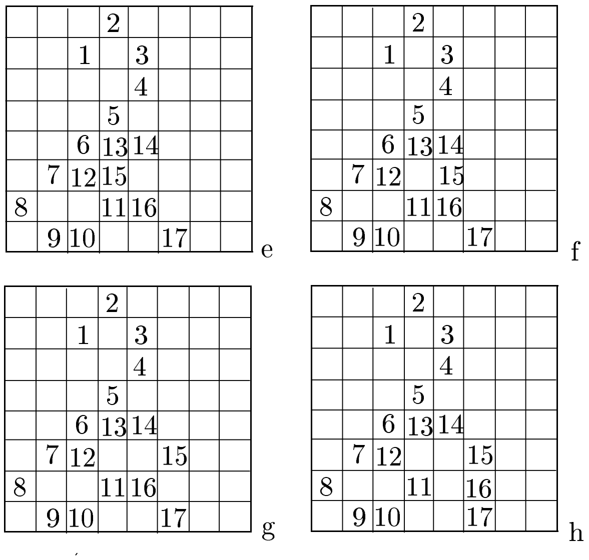

Figure 6: Self-Organizing process (cont.)

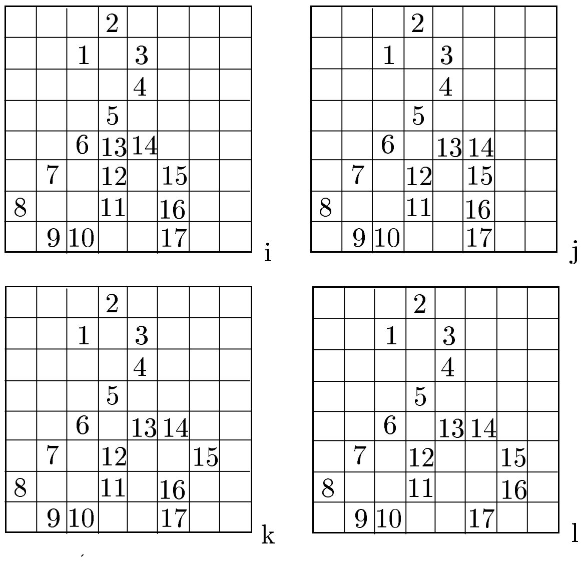

Figure 7: Self-Organizing process (cont.)

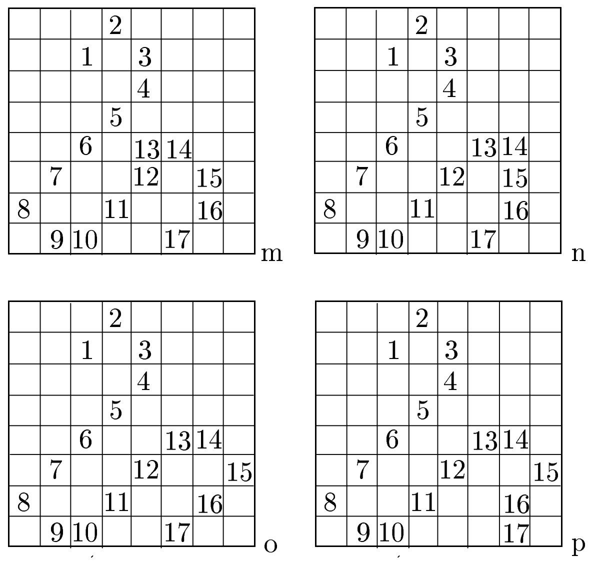

Figure 8: Self-Organizing process (cont.)

6 The Eco-Chip: a Cellular Automaton with CM's Features

6.1 The Definition

In order to perform the two molecular mechanisms, previously described, on the EC the CM function set must be adapted redefining a few simple functions. The main new features introduced by these new functions is related to the possibility to act in all the marked places on the string or on the strings stored in the cellular automaton.

Definition 10 *The* Eco-Chip *(EC) is a bi-dimensional cellular automaton each cell having two states:*

- *the* inactive *state in which the cell is unconnected with any other cell*
- *the* active *state in which the cell is connected in a string with other cells and stores a string of variables, each having a value and two states: marked and non marked.*

Over the strings stored in the active cells the following functions can be executed:

- *RESET s : all the cells from the first diagonal become active, take the value s and switch in the state non-marked*

- $FIND\,s$ *: all the variables that follow a variable having the value s switch in the marked state and the rest switch in the non-marked state*
- $CFIND\,s$ *: (conditioned find) all the variables that follow a variable having the value s and being in the marked state switch in the marked state and the rest switch in the non-marked state*
- $INSERTA\,s$ *: the value s is inserted before the* **all** *marked variables*
- $LEFT\,|\,RIGHT$ *:* **all** *the markers move one position to the left or to the right*
- $DELETEA$ *: the values stored in the* **all** *marked position are deleted, the positions remain marked and the symbols from the right are moved one position left*
- CUT *: in all marked points the strings are divided (the link is removed) in two independent substrings*
- $PASTE\,s,t$ *: all the strings move over the area of the cellular automaton and when an end of a string having the value s meets another end having the end with the value t, the two strings become one and the symbols s and t are deleted.* $\diamond$

Excepting the last function all are executed in a single clock cycle. The function $PASTE\,s,t$ depends by a random process and the end can be tested waiting for the disappearance of all the symbols s or of all the symbols t. A very important parameter in this process is the string's "mobility". The work is in progress for a simulator for experimenting algorithms that offer a lot of "mobility" to the strings stored over the cellular automaton. Many algorithms can be used, but the efficiency in performing the PASTE function must be measured only in some formal experiments using a simulator.

The previous definition describes only a theoretical model. For an actual circuit must be added some simple input-output functions.

6.2 The Full Parallel Insert/Delete Operation on the EC

The Insert/Delete operation described in Subsection 2.2 by the procedures INSERT and DELETE as such can be performed using the EC function set. The input data consists in many strings stored in the active cellular automata's cells. All the occurrences of the substrings uv or uxv are marked in the same time as for the CM applications, but now the insert or the delete of the string x is parallel performed in **all** marked points, rather than in the previous case when the operation was performed only in the first marked point.

The execution time for all insert or delete operations that can be performed over the content of EC depends only by the length of the $(u,x,v) \in I\,|\,D$, thus:

$$T_{allI/D} \in O(l(uxv)).$$

The size of the structure that performs these operations is in $O(n)$, where n is the number of cellular automata's cells.

6.3 The Full Parallel Splicing Operation on the EC

The algorithm for performing splicing operations over strings stored in EC starts similar as in the case of the system using CM, but is completed different because all the splicing operations that can be performed will be performed in parallel. The main steps are the following:

1. all the occurrences of the substring u_1u_2 (see Definition 1) are found and the special symbol α is inserted between u_1 and u_2
2. all the occurrences of the substring u_3u_4 (see Definition 1) are found and the special symbol β is inserted between u_3 and u_4
3. perform the sequence: $FIND\,\alpha$, CUT, thus generating a set of strings having the form $x_1u_1\alpha$ (and another set of strings having the form u_2x_2)
4. perform the sequence: $FIND\,\beta$, $LEFT$, $LEFT$, CUT, thus generating a set of strings having the form βu_4y_2 (and another set of strings having the form y_1u_3)
5. perform $PASTE\,\alpha, \beta$ until all αs or all βs disappear.

The first four steps are performed in time related with the length of the rule r, $T \in O(l(r))$. The execution time in the last step depends on the "mobility" of the strings stored in EC.

7 Conclusions

In this paper we have presented two solutions for performing molecular operations: the Connex Memory and its extension over a cellular automaton: the Eco-Chip. Both can be used to perform the splicing operation, the insert/delete operations or the matching operations (the last was ignored in this approach). All the three operations have two main steps:

1. identifying the places where the operations can be applied
2. performing the proper operation.

The first step parallel performed in CM or in EC, but the second can be parallel executed only with EC.

The performances of the system built around the CM circuit are the followings:

1. The execution time for **each** splicing operation is

$$T_S \in O(max.\ length\ of\ a\ string)$$

for any number of strings stored in the Connex Memory. For the insert/delete operations the execution time for **one** insert or delete operation is $T_{I/D} \in O(l(uzv))$.

2. The size of the structure of Connex Memory is: $S_{CM} \in O(n)$ where n is the number of cells.

3. Therefore: using an $O(n)$ sized data structure (containing many strings) we can perform on it only one splicing operation or insert/delete operation in $O(1)$.

4. Instead of the von Neumann architecture:

Processor - Channel - RAM

we propose a new one:

Processor - Channel - CM

in which a **very fine grain parallelism**, performed by the CM, avoids the main effect of the channel's bottleneck.

5. Substituting the RAM with the Connex Memory the area on the silicon is multiplied only by a constant and the time decreases from $O(n)$ to $O(1)$ for a system executing one splice, delete or insert at a time. Thus, the proposed architecture has a concrete, economic and performant solution.

The performances of the system built around the EC circuit are the followings:

1. The execution time for **any** number of insert or delete operations associated to one applications of the operations is in $O(l(uzv))$. For the splicing operations the time depends on the "mobility" of the strings that wind over the cellular automaton.

2. The size of EC is proportional with n, the number of cells.

3. An EC can be used as a performant co-processor for specific applications.

At the end of this paper we make a suggestion. In order to improve the "mobility" of the snakes of symbols that wind over the cellular automaton surface, a *multi-level cellular automaton* is proposed. Thus, an additional level of cells can be used to propagate the "smell" of the arguments of the function PASTE. Each cell on the basic level is connected with a correspondent cell in the second layer, used to propagate the "smells". The snakes of symbols will be oriented after the "smells" received from the second layer. We hope that using this model the strings $x_1u_1\alpha$ will meet as soon as possible the strings βu_4y_2. The experiments made with a simulator will decide on the opportunity to use "smells" to accelerate the splicing operation.

We believe that implementing in silicon the molecular mechanisms is a real challenge for the implementation in molecules.

1. The interface with a silicon machine is simpler.

2. The size of the silicon machine has the smallest dimension.

3. The time for the insert/delete operation the smallest possible: is the time to express the rule.

4. The time for the splicing operation is in the same order on the silicon and on the molecules, being related to the "mobility" of strings/molecules on the physical support.

5. But the molecular computing has a definitive advantage: it is an amazing suggestion for silicon based computing.

References

[Adleman '94] Adleman, L. M.: "Molecular Computation of Solutions to Combinatorial Problems", *Science*, 226 (Nov. 1994), pp 1021 - 1024.

[Csuhaj-Varú '93] E. Csuhaj-Varú, J. Kelemen, A. Kelemenová, Gh. Păun: *Eco (Grammar) Systems: a Generative Model of Artificial Life*, manuscript, 1993.

[Conrad '85] Conrad, M.: "On Design Principles for a Molecular Computer", *Communications of the ACM*, 28 (My 1985), pp 464 - 480.

[Hascsi, Ştefan '95] Hascsi, Z., Ştefan, G.: "The Connex Content Addressable Memory (C^2AM)", *ESSCIRC'95 Twenty-first European Solid-State Circuits Conference*, Lille-France, 19-21 Sept., 1995.

[Head '87] T. Head: "Formal Language Theory and DNA: an Analysis of the Generative Capacity of Specific Recombinant Behaviours", in *Bull. Math. Biology*, 49, p. 737-759, 1987.

[Head '92] Head, T.: "Splicing Schemes and DNA", *Lindenmayer Systems: Impacts on Theoretical Computer Science and Developmental Biology* (G. Rozenberg, A. Salomaa, eds.), Springer-Verlag, Berlin, 1992, pp 371 - 383.

[Kari '96a] L. Kari, Gh. Păun, A. Saloma, S. Yu: "DNA Computing by Using the Matching Systems", in *manuscript*, 1996.

[Kari '96b] L. Kari, Gh. Pău, G. Thierrin, S. Yu: "Characterizing RE Languages by Insertion-Deledion Systems", in *manuscript*, 1996.

[Lindenmayer '68] Lindenmayer, A.: "Mathematical Models of Cellular Interactions in Development I, II", *Journal of Theor. Biology*, 18, 1968.

[Markov '54] Markov, A. A.: "The Theory of Algorithms", *Trudy Matem. Instituta im V. A. Steklova*, vol. 42, 1954. (Translated from Russian by J. J. Schorr-kon, U. S. Dept. of Commerce, Office of Technical Services, no. OTS 60-51085, 1954)

[Păun '95a] Păun, G. (ed.): *Artificial Life. Grammatical Models*, Black Sea University Press, 1995.

[Păun '95b] Păun, G. : “On the Power of the Splicing Operation”, in *Intern. J. Computer Math.*, Vol. 59, pp 27-35, 1995.

[Păun '96] Păun, G; Salomaa, A.: ”DNA Computing Based on the Splicing Operation”, *Math. Japonica*, Vol. 43, No. 3, 1996.

[Păun '97] Gh. Păun: ”Calculatoarele pe bază de ADN”, in *manuscript*, 1997.

[Ştefan '85(91)] Ştefan, G., Bistriceanu, V., Păun, A. :“Toward a Natural Mode of the Lisp Implementation”, (in Romanian), Comm. to the Second National Simpson on Artificial Intelligence, Romanian Academy, Sept. 1985; published in *Systems for Artificial Intelligence*, Romanian Academy Pub. House, Bucharest, 1991.

[Ştefan '86] Ştefan, G.: ”Memoria conexă”, in *CNETAC '86*, 1986. (in Romanian)

[Ştefan '95] Ştefan, G., Malitza, M. : “The Eco-Chip: A Physical Support for Artificial Life Systems”, in [Păun '95].

[Ştefan, Malitza '96] Ştefan, G., Malitza, M. : “Chaitin's Toy-Lisp on Connex Memory Machine”, *Journal of Universal Computer Science*, vol. 2, no. 5, 1996, pp. 410-426.

[Ştefan '97] Ştefan, G., Malitza, M. : “The Splicing Mechanism and the Connex Memory”, in *Proc. of the 1997 IEEE Int. Conf. on Evolutionary Computation*, Indianapolis, April 13 -16, 1997.

APPENDIX

The Structure of the Connex Memory

The whole structure of the CM is represented in Figure 9, where:

- Ci, for $i = 1, 2, ..., n$, represents the i-th cell
- Transcoder is a combinational circuit that receives from each cell the markers
$$m_0 m_1 ... m_n = 00...01XX...X$$
($X \in \{0, 1\}$) and generates:
$$m'_0 m'_1 ... m'_n = 00...011...1$$
substituting with 1 all the symbols after the first occurrence of 1 and
$$m''_0 m''_1 ... m''_n = 00...010...0$$
emphasizing the first occurrence of a marked symbol.

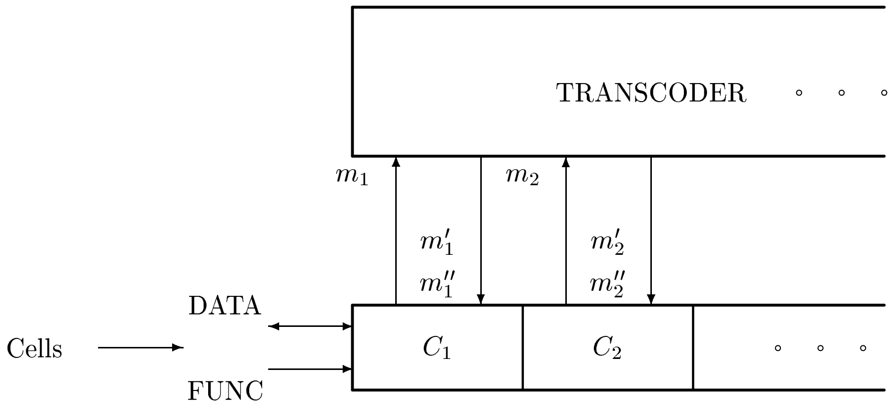

Figure 9: The structure of CM. The cells C_i, each storing a variable, are serially connected. All are externally connected through the bi-directional DATA bus and by the FUNC bus. In addition, each cell generates the marker m_i toward TRANSCODER and receives signals that classify them.

In consequence, the cells of the CM can be divided in three classes:

- the class of cells before the first cell having the marker ($m_i = 1$)
- the first marked cell
- the class of cells after the first marked cell.

Using the signals m'_i,m''_i, E_{i-1} and m_{i-1}, according to the current command (COM), each cell has enough information to switch into the next state. The size of this structure is $O(n)$ for the string of the cells and is $O(n\ log\ n)$ for Transcoder. In order to reduce the size of CM to $O(n)$ we must use a bi-dimensional solution for the cell array. It follows that the transcoder is substituted by two transcoders, each having the size $O(\sqrt{n}\ log\ n)$. This second solution allows us to define a CM with $O(n)$ complexity.

Beside the connections with the transcoder, each cell is connected with the previous and the next cell. In addition, each cell is connected to the two buses: the bi-directional DATA bus and the FUNC bus. The cell structure is presented in Figure 10, where:

- R is a p-bits register that stores the value of the variable v_i
- D is a D (delay) flip-flop that stores the value m_i of the marker
- MUX_0 is a multiplexer that selects in each clock cycle, according to the bits c_1 and c_2, the value to be stored in the register R

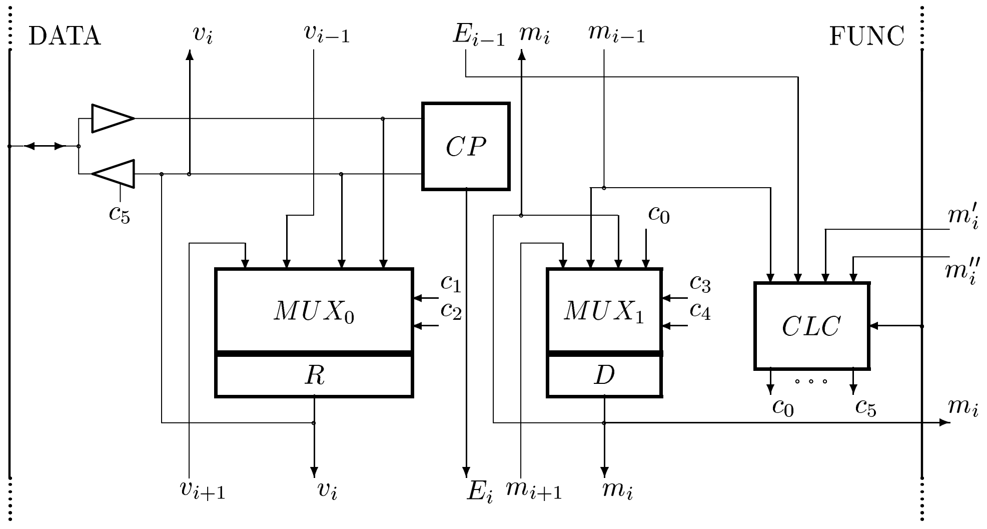

Figure 10: The organization of a CM cell.

- MUX_1 is a multiplexer, selected by c_3 and c_4, which allows storing the current value of the marker in D, in each clock cycle
- CP is a comparator that shows by the output E_i if v_i is equal with the value applied to the input DIN
- CLC is a combinational circuit that generates the control bits c_0, c_1, ..., c_5, according to: the command received from COMP, the bits E_{i-1}, m_{i-1}, received from the previous cell, and the bits m'_i, m''_i, generated by the Transcoder (see Figure 9)
- c_0 is the value of the locally generated marker
- c_5 is the enable input for the tristate circuit that drives the bi-directional bus DATA.