The Connex Memory.
A Physical Support for Tree / List Processing

Gheorghe Ştefan*

Abstract

Architectural deficiencies in symbolic processing systems are generated by the lack of an appropriate implementation of their memory functions. There are many well-defined memory functions, but there is not yet an adequate structural implementation for any of them. The main memory functions are usually software implemented as data structures, using RAM as hardware support, but having low performance in applications because all implementations rely on sequential mechanisms. Our proposal, *the connex memory (CM)*, involves a base-level structural parallelism that would increase the performance in tree/list processing. This paper suggests, also, the architecture and the structure of a computation model: *the automaton with connex memory (ACM)*, which we consider a superior alternative for sequential memory in the design of tree/list oriented architectures.

Keywords: computation models, automata, tree, list, architecture, data structure, memory functions, fine grain parallelism, content addressable memory.

1 Introduction: Why Another Model?

Computer architectures were developed promoting mainly the numerical features of the basic computation models, such as Turing machine or Kleene's recursive functions. This orientation has dominated for a long period the computer science evolution. The first proposal for concrete computer architecture was the von Neumann's model, strongly oriented toward numerical applications [Neumann '45]. Lambda-calculus, the computation model proposed by Church, has been used later, initially only in the theoretical approach to computation and, after that, for developing the large class of the non-numerical (symbolical) applications. There are many incompatibilities between von Neumann architecture and the mechanisms used in symbolic processing; as a consequence they

*Politehnica University of Bucharest, Dept. of Electronics Bd. Iuliu Maniu 1-3, Bucharest 6, ROMANIA; E-mail: stefan@agni.arh.pub.ro

can't be used efficiently in concrete applications. Because tree / list are the most common data structures used in symbolic-oriented applications, new architectures are required that accommodate better to the tree/list processing. In this paper *the connex memory (CM)* [Ştefan '85] is considered, as a support for such new architectures.

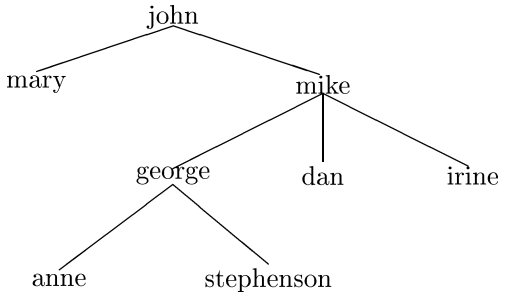

Fig. 1 An example of tree

Starting from the technological limitations and conceptual restrictions, computer science has developed, preponderantly, serial accessed data structures as queues, stacks, lists ... . All these memory functions are implemented like *pipes* (queue, stack) or *combinations of pipes* (the list that can be seen as two stacks), in such a manner that almost *all the stored content is hidden* from the user all the time. In this condition, it seems that the first function of a usual memory structure is *to hide* and only the second function is *to store*. This is the main disfunctionality of the actual approach.

The main feature of a new memory function must be to allow a less restricted access to the stored configuration. Starting from this point of view we define, in this paper, *a new memory function: the connex memory*. Instead of "pipe oriented" implementation procedures we promote "eaves oriented" mechanisms to build memory functions (waiting for the moment when "plane oriented" philosophy will be possible).

1.1 The first goal: a global accessed memory

Our proposal introduces a memory device for solving the contradiction between the list, as a serial accessed data structure, and the necessity to access in a global manner the content of a list, as a usual mechanism involved by symbolic processing.

Let us explain. As we know, the tree is a complex general data structure, which has the list as its best symbolic representation. The power of the tree representation is due to its spatial suggestion in the graphic representation. We loose this suggestion in list representation because, in the common implementation, the list is a serial accessed data structure. In this sense, we introduce a

base-level parallelism in symbolic processing, "opening" the *pipes* and so generating *eaves implemented memories.*

First Example. Let be the list representation of the tree from figure 1:

$$(john(mary\, mike(george(anne\, stephenson)dan\, irine)))$$

If we want to find George's successors, using a list implementation as a *dynamic* data structure on a standard computer, the processing time is $O(f(n))$, where n is the number of tree's nodes. A *global access* to the list allows us to find the desired information in $O(1)$, using parallel facilities developed in an appropriate implementation of a new memory structure. The first thought led us to the well-known memory function: the circuit that implements the *content addressable memory (CAM)* [Kohonen '87].□

john	mary	mike	
mary			
mike	george	dan	irine
george	anne	stephenson	
dan			
anne			
stephenson			
irine			

Fig. 2 The image of the tree in CAM

But this approach is not sufficient because it doesn't handle efficiently the following important situations:

1. We want to maintain the list associated to the tree in the "natural" form, as a string that occupies a minimum space on the memory support, having a *connex* representation. Unfortunately, in a standard CAM this tree has the form represented in the figure 2 (or an equivalent other one).

A big amount of space is lost in the standard CAM circuit, *due to the inflexibility of the storage cell dimension* and of the access mechanisms [Hascsi '94]. In a CAM circuit, for n-ary trees, are allocated $O(n^2)$ storage cells, instead of $O(n)$ storage cells in a "natural" form. Only for binary trees in CAM the space has the dimension $O(n)$, the same as in a "natural" form, requiring always the translation from n-ary trees to binary trees.

2. The relation between names is not so evident as in the "natural" form. For example, we can't answer easily, using the CAM circuit as a support for the tree representation, the question *"what is the imbrication level of the name George?"*.

3. Another problem is the fixed dimension of the storage cell used for each name. In CAM circuit implementation we must use the same dimensioned cell for each name, loosing, in consequence, a big amount of elementary cells (each cell stores one symbol). Indeed, for the name *Anne* and for the name *Stephenson* we must use two memory cells having the same dimension, equal to the length of the greatest name expected to be memorized.

4. A final objection: if a CAM circuit is used as a storage support,then a string of symbols having *certain length* can be found only using complicated procedures. Indeed, if we want to check whether the subtree

$$(mike(george(anne\ stephenson)dan\ irine))$$

exists in our representation or not, then a very complex procedure must be used because in CAM circuits the list associated to the tree has not a "natural" form. In CAM each atom can be searched very easily, but it is not the same for a list.

Therefore, starting from the CAM facilities, we must enhance this circuit with the functions that allow us *to extend* the possibilities to operate with atoms (names) toward *the facilities to operate with lists and trees.* In other words, *we must replace the local access* (limited to symbolic strings with $O(1)$ length) *with the global access* that *allows us to operate over strings having unspecified length.* In the same time, we want to define and implement a circuit that maintains an internal "natural" representation of lists in a *connex* form that assures a better internal space management (each list uses a number of cells equal to its own number of symbols). The pointers (in RAM oriented implementation) or the additional cells (in CAM oriented implementation) are avoided in the *connex memory philosophy.*

With the connex memory we try to introduce a memory circuit very useful in the *list* oriented operations, instead of the old CAM that is efficient only in the *atom* oriented operations.

1.2 The second goal: declutching of data processing from data structuring

In the current computer architectures the main subsystems, the *processor* and the *memory*, have an unbalanced role in computation. Indeed, the processor performs, in the same time, two basic actions:

- data structuring (the processor organizes the content of the random access memory in data structures)
- data structure processing

and the memory is only a *passive* storage support.

The second goal of our model is to offer a physical device that helps to the declutching of *data processing* from *data structuring* as processes developed

strictly at the level of the processor. If we have an "appropriate" memory device, we can promote an efficient *independent implementation* for data structures at the storage support level.

A computer architecture, built on the **Processor + Channel + RAM** structure, implies strong dependencies between languages, their implementation, data structures and applications. A good example is the correlation between the Lisp language, the role of the functions CAR, CDR and CONS in the implementation and the list data structure based on the two pointers mechanism that implies the prevalence of the binary trees instead of n-ary trees. Even if we add the structural facilities of CAM, we can't avoid, without a big price, the necessity to convert the trees in binary form.

For an *independent list implementation* in a Lisp or Prolog oriented architecture we can use an appropriate memory circuit, having a set of functions oriented toward *list data structures.* We propose this circuit as *the connex memory* (CM).

1.3 The third goal: parallelism oriented toward the memory function instead of the processing function

The usually spread solution for parallelism is to multiply the processing units and to build an appropriate storage system. In this case the processing function is parallelized. The simplicity of the memory function does not stimulate its possible expansion in parallel processes. But, if we add some processing functions at the storage system level, then an implicit parallelizing process is started. Indeed, if we add to each storage cell some simple processing functions, then these functions will be executed in each cell at the same time. Therefore, a natural parallelism appears. More than that, for these simple functions we avoid the communications through "von Neumann's bottleneck" [Backus '78].

Our main problem is to find a simple and efficient function set to be executed at the storage cell level. Functions must be simple, allowing only a small increasing of the cell size, and, at the same time, they must be efficient in large scale applications. Our proposal tries to find a memory function set and its related structure so as the cell size should increase only 10-20 times. Therefore, the performance of the system that uses such a memory must increase "much more" than $O(1)$ times.

Our approach supposes that some processing functions migrate from the processing unit into the storage unit. Each stored bit must gain its own small "processing unit". The memory becomes "smart" and a very fine grain parallelism can be started in the system.

2 What is the Connex Memory?

Let us imagine a line of soldiers walking on a narrow bridge one after the other. Above them flies a helicopter with their commanding officer. Each soldier has a

phone that connects him with the officer. The soldiers have important different missions, and the order in which they step out from the narrow bridge is of the utmost importance. The officer can pick up on the helicopter any soldier and drop any soldier in any place. But the helicopter has only one rope that can be in one place only on the bridge. Suppose the officer has to move the soldiers having the best guns in front of the line. He commands all the soldiers having good guns raise their hands. He goes to the first with hands up and drops the rope to haul him. Then he goes to the first in line (he always knows where the first and last soldiers are) and drops the soldier there. And so on.

Now let's imagine another scenario, in which the officer wants to identify a sequence of three soldiers, the first having a gun G1, the second a gun G2 and the third a gun G3, and to put, with the helicopter, in front of this small team, a soldier having a gun G. To achieve this goal the commands must be in the order:

1. *Only the soldiers having a gun G1: raise their hands!*
2. *Only the soldiers being preceded by a soldier with hands up and having a gun G2: raise their hands!*
3. *Only the soldiers being preceded by a soldier with hands up and having a gun G3: raise their hands!*

After these commands the officer can "insert" a new soldier, dropping from the helicopter a soldier equipped with the gun G before the first soldier having raised hands, if this soldier exists, using the command:

Insert the soldiers with the gun G.

With the first three commands the officer selects the first occurrence of the desired subsequence (team) of soldiers in the "string" of soldiers and with the last command he modifies the content of the string by inserting a new soldier.

We cannot imagine such operations with a line of soldiers walking in a tunnel. This is the main difference between the *pipe* oriented and the *eaves* oriented mechanisms used in memory implementation, when looking for global access to the content of the data structure in a parallel mode. The distinction between our proposal and the standard implementation of usual data structures (i.e., queues, stacks, lists, trees, ...) is best illustrated by the distinction between a *narrow bridge* and a *tunnel*.

2.1 Informal Definitions of CM

An informal description of the CM can be the following: the physical support for a symbolic string in which we can find any substring, thus identifying any place in the string, for reading, inserting or deleting a symbol or a substring of symbols.

The CM is *a sort of CAM*, structured as a *bi-directional shift register* in which a significant point is *marked*, as a consequence of an *associative sequential mechanism* (see *Second Example* in subsection 2.2) used to find a name in a number of steps equal to the length of the name. In the marked place, *read*,

insert and *delete* can be performed. Theoretically, the CM is unlimited at the right end, and, consequently, can be used for simulating or emulating efficiently any number of registers having an unspecified length (theoretically infinite).

Informal Definition. The *connex memory* CM has the following description: a physical support of an unlimited string of variables (figure 3), having values from a finite set of symbols and two states: *nonmarked* or *marked*, over which we can apply the following set of functions:

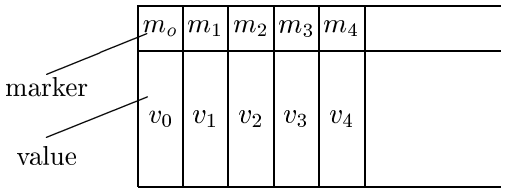

Fig. 3 The content of the CM

- $RESET\ s$: all the variables after the first marked variable 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 nonmarked state
- $CONDITIONAL\ FIND\ s$: all the variables that follow a marked variable having the value s switch to the marked state and the rest switch to the nonmarked state
- $INSERT\ s$: the value s is inserted before the first marked variable
- $READ\ up|down|-$: 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 (in the current cycle the output has the value of the deleted variable) and the symbols from the right are moved one position left.

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

2.2 The Formal Definition of CM

The connex memory is a parallel non-deterministic automaton defined by the following quintuple:

$$CM = (X, Y, Q_0 \times Q_1 \times \ldots \times Q_n, f, g)$$

where:

$X = S \times F$ is the input set

$S = \{s_1, s_2, \ldots, s_m\}$ the finite set of input symbols

F is the set of functions executed by the CM:
$RESET\, s$, $INSERT\, s$, $FIND\, s$, $CONDITIONAL\, FIND\, s$, $READ$,
$READ\, up$, $READ\, down$, $DELETE$

$Y = S$ is the set of output symbols

$Q = Q_0 \times Q_1 \times \ldots \times Q_n$ is the state of the automaton, with:
$Q_i = S \cup S'$, where $S' = \{s'_1, s'_2, \ldots, s'_m\}$, for $i = 0, 1, ..., n$

f is the state transition function

g is the output transition function.

The functions f and g will be defined, through the action of the functions from F, in procedures written using the following conventions:

$(q_0, \ldots, q_n)$ is the instantaneous configuration of variables in CM, $q_i \in S \cup S'$

$q_i = s$ means: the q_i element of the instantaneous configuration has the value $s \in S$ and the state non-marked

$q_i = s'$: the q_i element of the string has the value s and is marked

$p = min(i | value\, of\, q_i \in S')$**:** q_p is the first marked variable in the string

$q_i \leftarrow s$: the variable q_i takes the value s and the non-marked state

$q_i \leftarrow s'$: the variable q_i takes the value s and the marked state

$q_i \leftarrow q'_i$: the state of q_i switches in the marked state without changing the value

$q_i \leftarrow q_i$: q_i switches in a non-marked state, without changing the value

$q_i \leftarrow q_j$: the variable q_i takes the value and the state of variable q_j.

The set of functions, F, executed by CM is described in the following:

Procedure $RESET\, s$ / reset to s all the values in CM /
$RESET_0\, s | \ldots | RESET_n\, s$ / parallel non-deterministic behaviour /
end $RESET\, s$

Procedure $RESET_i\ s$ / for $i = 0, 1, ..., n-1$ /
if$i > p$
then $q_i \leftarrow s$/ the variable takes the value s/
else noop
endif
end $RESET_i\ s$

Procedure $INSERT\ s$ / insert s in the place of the first marked variable /
$INS_0\ s | INS_1\ s | \ldots | INS_n\ s$
end $INSERT\ s$

Procedure $INS_i\ s$ / for $i = 0, 1, ..., n$ /
case : $i < p$ **:** noop / no operation /
: $i = p$ **:** $q_i \leftarrow s | q'_i \leftarrow q_i$ /q_p takes the value s/
else $q_{i+1} \leftarrow q_i$ /all q_i are right shifted one position /
endcase
end $INS_i\ s$

Procedure $FIND\ s$ / unconditional find of value s/
$UF_{-1}\ s | UF_0\ s | UF_1\ s | \ldots | UF_{n-1}\ s$
end $FIND\ s$

Procedure $UF_i\ s$ / for $i = 0, 1, ..., n-1$ /
if$q_i = s$
then $q_{i+1} \leftarrow q'_{i+1}$/ the next variable is marked /
else $q_{i+1} \leftarrow q_{i+1}$ / the next in non-marked state /
endif
end UF_is

Procedure $UF_{-1}\ s$
$q_0 \leftarrow q_0$
end UF_{-1} s

Procedure $CONDITIONAL\ FIND\ s$ / conditioned find of value s/
$CF_0\ s | CF_1\ s | \ldots | CF_{n-1}\ s$
end $CONDITIONAL\ FIND\ s$

Procedure $CF_i\ s$ / for $i = 0, 1, ..., n-1$ /
if $q_i = s'$
then $q_{i+1} \leftarrow q'_{i+1}$
else $q_{i+1} \leftarrow q_{i+1}$
endif
end $CF_i\ s$

Procedure $READ$ /q_p is sent to the output OUT/
$OUT \leftarrow q_p$
end $READ$

Procedure $READ\,up$ / $READ$ and the marker moves one position right /
$OUT \leftarrow q_p | q_p \leftarrow q_p | q_{p+1} \leftarrow q'_{p+1}$
end $READ\,up$

Procedure $READ\,down$ / $READ$ and marker moves one position left /
$OUT \leftarrow q_p | q_p \leftarrow q_p | q_{p-1} \leftarrow q'_{p-1}$
end $READ\,down$

Procedure $DELETE$ / delete and read the value of q_p/
$OUT \leftarrow q_p | DEL_1 | DEL_2 | \ldots | DEL_n$
end $DELETE$

Procedure DEL_i / the value of q_p is deleted but the state is preserved /
case : $i = n$: nop
: $i < p$: nop
: $i = p$: $q_i \leftarrow q_{i+1} | q_i \leftarrow q'_i$
else $q_i \leftarrow q_{i+1}$
endcase
end DEL_i

2.3 Utility of the Connex Memory Functions

Let us see two examples illustrating the main abilities of the CM [Ştefan '96].

Second Example Using the CM we can find any substring in a string. Suppose that we have in a CM the following string:

$$...(bubu\ (bad\ butcher))...(bulgaria(...))...$$

and we want to select the list of *bubu*'s properties (i.e., (*bad butcher*)). The following sequence of functions is executed:

FIND '('
CONDITIONAL FIND b
CONDITIONAL FIND u
CONDITIONAL FIND b
CONDITIONAL FIND u
CONDITIONAL FIND blank
loop
READ up
until ')'
repeat

The content of the CM becomes successively (the marked places are indicated by oversigned symbols):

...(b̂ubu (b̂ad butcher))...(b̂ulgaria (...))...
...(bûbu (bâd butcher))...(bûlgaria (...))...
...(bub̂u (bad butcher))...(bul̂garia (...))...
...(bubû (bad butcher))...(bulgaria (...))...
...(bubu^(bad butcher))...(bulgaria (...))...
...(bubu (̂bad butcher))...(bulgaria (...))...
...(bubu (b̂ad butcher))...(bulgaria (...))... / $out = ($ /
...(bubu (bâd butcher))...(bulgaria (...))... / $out = b$ /
...(bubu (bad̀ butcher))...(bulgaria (...))... / $out = a$ /
...(bubu (bad^butcher))...(bulgaria (...))... / $out = d$ /

and so on, until:

...(bubu (bad butcher))̂...(bulgaria (...))... / $out =)$ /. □

Third Example. Let us suppose that we want to change the first property of *bubu* with the value *good*. The sequence of functions will be:

FIND '('
CONDITIONAL FIND b
CONDITIONAL FIND u
CONDITIONAL FIND b
CONDITIONAL FIND u
CONDITIONAL FIND blank
READ up
loop
  DELETE
  until blank
repeat
INSERT g
INSERT o
INSERT o
INSERT d
INSERT blank

Now the new content of the CM becomes:

...(bubu (good butcher))...(bulgaria (...))... □

2.4 The Structure of the Connex Memory

In this paper we present only a brief description of the CM internal structure for illustrating the basic idea of the CM family chips. More details about the structure of CM and other CM chips are presented in [Ştefan '94]. The internal structure of a cell, associated to one symbol, is presented in the figure 4, where:

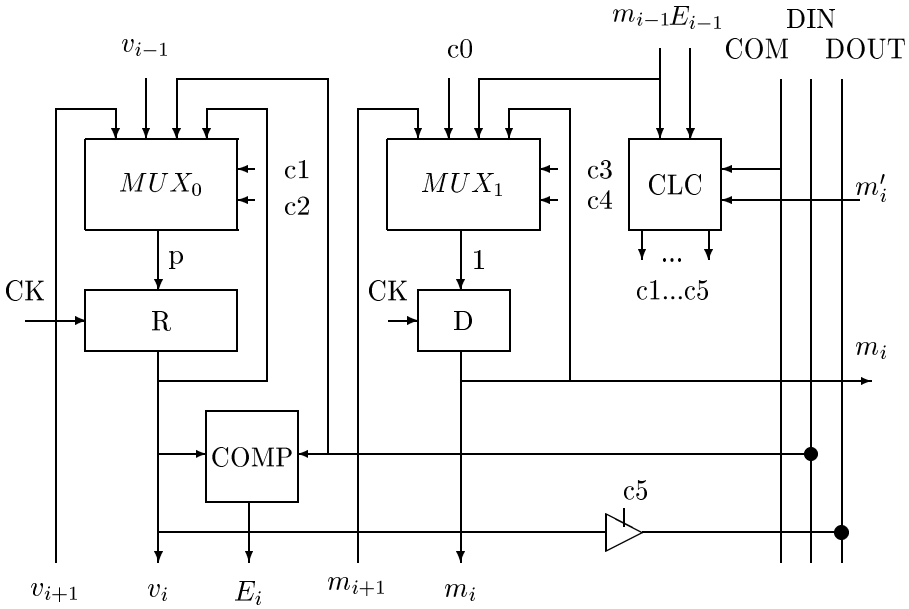

Fig. 4 The structure of the CM's cell

- 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 (CK) cycle, according to the bits $c1$ and $c2$, the value to be stored in the register R
- MUX_1 is a multiplexer, selected by $c3$ and $c4$, which allows storing the current value of the marker in D, in each clock cycle
- COMP 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 $c0$, $c1$, ..., $c5$, according to: the command received from COMP, the bits E_{i-1}, m_{i-1}, received from the previous cell, and the bit m_i generated by a Transcoder (see Fig. 5)
- $c0$ is the value of the locally generated marker
- $c5$ is the enable input for the tristate circuit that drives the output DOUT.

We can represent the whole structure of the CM as in the figure 5, where:

- Ci, for $i = 1, 2, ..., n$, represents the i-th cell

- Transcoder is a combinational circuit that receives from each cell (X means don't care value) the markers

$$m_0 m_1 ... m_n = 00...01XX...X$$

and generates

$$m'_0 m'_1 ... m'_n = 00...011...1$$

substituting with 1 all the symbols after the first occurrence of 1.

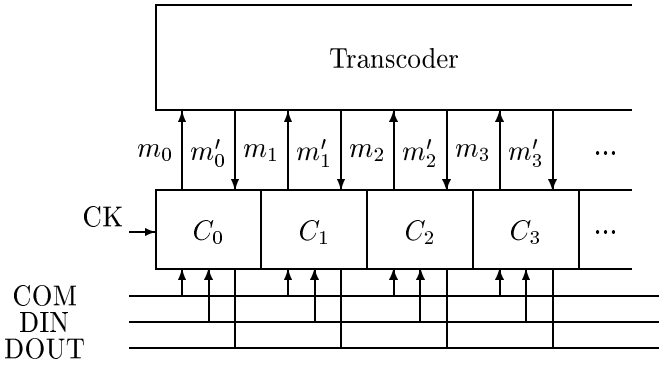

Fig. 5 The functional structure of the CM

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, 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 find another solution for the Transcoder. This solution (see Fig. 6) is a bi-dimensional array, for cells, and two transcoders (TCX and TCY), having the size $O(\sqrt{n} log\ n)$. This second solution allows us to define a CM with $O(n)$ complexity.

The actual size of CM depends on the cell dimension. A classical approach using dynamic latches leads us to around 20 MOS transistor per bit. Using a bucket-brigate technology the transistor number can be reduced to 10. Smallest area can be used when a CCD technology is adopted. A standard CAM can be built with around 10 transistor per cell. The additional features of CM justify a small size increasing. But is very important that we remain in the same magnitude order.

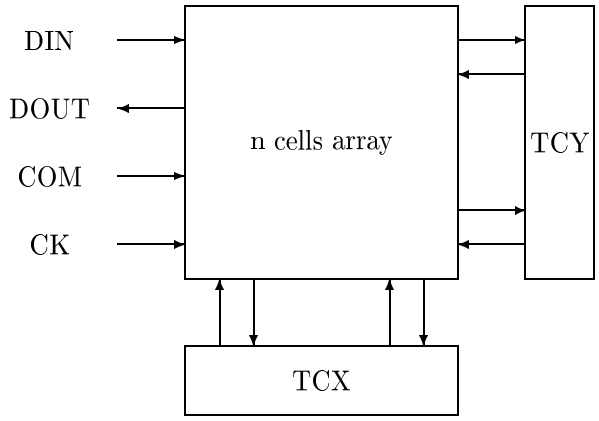

Fig. 6 The CM organized as a bi-dimensional array of cells

Because the depth of transcoder is $O(1)$ for a theoretical model (one chip approach) [Ştefan '94], the execution time for each CM's function is $O(1)$. (In real applications, for n cells, if we use constant sized CM chips, then the depth of the expanded transcoder becomes $O(log\ n)$ and the execution time increases at $O(log\ n)$. But we can assume that the execution time for each function remains "almost" $O(1)$, as in systems that use RAM chips.)

3 The Connex Memory at Work

3.1 Automaton with Connex Memory

The simplest system configuration using CM is the *Automaton with Connex Memory* (ACM). At the same time, ACM can be seen as a *computability model.*

Definition 1. An ACM (see Fig. 7) is structured using a CM and a finite automaton defined as follows:

$$A = (S, S \times F, Q, f, g; q_0)$$

where:

- the sets S and F are defined in the formal definition of CM (see subsection 2.2 in this paper)
- Q is the finite set of internal states of the automaton
- f is the state transition function, defined in $Q \times S$, with values in Q (generates the new state according to the present state and sometimes to the present output of the CM, if DATA of the CM (see Fig. 7) is in output mode)

- g is the output function of the automaton, defined in $Q \times S$, with values in $S \times F$, if DATA (see Fig. 7) is in input mode, or else, in F
- q_0 is the initial state of the finite automaton.

The ACM starts from q_0 having in CM an initial string of symbols and stops in a final state with a string of symbols, as result, in CM.□

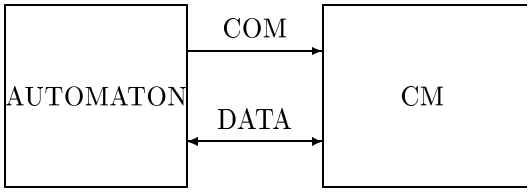

Fig. 7 Automaton with Connex Memory (ACM)

Obviously, all the partial recursive functions can be computed using an ACM. Indeed, each action performed by the Turing machine can be performed with an ACM.

Proposition 1. The ACM can expand the function set of the *First Non-formal Definition*, defined on symbols, adding similar functions on strings of symbols, defining in this way a List Oriented Reduced Architecture (LORA). The additional new functions are:

- $RESET < string >$: the variables of the CM take, after the first marked variable, the values of $< string >$ and the first symbol of the $< string >$ is repeated, after $< string >$, until the end of the CM
- $FIND < string >$: the marker is set to the end of all substrings having the value $< string >$
- $INSERT < string >$: the string of symbols $< string >$ is inserted in front of the first occurrence of the marker
- $READ\, s|-$: the first marked s-expression, if s, or the first marked symbol are read executing $READ\, up$
- $WRITE < string >$: the string of symbols $< string >$ substitutes a string having the same length starting with the first marked symbol
- $SKIP\, up|down,\, s|-$: the next (up) or previous ($down$) s-expression (s) or symbol (-) is skipped with the first marker
- $DELETE\, s|-$: the s-expression that starts with the first marked variable (s) or the first marked symbol (-) is deleted
- END : executes $READ$.

By s we understand a Lisp object (atom or list). All these new functions are executed in a number of cycles equal with the length of the string $< string >$ or of the s-expression s. □

We can prove immediately this proposition by defining the procedures that describe the automaton evolution for each function.

3.2 Functions Defined on Trees/Lists

In applications we can introduce a Bus Connected ACM (BCACM), see figure 8, as the smallest complete system configuration, in which the INT (interface) interconnects an ACM having, for example, associated LORA, with a standard computer system, to implement a powerful coprocessing function in the list oriented jobs.

In the next examples we suppose that:

- in CM there is only one list, representing a tree (see *First Example*), with n symbols, followed by an unending sequence of #
- the symbols #, @, $ are used only by the system, they do not appear in the initial form of the list.

Fourth Example Using LORA, in a BCACM configuration, we can answer, with the procedure SUBTREE $< name >$, the question: *is there a subtree with a root* $< name >$*?*

```
Procedure SUBTREE < name >
            FIND < name >
            SKIP down
            if out = blank
              then SKIP up
                   if out = '('
                     then FIND #
                          INSERT yes
                     else FIND #
                          INSERT only leaf
                   endif
              else FIND #
                   INSERT no
            endif
end SUBTREE
```

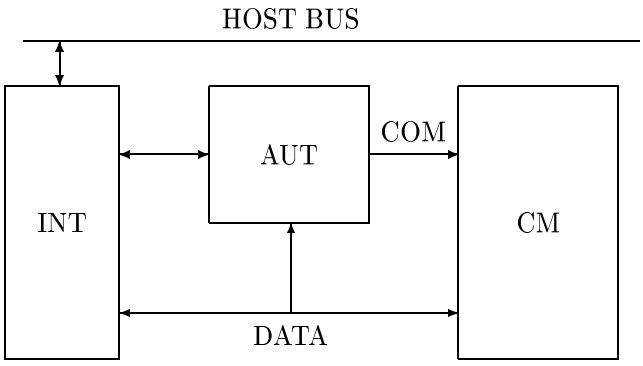

Fig. 8 Bus connected ACM, as a tree/list coprocessor

At the end of the list, after the first # the procedure SUBTREE inserts the answer: *no*, *only leaf* or *yes*. The execution time is $T(n) = O(1) = number\ of\ symbols\ of\ the\ name + constant$.

For example, if in the CM we have the list from the first example (see section 1.1) and we call SUBTREE *george*, then at the end of the execution the content of CM becomes:

$$(john(mary\ mike(george(anne\ stephenson)dan\ irine)))\#yes\#\#\ldots$$

If we call SUBTREE *dan*, then the answer will be:

$$(john(mary\ mike(george(anne\ stephenson)dan\ irine)))\#only\ leaf\#\#\ldots$$

and for SUBTREE *johny* the answer will be:

$$(john(mary\ mike(george(anne\ stephenson)dan\ irine)))\#no\#\#\ldots.$$

□

Fifth Example Using LORA we can solve the problem of counting the levels of the tree by procedure DEPTH.

```
Procedure DEPTH
          FIND '('
          while out = blank
                do SKIP down
                   WRITE @
                   FIND #
                   INSERT $
                   FIND '('
          repeat
          FIND @
          while out = blank
```

```
                do SKIP down
                   WRITE '('
                   FIND @
          repeat
end DEPTH
```

The answer is stored in 1-ary representation after the first # using the symbol $. In the first *while loop* all '(' are substituted by @ and are 1-ary counted. The second loop restores the list substituting all @ with '('. If the length of the tree representation is n and the depth of tree is m, then the execution time is $T(n) = O(m)$, independent of n. □

Sixth Example Another problem is: *what is the level on which the string* $<$ *string* $>$ *is positioned.*

```
Procedure LEVEL < string >
          FIND < string >
          SKIP down
          if out = blank
            then READ up
                 while out ≠ ')'
                       do SKIP up s-ex
                 repeat
                 loop  WRITE @
                       FIND #
                       INSERT $
                       FIND @
                       until out ≠ ')'
                 repeat
                 FIND @
                 SKIP down
                 loop  WRITE ')'
                       SKIP up
                       until out ≠ @
                 repeat
            else FIND #
                 INSERT no
          endif
end LEVEL
```

The answer can be *no* or a 1-ary number after the first #. Unfortunately, the execution time is $T(n) = O(n)$ because $SKIP\ up\ s$ is executed symbol by symbol going through over s-expressions with the function $READ\ up$. □

4 Conclusions

This paper presented only the basic idea of a class of logic-in-memories: *connex memories*. Some preliminary conclusions will be enumerated.

1. The connex memory allows a *base-level parallelism* (*all* the soldiers receive and can execute a command) in a *surface sequential process* (the strategy applied by the officer using a string of commands). Our mind generates, often, an inherent sequential algorithm to be executed. In this situation the parallelism can be initiated only at a base-level in the computer structure. One solution is to use a connex memory as a circuit for storing and manipulating data structures.

A small amount of processing functions were shifted near each stored bit. This is one of the main idea of CM. Therefore, there are some simple functions that can be parallel performed avoiding the communication through the "von Neumann's bottleneck". There are many functions in symbolic computation that imply simple local performed actions. These facilities allow us to trigger parallel processes at the base level in the structure of the computing machine. For example, search operations in dynamic data structure have better solutions in systems with CM.

2. If we use a parallel approach with multiprocessors or multicomputers, then our best expectation is to multiply the execution speed only with a *constant* value. Elsewhere, when a base-level parallelism is initiated, as in CM approach, the *magnitude order* of the processing time will be affected for some functions.

For example, the environment in a Lisp program interpretation is a *dynamical data structure* in which finding a pair *name-value* can be performed in constant time instead of $O(f(n))$ time in system that uses large RAMs and sophisticated algorithms. In this case compilation techniques can not be used because of the dynamic growth of the structure (for static data structure the compiler translates the algorithm to low level without significant loss of efficiency).

3. Our belief is that a good *architectural approach*, supported by a well-adapted memory device, will help us to find better solutions for the problems generated by the circuit complexity. Instead of large RAM and sophisticated algorithms running on powerful processors we propose a new architectural facility: **the connex memory (CM)**, hardware set-up as a simple device. Therefore, **new functions instead of large structures**.We think that the *architectural flexibility* is more efficient than the *structural flexibility* for solving the problem of size and of time in computing complexity.

In [Ştefan '96] where presented (subsection 4.4) comparisons between a CM architecture and other models concerning basic operations on strings. The architecture with CM has the better theoretical time performances thanks to the functions performed by CM in one cycle.

4. The connex memory can be a very good choice for standard content addressable memory, because it can execute the same functions with a smaller structure (see *First example*). Also, memory functions as stacks, queues, and

data structures as vectors, arrays, trees or graphs that are implemented with CAM [Potter '94], are better implemented in CM. The *connex* representation in CM allows a very good management of this memory.

5. CM, as a CAM with some additional features, can improve all earlier associative processing approaches (such as Staran machine [Batcher '74]), because it optimizes content-addressability and it adds new possibilities, for *dynamic data structures*, such as *insert*, *delete* and *string matching*. CM's functions satisfy, almost completely, the requirements specified [Potter '94] for some advanced applications (such as genome sequencing): "Genome sequencing requires integration of knowledge retrieval, efficient insertion and deletion of data elements, and efficient manipulation of matrices for the heuristic matching of sequences."

6. DNA computing by the *splicing operation* opens new ways for CM application. Păun and Salomaa proved [Păun '95, '96] that the splicing operation grounds a computational model and in [Ştefan '96a] are presented the facilities offered by a CM architecture for implementing a "splicing machine".

7. The future work on this subject will be focused on developing new application oriented functions, starting from the main idea of the CM. At the same time new functions will be added to the basic definitions starting from the experience with the first variant of CM used in applications.

For example, because the time performance of the function $SKIP\ up|down, s|-$ depends on the length of the s-expression, we must look for new facilities in the next versions of the CM, such as functions on predefined "windows". See more on new functions of CM in [Mîţu '94].

8. The idea of the CM can be valued in developing useful specific applications, such as coprocessors for accelerating Lisp language interpretation [Ştefan '96], [Mîţu '96] or Prolog execution [Mîţu '96a] (the *unification mechanism* is very difficult to be implemented on conventional machines because the inefficiency of searching techniques (such as searching trees or hashing techniques) on dynamic data structures). Nowadays, the architectures oriented toward Lisp or Prolog language pay attention mainly for the processing unit. CM offers a support for a memory oriented approach in solving the critical functions involved by Lisp interpretation or Prolog execution.

9. Architectures based on the *rewriting systems* represent another domain in which the systems with CM are fery useful. In [Ştefan '98] are investigated systems based on the Markov algorithms and the Lindermeyer grammars.

10. Functions defined on not disjoint sets are very well performed with a connex memory oriented architecture [Mîţu '95]. Also, theoretical studies can be written starting from the idea of the CM [Paun '94].

Acknowledgments The author wishes to thank Bogdan Mîţu and Zoltan Hascsi for useful discussions and suggestions about the functions and the structure of the connex memory. For their help in improving the general quality of this paper, the author is extremely grateful to Dr. Mihaela Malitza, Prof. Cristian Calude, Dr. Gheorghe Paun and to the reviewers.

References

[Batcher '74] Batcher, K, "Staran Parallel Processor System", *Proc. National Computer Conf.*, AFIPS, 1974, pp. 405-410.

[Backus '78] Backus, J.: "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM*, Aug. 1978.

[Chaitin '87] Chaitin, G. J. : *Algorithmic Information Theory*, Cambridge Univ. Press, 1987.

[Chaitin '94] Chaitin, G. J. : *The Limits of Mathematics IV*, IBM Research Report RC 19671, e-print chaodyn/9407009, July 1994.

[Hascsi '95] Hascsi, Z., Ştefan, G.: "The Connex Content Addressable Memory (C^2AM)", *Proceedings of the Twenty-first European Solid-State Circuits Conference*, Lille -France, 19-21 September 1995, pp. 422-425.

[Kohonen '87] Kohonen, A.T. : *Content-addressable Memories*, 2nd. ed., Springer Ser. Inform. Sci., Springer-Verlag, Berlin, Heidelberg, 1987.

[Mîţu '94] Mîţu, B., Ştefan, G. : "Enhanced Version of the Connex Memory used in List Processing Oriented Architecture", preprint, Center for New Electronic Architecture of the Romanian Academy, Sept. 1994.

[Mîţu '95] Mîţu, B.: "Symbolic Processing Using Connex Memory", preprint, Center for New Electronic Architecture of the Romanian Academy, Jan. 1995.

[Mîţu '96] Mîţu, B., Corina Mîţu : *ToyLisp Interpreter on a Connex Memory Machine*, *Journal of Universal Computer Science*, vol. 2, no. 5, 1996, pp. 427-438.

[Mîţu '96a] Mîţu, B.: "A Logic-in-Memory Approach for Prolog Unification", preprint, Center for New Electronic Architecture of the Romanian Academy, Sept. 1996.

[Malitza '88] Malitza, M.: private communication.

[Neumann'45] John von Neumann: "First Draft of a Report on the EDVAC", reprinted in *IEEE Annals of the History of Computing,* Vol. 5, No. 4, 1993.

[Păun '94] Păun, G. : "String functions based machines", preprint, Institute of Mathematics of the Romanian Academy, Sept. 1994.

[Păun '95] 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*, 43, 3 (1996), p. 607 - 632.

[Potter '94] Potter, J., Baker, J., Scott, S., Bansal, A., Leangsuksun, C., Asthagiri, C.: "ASC: An Associative-Computing Paradigm", *Computer*, vol. 27, No. 11, 1994, pp. 19-25.

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

[Ştefan '91] Ştefan, G., Drăghici, F.: "Memory Management Unit - a New Principle for LRU Implementation", *Proceedings of 6th Mediterranean Electrotechnical Conference*, Ljubljana, Yugoslavia, May 1991, pp. 281-284.

[Ştefan '94] Ştefan, G., Hascsi, Z.: "The Internal Structure of the Connex Memory", preprint, Center for New Electronic Architecture of the Romanian Academy, April 1994.

[Ştefan '95] Ştefan, G., Maliţa, M.: "The Eco-Chip: A Physical Support for Artificial Life Systems", *Artificial Life. Grammatical Models*, ed. by Gh. Păun, Black Sea University Press, Bucharest, 1995, pp. 260-275.

[Ştefan '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", *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, Indianapolis, April 13 - 16, 1997. p. 225-229.

[Ştefan '98] Ştefan, G., Benea, R.: "Connex Memories & Rewrieting Systems", accepted to *MELECON '98*, Tel-Aviv, May 18 -20, 1998.