FUNCTIONAL ELECTRONICS

*

Lecture Notes on Parallel Embedded Systems

(work in progress)

Gheorghe M. Ştefan

– 2018 version –

Introduction

Functional Electronics (FE) means Embedded Computation (EC), i.e., circuits and information. In the domain of FE we are interested by the High Performance FE (HPFE), which means EC as Artificial Intelligence (AI).

EC becomes increasingly dominated by parallel computation in the form of ***parallel accelerators***. Thus, PHFE emerges.

AI in its new embodiment, after the last AI winter, is based on Machine Learning (ML). It is in the top of a stack build starting from Hybrid Parallel Accelerators (HPA). In Figure 1, HPFE includes, by turn:

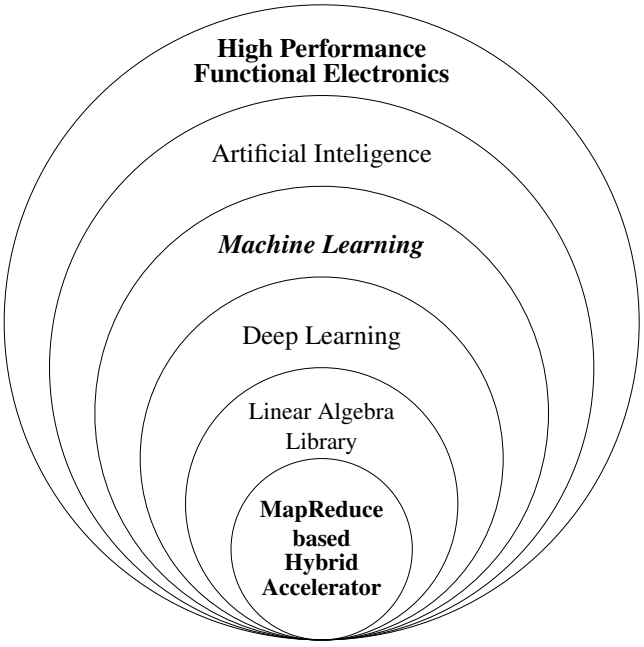

Figure 1:

- AI
- ML
- Deep Learning
- Linear Algebra Library with non-linear activating functions (unfortunately, we do not have a positive definition for non-computable functions to use them instead of the non-linear functions)
- MapReduce based HPA

What kinds of functions are considered for AI level? Mainly, the iPAL[1] functions, which can be layered [ThoughtSpot] top-down as follows:

- dynamic z-scores
- cross correlations
- regression analysis
- k-means clustering

Chapter 4 introduces, starting form the computational model proposed by Stephen Kleene, the concept of ***parallel computation***.
Chapter 5 describes the generic version of the parallel accelerator used in these lectures.
Chapter 6 presents the 13 class of problems ("dwarfs") typical for parallel accelerators.
Chapter 7 presents the dense linear algebra problems.
Chapter 8 presents the sparse linear algebra problems.

Appendix A provides the Kleene's computational model of Partial Recursive Functions and two theorems which allow us to use only the composition rule.

[1] PAL: Personal Assistant Linker

Contents

Part I

Function & Structure & Information

Chapter 1

Formal Languages

The correspondence between the formal languages and digital machines that recognize and/or generate them is a well-known subject. Noam Chomsky has established a hierarchy in formal languages. Therefore, we can ask the question: *the machines associated to each type of formal language are they belonging to a corresponding hierarchy?*

In this book we started with a developing mechanism for digital systems, generating an ordered *structural hierarchy*, and we continued associating to this structural hierarchy a *functional hierarchy*. Each new order having more autonomy accepts functional gains. We proved that the functional gain, passing from an order to the next, is given by an additional structural loop.

This functional hierarchy will lead us to emphasize a well-fitted correspondence that associates to each *language type* a *structural order*. Therefore, our main aim in this chapter is to prove the following correspondences:

1. type 3 languages - two loops machines (2-OS)
2. type 2 languages - three loops machines (3-OS)
3. type 1 languages - four loops machine (4-OS)

If the "expressiveness" of the languages grows, from 3 to 1, then the autonomy of the associated machines must also increase.

1.1 Chomsky's Generative Grammars

Noam Chomsky's papers on formal languages, starting from '50s, have founded many technical approaches in computer science, from the automata theory to the high level languages and computational linguistics. This section is devoted to introduce only the basic concepts necessary to explain the correlation between the formal languages and the associated physical structures.

Definition 1.1 *The finite set of symbols A is an alphabet and the infinite set of strings built with the symbols of A is* A^*. $\diamond$

Definition 1.2 *A language L, finite or infinite, is a sub-set of* A^*. $\diamond$

Two kind of formal languages can be specified:

1. complex formal languages, by an explicit *enumeration* of the elements of the subset L
2. simple formal languages, given by the *rules* for generating the subset L.

Obviously, the second is the best way to define a language because it gives us a concise, simple form to manipulate a big size set (frequently infinite). The *generative grammars* were introduced by Noam Chomsky in order to define and to study the properties of the programming languages. Choosing the second way the researchers decided to study only the *simple* languages having a constant sized definition. The first way is compulsory only for *complex* languages which have no rules to define them.

Definition 1.3 *A generative grammar is defined as the 4-tuple $G = (N, T, P, n_0)$ where: N is the finite set of the non-terminal symbols, T is the finite set of the terminal symbols, P is the finite set of the generation rules, or productions, by the form $p \to q$ with: $p \in (N \cup T)^*$ is a non-empty string of terminals and non-terminals having compulsory an element from N, $q \in (N \cup T)^*$; n_0 is the start symbol.* $\diamond$

Definition 1.4 *If $n_0 \to p_1 \to p_2 \to \ldots \to q$ and all the production rules used are from P of G, then we say that q is generated in G starting from n_0: $n_0 \Rightarrow q$.* $\diamond$

Definition 1.5 *The language generated by the grammar G is the set $L(G) = \{p \mid n_0 \Rightarrow p\}$.* $\diamond$

Example 1.1 *Let be the grammar:*

$$G_1 = (\{S, A\}, \{a, b, c\}, \{S \to aAa, A \to aAa \mid bAb \mid c\}, S)$$

An example of generation is:

$$\begin{gathered} S \to aAa \\ aAa \to aaAaa \\ aaAaa \to aabAbaa \\ aabAbaa \to aabaAabaa \\ aabaAabaa \to aababAbabaa \\ aababAbabaa \to aababcbabaa \end{gathered}$$

The generated strings are symmetrical, growing in two distinct points in the string. The generating process stops when no rule can be applied.

$\diamond$

Example 1.2 *The grammar G_2 is used for generating fell formed algebraic expressions.*

$$G_2 = (\{S, M, F\}, \{a, +, *, (,)\}, P, S)$$

where:

$$P = \{S \to S + M \mid M, M \to M * F \mid F, F \to a \mid (S)\ \}$$

Let us consider the expression:

$$a + a * (a + a)$$

The grammar G_2 generates it as follows:

$S \to S + M;$	
$S + M \to M + M;$	*is applied* $S \to M$
$M + M \to F + M;$	*is applied* $M \to F$
$F + M \to a + M;$	*is applied* $F \to a$
$a + M \to a + M * F;$	*is applied* $M \to M * F$
$a + M * F \to a + F * F;$	*is applied* $M \to F$
$a + F * F \to a + a * F;$	*is applied* $F \to a$
$a + a * F \to a + a * (S);$	*is applied* $F \to (S)$
$a + a * (S) \to a + a * (S + M);$	*is applied* $S \to S + M$
$a + a * (S + M) \to a + a * (M + M);$	*is applied* $S \to M$
$a + a * (M + M) \to a + a * (F + M);$	*is applied* $M \to F$
$a + a * (F + M) \to a + a * (a + M);$	*is applied* $F \to a$
$a + a * (a + M) \to a + a * (a + F);$	*is applied* $M \to F$
$a + a * (a + F) \to a + a * (a + a);$	*is applied* $F \to a$

For example, the expression

$$a + (a * +$$

can not be generated using G_2.

$\diamond$

Example 1.3 *Let be the grammar:*

$$G_3 = \{\{S,B\},\{a,b\},\{S \to aS \mid aB, B \to bB \mid b\},S\}$$

A possible generation is:

$S \to aS$
$aS \to aaS$
$aaS \to aaaB$
$aaaB \to aaabB$
$aaabB \to aaabbB$
$aaabbB \to aaabbbB$
$aaabbbB \to aaabbbb$

The language generated by this grammar is:

$$L(G_3) = \{a^n b^m \mid n,m \geq 1\}$$

$L(G_3)$ *generate a string n* a*s followed by a string of m* b*s.*
◇

The grammar from the prewious example generates strings growing at one end only.

Example 1.4 *The language:*

$$L(G_4) = \{a^n b^n \mid n \geq 1\}$$

is generated by the following grammar:

$$G_4 = \{\{S\},\{a,b\},\{S \to aSb \mid ab\},S\}$$

$L(G_4)$ *generates n* a*s followed by a the same number of* b*s.*
◇

The language generated by G_4, $L(G_4)$, is more "expressive" than the language generated by the grammar G_3, $L(G_3)$, because both generate *a*s followed byy *b*s, but G_4 satisfies an additional condition: the number of *a*s is equal with the number of *b*s.

Example 1.5

$$G_5 = \{\{S,B\},\{a,b,c\},P,S\}$$

with P containing the following productions:

$S \to aBSc \mid abc;$
$Ba \to aB;$
$Bb \to bb;$

A possible generation is:

$S \to aBSc$
$aBSc \to aBaBScc$
$aBaBScc \to aBaBabccc$
$aBaBabccc \to aaBBabccc$
$aaBBabccc \to aaBaBbccc$
$aaBaBbccc \to aaaBBbccc$
$aaaBBbccc \to aaaBbbccc$
$aaaBbbccc \to aaabbbccc = a^3b^3c^3$

It is obvious that::

$$L(G_5) = \{a^n b^n c^n \mid n \geq 1\}$$

◇

The productions in G_5 are context sensitive becuse, for example, the last production substitutes B with b only in the context of a b preceded by a B.

Example 1.6 *Let be the grammar:*

$$G_6 = \{\{S,A,B,C,D\},\{a,b\},P,S\}$$

where P contains:

$S \to CD$
$C \to aCA \mid bCB$
$AD \to aD$
$BD \to bD$
$Aa \to aA$
$Ab \to bA$
$Ba \to aB$
$Bb \to bB$
$C \to \lambda$
$D \to \lambda$

We remind that λ is the null element. The last two productions allow the disappearance of elements from the already denerated stream of symbols. A possible derivation starting from S is:

$S \to CD;$	
$CD \to aCAD;$	*s-a aplicat* $C \to aCA$
$aCAD \to abCBAD;$	*s-a aplicat* $C \to bCB$
$abCBAD \to abBAD;$	*s-a aplicat* $C \to \lambda$
$abBAD \to abBaD;$	*s-a aplicat* $AD \to aD$
$abBaD \to abaBD;$	*s-a aplicat* $Ba \to aB$
$abaBD \to ababD;$	*s-a aplicat* $BD \to bD$
$ababD \to abab;$	*s-a aplicat* $D \to \lambda$

During the generation process the string did not increases in each step.

The first two productions allow the stream to enlarge. Follow productions used to reconfigure it depending on the context. The last two productions reduce the size of the stream.

◇

1.2 Chomsky's Hierarchy of Generative Grammars

'In [1] [2] [4] Noam Chomsky a introdus conceptul de ***ierarhie a gramaticilor*** 'in func'tie de restric'tiile impuse regulilor de generare.

Definition 1.6 *A generative grammar G could be:*

regular *or* ***type 3*** *if each production in P has the form*

$$A \to xB \mid x$$

where $A,B \in N$ 'si $x \in T^*$

context-free *or* ***type 2*** *if each production in P has the form*

$$A \to \alpha$$

where $A \in N$ *'si* $\alpha \in (N \cup T)^*$

context-sensitive *or* ***type 1*** *if each production in P has the form*

$$\alpha \to \beta$$

where $\alpha, \beta \in (N \cup T)^*$ *'si* $|\alpha| \leq |\beta|$

recursively enumerable *or* ***type 0*** *if each production in P has no restrictions.*

where $|I|$ *is the length of the string I.*

◇

Between the grammars of type 1 and 0 there are, for sure, other grammars based on rules governed by weaker restrictions than those applied to the productions in type 1 grammars. We are not interested in studying them because almost all programming languages are of type 2.

Regarding to the generating rules, Chomsky emphasized three restrictions:

first restriction $|p| \leq |q|$, the length of p cannot be larger than the length of q in the production $p \to q$

second restriction : $|p| = 1$, p has length equal with one in the production $p \to q$

third restriction : $q = \alpha A$, where $\alpha \in T^*$, $A \in N \cup \{\lambda\}$, the string grows by the generative mechanism only at one end.

Definition 1.7 *The generative grammars are classified as follows:*

- *type-0 grammars, having unrestricted rules*
- *type-1 grammars, named context-sensitive grammars, having the productions limited by the* **first restriction**
- *type-2 grammars, named context-free grammars, having the productions limited by the* **first** *and* **second restriction**
- *type-3 grammars, named regular grammars, having the productions limited by the* **first**, **second** *and* **third restriction**. ◇

Definition 1.8 *The language* $L(G)$ *is a type-i language, if the grammar G is a type-i grammar, for* $i = 0, 1, 2, 3$. ◇

Definition 1.9 *The set* $\mathscr{L}_i$ *is the set of type-i languages, for* $i = 0, 1, 2, 3$. ◇

Theorem 1.1 $\mathscr{L}_0 \supset \mathscr{L}_1 \supset \mathscr{L}_2 \supset \mathscr{L}_3$.

◇

Proof 1.1 *Directly, using the Definition 1.7*

◇

An important consequence of this theorem is that a machine associated to a language in $\mathscr{L}_i$ is able to recognize or to generate any string belonging to a language in $\mathscr{L}_j$ daca $j > i$.

Example 1.7 *In the previous examples the following types of grammars could be identified:*

- $G_3 \in \mathscr{L}_3$ *because*
  $G_3 = \{\{S, B\}, \{a, b\}, \{S \to aS \mid aB, B \to bB \mid b\}, S\}$ *has only productions of form:* $A \to xB \mid x$

- $G_1, G_2, G_4 \in \mathscr{L}_2$ *because all the three grammars*
  $G_1 = (\{S,A\},\{a,b,c\},\{S \to aAa, A \to aAa \mid bAb \mid c\},S)$
  $G_2 = (\{S,M,F\},\{a,+,*,(\,)\},P,S)$ *with*
  $P = \{S \to S+M \mid M, M \to M*F \mid F, F \to a \mid (S)\}$
  $G_4 = \{\{S\},\{a,b\},\{S \to aSb \mid ab\},S\}$
  have only productions of form: $A \to \alpha$ *unde* $A \in N$ *'si* $\alpha \in (N \cup T)^*$

- $G_5 \in \mathscr{L}_1$ *because*
  $G_5 = \{\{S,B\},\{a,b,c\},P,S\}$ *cu*
  $P = \{S \to aBSc \mid abc, Ba \to aB, Bb \to bb\}$ *has productions of form* $\alpha \to \beta$ *where* $\alpha, \beta \in (N \cup T)^*$ *'si* $|\alpha| \leq |\beta|$

- $G_6 \in \mathscr{L}_0$ *because* $G_6 = \{\{S,A,B,C,D\},\{a,b\},P,S\}$ *with*
  $P = \{S \to CD, C \to aCA \mid bCB, AD \to aD, BD \to bD, Aa \to aA, Ab \to bA, Ba \to aB, Bb \to bB, C \to \lambda, D \to \lambda\}$ *has also productions of type* $A \to \lambda$ *whose application in the generating process result in reducing the length of the string.*

$\diamond$

Chapter 2

Structures & Languages

Two functions are involved in the relation between languages and machines: a string belonging to a language must be *recognized* or must be *generated*. **Recognition** and **generation** are fundamental functions in digital processing. Indeed, a string of symbols has a meaning that must be understood (recognized) and, according to the recognized meaning, an answer is computed (generated). One of the simplest *processing models* can be proposed according to these two steps:

RECOGNIZER is a digital system or a process that verifies if the input string has a meaning and recognizes that meaning

GENERATOR is a digital system or a process that, starting from the received meaning and from its own **internal state**, modifies the internal state and generates the output string.

The simplest digital system having an **internal state** is the automaton. Therefore, starting from the second order systems (2-OS) and ending with the fourth order in digital systems, the characteristics regarding recognition and generation will be analyzed in correlation with the associated formal languages.

The main formal constraint we impose in recognizing and generating languages is to use only *simple* machines, i.e., machines with constant sized definitions. For a string of n symbols, we must use a machine having a definition with the size belonging to $O(1)$, even if the size of the machine belongs to $O(f(n))$.

The small complexity, even if the system size or the input string are very large, is the key to define useful and easy to build machines. There are two theoretical types of machines:

infinite machines if $C_{Machine} \in O(g(n))$, when the input dimension is in $O(n)$

finite machines if $C_{Machine} \in O(1)$ having a constant size, independent of the input dimension (even if for an infinite input string the machine has a finite size; this being the deep meaning of the term "finite automaton").

If the complexity of machines is "infinite" it is out of our interest; we are unable to "say" anything about an "infinite" machine because the definition is useless to handle. The criteria upon which we select the useful machine is to be a finite machine, i.e., to have a constant complexity.

The previous discussion is very important because any language can be recognized and generated using any type of physical machine. We can use for all languages combinational circuits or finite automata. The theory imposes restrictions because of the efficiency of defining and building concrete machines. For example, we can design an automaton that recognizes the context free language L_2, but this automata must have a number of states in $O(n)$ for processing the strings having maximum n bits. This automaton will not be a *finite automaton*, it will be an infinite machine having a huge definition. Therefore, we associate *optimal* machines with languages only under the restriction that the machine must be simple because they have constant definitions, even the size is theoretically unbounded.

Theorem 2.1 *The formal languages generated by Chomsky's grammars and the machines that recognize and/or generate them can be optimal associated as follows:*

1. $\mathscr{L}_3$ *- finite automaton*
2. $\mathscr{L}_2$ *- push-down automata*

3. $\mathscr{L}_1$ - *linear memory bounded automata*

4. $\mathscr{L}_0$ - *Turing machines.* $\diamond$

All the textbooks prove this theorem and in the next section we will give some proofs regarding it with emphasis on the correlation between the type of a language and the number of loops closed inside the associated machine.

The aim of this chapter is to prove the consistency of the featuring mechanism of digital systems by loops using a new argument: the correspondence with another hierarchy emphasized in a related domain: Chomsky's formal languages theory. Maybe some important thing happen when a new loop is added in a digital system if it is the only way to move from a machine associated with a type of formal language toward the machine associated with a more "expressive" language in the hierarchy. Let us examine this strange effect of the correlation between the machine's *autonomy* and the *expressiveness* of the language.

2.1 Type 3 Grammars & Two Loops Machines (2-OS)

Here we prove that a *simple* (finite) digital system must have *at least two* internal loops for recognizing or generating the regular (type 3) languages. We will start reminding some basic results in formal language theory.

Theorem 2.2 *Any type-3 languages can be recognized by the final states of an initial deterministic half-automaton.*
$\diamond$

Indeed, because of the fact that the regular grammars generate only at one end of the string the "knowledge" of the automaton must refers only to the last received symbol. Therefore, the number of states can be finite because the alphabet is also finite. In order to offer supplementary information about the string some counters must be added, but a new loop is not compulsory.

Theorem 2.3 *Any type-3 language has a non-deterministic finite automaton which generates it.*
$\diamond$

For similar reasons a finite automaton is enough for generate randomly regular strings. A regular string grows only according with the last symbol generated and a randomly selected rule from which are applicable.

Now, returning to our subject, we must say something about the minimum number of loops needed for building a machine that recognizes or generates the regular languages.

Theorem 2.4 *The lowest order of a system that implements any finite automaton is two.*
$\diamond$

Proof 2.1 *We remind that finite automaton is defined by the 5-tuple $A = (X, Y, Q, f, g)$, where: X is the finite input set, Y is the finite output set, Q is the finite set of the states, $f : X \times Q \to Q$ is the state transition function and $g : X \times Q \to Y$ is the output transition function. The structure of a finite automaton (Mealy without delay) is presented in Figure 2.1, where:*

- CLC *is a combinational logic circuit that computes the transition functions f and g*
- REGISTER *is a collection of D flip-flops having a two level internal organization:*
  - Master Latch*, which is a collection of one bit latches that* store *the current state (the current value from Q)*
  - Slave Latch*, which is a latch that allows to close in a non-transparent fashion the loop over the entire system, allowing a synchronous behavior (it is avoided if an automaton is designed in the asynchronous variant).*

In the system there are two level of loops:

- *the* first loop *level in each one bit latch (from the master latch), allows the* storing *function*
- *the* second loop *level (through CLC,* Master Latch *and* Slave Latch*) is imposed by the* state transition function*, f, which is defined in $X \times \mathbf{Q}$, with values in* $\mathbf{Q}$. Slave Latch *has only an electrical role, allowing only the synchronous transition of the system under the control of the clock signal.*

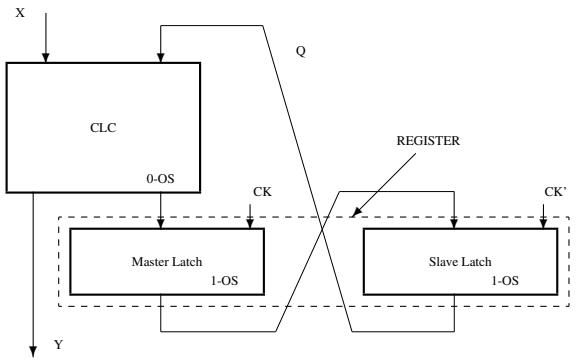

Figure 2.1: The internal structure of a Mealy automaton

◇

We can summarize saying that two levels of loops are enough to manage regular languages because:

- the first loop is used to build the circuit that *stores* the last received or generated symbol: the master latch from the state register
- the second loop, closed through register and combinational circuit, is for *sequencing* the process of recognition and generation.

No more memory is needed because the productions are very simple. The string can be recognized (understood) in "real" time because of the simple rules which generated it. The recognition process can fail before the ending of the string, because each symbol is related (correctly or incorrectly) only to the previous symbol. The finite automata are the simplest digital machines that recognize and generate regular strings. We can define a more structured simple machine, but never a less structured machine having the order 1 or 0. A 0-OS or a 1-OS can be used, but only renouncing to the simplicity.

2.2 Type 2 Grammars & Three Loops Machines (3-OS)

We are expecting that the step towards the type-2 languages, more expressive languages, should require a better, more autonomous machine to recognize or to generate them. Do it work the automata dealing with the context-free languages? Yes, they work, but not as *finite* automata. Only "infinite" automata are useful for these purposes. If we don't agree "infinite" automata, then third order systems (3-OS) must be used. An "infinite" automaton has a space state dimensioned according to the input set dimension, or according to the length of the input sequence. If we wish to use an automaton to recognize strings belonging to the second type language, then an automaton having $|Q| \in O(n)$ must be used, where: n is the length of the string and $|Q|$ is the number of states. Our aim is to investigate only the finite, simple machines and in this respect we must find a solution having constant complexity.

Let us start with a short discussion about the classical example offered by the language $\{a^n b^n | n > 0\}$. If we want to recognize this language using a half-automaton, then the problem raised is to know what is the number of a's received before the first b. The machine must memorize somewhere the number of received symbols having the value a. The only place for an automaton is in the "state space", but in this case the automaton becomes an "infinite" machine. The solution to maintain the machine in the limit of the simple machines is to add a kind of memory to "count" and "memorize" the number of a's in order to compare it with the number of bs. Instead of an automaton is better to use the machine represented in Figure 2.2, where the reversible counter counts up the received as and counts down for each received b. Thus the *finite* automaton helped by the counter (an "infinite" but simple automaton) solves the problem.

The counter is a very simple "memory", but has the inconvenient that forgets in the "reading" process. Another inconvenient is its loss of generality. A more general memory is the *stack memory*. It also forgets by reading but it is able to

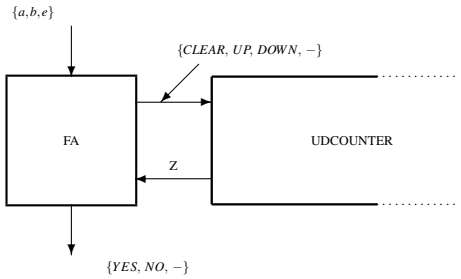

Figure 2.2: Finite automaton with counter - a 3-OS that recognizes the language $\{a^n b^n | n > 0\}$

store the received string. Let us remember the push-down automata presented in 5.3.1 and Example 5.3 where the family of strings recognized belonged to a type-2 language. For general situations the next well known theorem works.

Theorem 2.5 *All type-2 languages can be recognized by the final state of a* push-down automaton *(PDA) (see Definition 5.1).*

◇

The main remark is that a PDA is a small and simple machine because the automaton is a finite machine and the stack is an infinite, recursive defined machine.

Theorem 2.6 *Any type-2 language are generated by a non-deterministic push-down automata.*

◇

And now, what is the main difference between a finite automaton and a PDA? What is the main step done in order to have a machine that recognizes or generates type-2 languages?

Theorem 2.7 *The lowest order of a system that implements a push-down automaton is 3.*

◇

Proof 2.2 *Because the push-down automata is build using a finite automaton loop coupled with a push-down stack (see Chapter 5 and Figure* **??***), then it is a third order system. Indeed, a finite automaton is a second order system and the push-down stack has the same order because it is an "infinite" automaton (the stack implementation implies a reversible counter serially composed with a RAM). The third loop through the stack has the role to memorize "the number n", to memorize the additional relation between the elements of the generated string (in our simple example the stack memorizes the value n).*

◇

The stack is the simplest memory device because:

1. stores only strings
2. has the access only to one end of the string (*last - in first - out*)
3. the read operation is destructive (the memory forgets the read information because of the access type).

The simplicity is the reason for using this memory in order to build the first machine a little more complex than a finite automaton. The first step beyond the automata level is made by PDA. But the same simplicity is also the reason for which we must renounce to this memory if we want to approach the next type of languages. For the next step we need a memory in which we can access many times the same stored content. We need a memory who does not forget when it remembers. Recognizing or generating the context dependent languages will implie to search for some substrings many times, in order to evaluate the context for different received or generated symbols.

2.3 Type 1 Grammars & Four Loops Machines (4-OS)

Let's try to solve the recognition of a language from $\mathscr{L}_1$ using an automaton! Even an "infinite automaton". After two minutes of thinking my conclusion is to leave this pleasure to other people ... . More chances we have with a pushdown automaton, but this solution implies also an "infinite" number of states for the automaton. It is evident the necessity to make the next step in introducing a new feature for the recognizing/generating machine.

For example when we try to build a machine associated to the language $\{a^n b^n c^n | n > 0\}$ we must add a supplementary device. Indeed, if we try to use a PDA for recognizing this language we will be in impossibility to finish our work because after reading the a's from the stack the information about n will be lost and we need this information for "counting" the c's. We must add something to compensate this disfunctionality. We must think to add a new reversible counter. But, this solution leads us toward the *third loop*.

In the general case we can use for $\mathscr{L}_1$ a *finite defined machine* (a machine with $C_{Machine}(n) \in O(1)$) only by adding, to the push-down stack automaton, a new push-down stack to make a back-up for each symbol read from the first stack. In this case a new loop is closed in the machine and it becomes a *four order system* (4-OS). The new stack compensates the limit of the stack memory that forgets by the reading.

The *third loop* of the system is necessary because it gives us access to a new external memory. This additional memory was imposed because a restriction that acts on the productions that define the grammars has been removed. But, the effect of the additional memory can be substituted if the machine should be equipped with a memory having more features: the *linear bounded memory*.

Definition 2.1 *The* linear bounded automaton *(LBA) is a finite automaton (FA) loop connected with a linear bounded memory (see Figure 2.3) that performs in each cycle the following sequence of operations:*

1. *generates to the output DOUT the content of the current accessed cell*
2. *stores to the current accessed cell the symbol applied on the input DIN*
3. *changes the accessed cell with the next right cell (UP) or the previous left cell (DOWN), or maintains the same accessed cell (-) (working like a bi-directional list memory). The formal definition of the LBA is:*

$$LBA = (I \cup \{\#\},\ Q, f; q_0)$$

where: $I \cup \{\#\}$ is the finite alphabet of the machine, Q is the finite state set, $q_0 \in Q$ is the initial state of the automaton and f is the transition function of the entire machine:

$$f = (I \cup \{\#\}) \times Q \to (I \cup \{\#\}) \times Q \times \{UP,\ DOWN,\ -\}$$

with the very important restriction: the symbol # is prohibited to be substituted. In each state, starting from the symbol read from memory and from the state of the automaton, a new symbol is written back into the memory, the automaton switches in a new state and the cell for the next state is selected. In the initial state of the machine the automaton is in q_0, the memory contains the string to be processed limited on both ends by # and the first symbol from the string is accessed.

◇

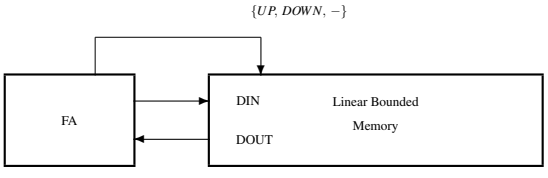

Figure 2.3: Linear Bounded Automata

Using the machine just defined the context dependent language were studied from the point of view of the machine that recognizes or generates it.

Theorem 2.8 *The context-sensitive languages (type-1 languages) are recognized only by the final states of a linear bounded automata.*

◇

If the string to be recognized is in a memory in which after reading a symbol it can be written back, then it can be inspected many times in order to perform a more complex recognizing process.

Theorem 2.9 *The context-sensitive languages are generated only by the machines that are at least linear bounded automata.*

◇

The possibility to re-memorize suggests us a new loop.

Theorem 2.10 *The lowest order of a system that implements a linear bounded memory automaton is 4.*

◇

Proof 2.3 *The simplest memory having non-destructive reading can be made by loop connecting two push-down stack memories. For each POP, from the initial stack, a PUSH with the same symbol or another, in the added stack, is performed. For each POP from the added stack, a corresponding PUSH can be made in the initial stack. Thus, these two stacks perform the functions of a memory which doesn't forget when reading. The sizes of each stack can be linearly bounded to the string's length. Thus, the two stacks simulate a bi-directional list.*

Because a push-down stack is a 2-OS, a memory with non-destructive reading is a 3-OS (made by loop connecting two stacks) and a linear bounded automaton is a 4-OS (see Figure 2.4).

◇

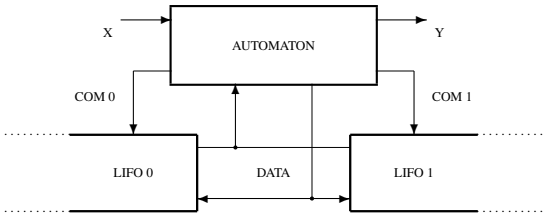

Figure 2.4: Push-down automaton with an additional stack memory

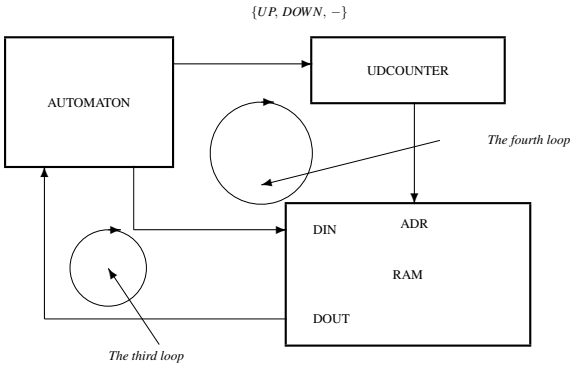

Figure 2.5: Automaton with Linear Bounded Memory

An equivalent structure for *LBA* is presented in Figure 2.5, where:

- $AUTOMATON$ is a finite automaton (a 2-OS)
- $UDCOUNTER$ is an "infinite" automaton, having a simple structure ($C_{UDCOUNTER} \in O(1)$, even the size is $S_{UDCOUNTER} \in O(logn)$), used to point a symbol in memory
- *RAM* is a random access memory for storing the string (a first order system)

This structure has two loops over a finite automaton. Therefore, it is also a 4-OS. The structure is more complex but the size is minimal. Instead of the previous solution, in which the content "moves" in front of the automaton, now the content of the memory is pointed by the content of an up-down counter. Now the pointer moves and the content is stable in RAM.

The hardware requirement for context-sensitive languages implies a more structured and a more functional segregated machine. This machine has two supplementary loops added to an automaton with two distinct roles:

- the first, through *RAM*, for accessing an external *memory support*
- the second, through $UDCOUNTER$ and *RAM*, for accessing an external *memory function*: a *bi-directional scanned list*.

The *list* can do more than the *stack*. Both are strings but the second allows only a limited and destructive access to the content of the string. In a memory hierarchy the list has a higher order because it is equivalent (sometimes it is implemented so) by two loop-coupled stacks.

2.4 Type 0 Grammars & Turing Machines

The computational model of the Turing machine is responsible, together with Kleene's model, for the (too) strong imposed *von Neumann architecture* [von Neumann '45] of the actual computers.

Definition 2.2 *Turing Machine (TM) is a finte automaton (FA) loop connected with an infinite memory (Figure 2.6). The automaton performs in each cycle the following sequence of operations:*

1. *receives from the output DOUT the content of the current accessed cell in the memory*
2. *stores to the current accessed cell the symbol generated, on the input DIN, according with the own state and with the received symbol*
3. *changes the accessed cell with the next right (UP) or next left (DOWN) cell, or maintains the same accessed cell (-).*

The formal definition of the TM is:

$$TM = (I,\ Q, f; q_0)$$

where: I is finite alphabet of the machine, Q is the finite state set, $q_0 \in Q$ is the initial state of the automaton and f is the transition function of the entire machine:

$$f = I \times Q \to I \times Q \times \{UP,\ DOWN,\ -\}.$$

In each state, starting from the symbol read from the memory and from the state of the automaton, a new symbol is written in the memory, the automaton switches in a new state and the cell for the next state is selected. In the initial state the automaton is in q_0, the memory contains the string to be processed ended on both ends by $\# \in I$, the selected symbol from the string is the first symbol.

◇

A detailed structure of TM is presented in Figure 2.7 for emphasizing its three main components:

1. the finite automaton (FA)
2. the infinite automaton that is the reversible counter, UDCOUNTER, a simple recursive defined device
3. *Infinite RAM* (also a simple recursive defined structure) addressed by UDCOUNTER.

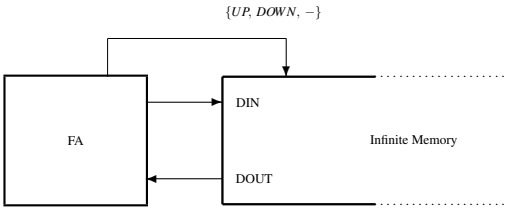

Figure 2.6: The Turing Machine

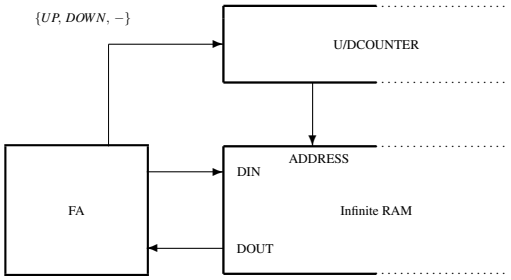

Figure 2.7: The structure of a Turing Machine

Theoretically, the *Infinite RAM* and UDCOUNTER are both more than two *"infinite" machines* because they must be in fact infinite. Therefore, TM is not a real machine and we can not classify it as a digital system; we can not discuss about the order of TM.

Type-0 grammars and the associated languages are characterized with rules having no restrictions. The last restriction being avoided (the string length can not be reduced in any step of the generative processes), the memory space cannot be evaluated before the process of generating or recognizing the string (in the generative process the string can reach an unpredictable length). Therefore the memory must be theoretically unlimited.

The formal language theory is centered on the context free (type-2) languages because these are the most used programming languages. Therefore, it is enough to study the languages that border the context free languages, i.e., regular languages and context dependent languages. Languages simpler that the regular language and, in the same time useful, maybe do not exit. But a question rises: *are there languages between type 1 languages and type 0 languages?* In other words, is there a less restrictive condition that the restriction imposed to the context sensitive language? The answer to this question should become important if we will need formal languages more less restrictive (or more "expressive") than the current ones.

2.5 Universal Turing Machine: the Simplest Structure

Is TM a simple or a complex machine? The complexity of TM is given by the complexity of the finite automaton because this part of the machine is actualized for each distinct problem. The finite automaton contains the single *random* structure from a TM: the combinational circuit that closes the loop of the automaton. We will prove that the structural complexity of TM can be reduced only transforming it in the Universal Turing Machine (UTM).

Early theoretical studies where devoted to reduce the number of states of the finite automaton with a minimal increasing of the number of symbols in the alphabet I [Shannon '56]. In this approach the complexity of the finite automaton increases very much. But we believe that, instead of reducing the number of states, the more important thing is to reduce the structural complexity of UTM. In this respect we will present the simplest UTM built only with recursive defined circuits.

The problem is to define a machine whose structure can remain unchanged when the executed function changes. In this case we need a machine with:

- an abstract representation for the needed TM, as a string of symbols stored in the memory
- an automaton, useful for all computable functions, that "understands" and "executes" by *interpretation* the abstract representation, stored on the tape, of any automaton associated to a TM.

Interpretation is a process that uses a string encoded representation of an abstract machine, to emulate the behavior of that machine. It allows us to deal with *representations* of machines rather than with the machine themselves.

Let be a machine M with the initial content of the tape T: $M(T)$. An interpreter of $M(T)$ will be the machine

$$U(< e(M), T >)$$

where $e(M)$ is the string that describes the machine M. On the tape of the machine U there is the description of M and the string, T, to be processed by the machine M.

Definition 2.3 *An UTM is a TM, $U(< e(M), T >)$, that has a finite automaton that interprets any TM's description, $e(M)$, stored in the same memory with the string, T, to be processed.*

◇

In order to implement an UTM we start from the fact that the transition function f from the state q_i can be reduced to a set of the pair of transitions having the next form:

$$f(q_i, out\ of\ RAM = x) = f(q_i, x) = (q_j, y, c_l)$$

$$f(q_i, out\ of\ RAM \neq x) = f(q_i, \neq x) = (q_k, z, c_m)$$

where: $q_i, q_j \in Q$, $x, y, z \in I$, and $c_l, c_m \in \{UP, DOWN, -\}$ having the following meaning:

if out of RAM=x
  then the next state is q_j
    the stored symbol is y
    the access head command is c_l
  else the next state is q_k
    the stored symbol is z
    the access head command is c_m

Each such a pair will be associated with a state of the automaton. Therefore, any state can be represented as a string of nine symbols having the form:

$$\&, q_i, x, q_j, y, c_l, q_k, z, c_m$$

where & is a symbol that points the beginning of the string associated with the state q_i.

A TM can be completely described by specifying the function f, associated to the *random structure* of the machine, using the above defined strings to compose a "program" P.

The tape of UTM will be divided in two sections, one for the string T to be processed by the machine M, and one containing the description P of the machine M. The content of the tape will be $\ldots\#P@T\#\ldots$ where:

- @ is a special symbol which delimits the "program" from the "data"
- the string $P \in (I \cup Q \cup \{DOWN, UP, -\} \cup \{\&\})^*$ is the "program" that describes the algorithm
- the string $T \in I^*$ represents the "data".

The automaton of UTM "knows" how to interpret the string P in order to process the string T. It is the only random structure in UTM. The question is: what are the possibilities to minimize this random structure in UTM? The answer is: performing a strong *functional segregation*.

For simplicity, we will use a TM having two tapes (**the first segregation!**), one for P and one for T. This machine has an actual implementation using a RAM with two ports for *read* and a port for *write*.

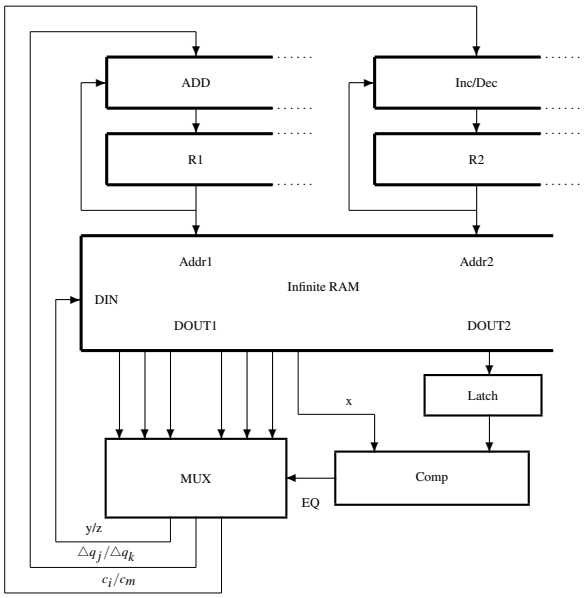

Figure 2.8: The structure of a recursive defined Universal Turing Machine

The previous form of P must be *translated* in P' that uses for each state, instead of the string $\&, q_i, x, q_j, y, c_l, q_k, z, c_m$ stored in 9 successive memory cells, the next form, as a single entity stored in one cell:

$$x, \triangle q_j, y, c_l, \triangle q_k, z, c_m$$

where: $\triangle q_j$ and $\triangle q_k$ represents the distance in memory between the current location and the locations that store the descriptions for the states q_j and q_k. Each program P has a P' form (this is the premise for **the second segregation**!).

The structure of UTM in the most segregated form is presented in Figure 2.8, where the counters are detailed and some simple combinatorial circuits are added. The program P' is stored in RAM starting with a certain address n, where the description of the state q_0 is loaded. In the following cells are stored the descriptions for $q_1, \ldots$. The string to be processed is stored starting with a certain address m, greater than the address in which the symbol @ is. The initial value of the first address "counter" (ADD & $R1$) is n, and for the second counter (Inc/Dec & $R2$) is m. The multiplexer MUX selects (see Figure 2.8), according to the output of $Comp$, the appropriate values for:

- the value (y or z) to be written in RAM to the current address generated by Inc/Dec & $R2$ (the value to be written on the tape in the current cycle of the simulated TM)
- the signed number to be added to the current value of "program counter" implemented by ADD & $R1$ (the relative address of the cell that stores the description of the next state: the next "instruction")
- the command applied to the counter (Inc/Dec & $R2$) that points in the data part of the tape

(The latch connected to DOUT2 has only an electrical role, avoiding the transparency on the loop closed through the RAM built by latches. If the RAM would have been built with master-slave flip-flops (a possible, but a very inefficient solution) the latch on DOUT2 output is not necessary.)

The strong functional segregation in UTM implies a machine with *no random circuits*. The randomness of the machine is totally shifted in the content of the tape (memory), where a "random" string describes an algorithm. Instead of random circuits we have random string of symbols. The *hard* random structure of the circuits is converted to the *soft* random structure of the string describing the function executed by the machine.

An UTM implemented in a variant with functional segregation emphasizes the fact that the relation between the *recursive* part and the *random* part is the same as the actual relation between the *hard* part and the *soft* part of a computer system.

In this last UTM variant the *interpretation* of T is substituted with the *execution* of T. The interpretation is a controlled process that involves a finite automaton. The execution is made by simple circuits (in this case, combinational). *Comp*, MUX, ADD, *Inc/Dec* are simple circuits that execute. **Removing the finite automaton** from the structure of UTM the machine substitutes the *interpretation* of P with the *execution* of P.

In order to use only the simplest structure for implementing the machine associated with any formal language it is evident that the best solution is UTM. The random part of its structure can be null. It is the time for a new theorem.

Theorem 2.11 The simplest physical structure of a machine that recognizes/generates a formal language is the physical structure of a 0-state UTM that executes, using only combinational circuits, the "program" P instead of interpreting P using a finite automaton. ◇

Proof 2.4 *There is no random part in UTM. The combinational circuit of a finite automaton is random and all the machines previously associated to formal languages contain at least a finite automaton. Therefore, only UTM is completely built with recursive defined circuits.* **The finite automaton is avoided** *and the interpretation is substituted with the execution.*

◇

2.5.1 The Halting Problem: the Price for Simplicity

The complexity of U depends only on the algorithmic complexity of the string $e(M)$. The structural complexity is converted in the complexity of the symbolic description of the computation that will be interpreted or executed in UTM. A *hard* complexity is converted into a *soft* complexity even for the problem having solutions with a less powerful machine (such as finite automata, PDA as LBA). What is the price for translating the complexity in a *soft* modeled space? The price is, at least, the unsolvability of the *Halting Problem* (HP).

HP is one of the most important problems that arise in computability. Let be a machine $M(T)$ having the tape content T (a program, M, and an input data, T). The question is: the machine does stops after a finite number of cycles or does not stop? The halting function sould be computed by another TM, named H, that returns 1 **if** M with initial content of tape T stops, **else** returns 0:

$$H(< e(M), T >) = 1 \textbf{ if } M(T) \textit{ halts}$$

$$H(< e(M), T >) = 0 \textbf{ if } M(T) \textit{ runs forever.}$$

Theorem 2.12 *The function $H(< e(M), T >)$ is uncomputable.* ◇

Proof 2.5 *Assume that the TM H exists for* any *encoded machine description and for* any *input tape. We will define an* effective *TM G such that for any TM F, G halts with the tape content $e(F)$ if $H(< e(F), e(F) >) = 0$ and runs forever if $H(< e(F), e(F) >) = 1$. G is an effective machine because it involves the function H and we assumed that this function is computable.*

Now consider the computation $H(< e(G), e(G) >)$ (G halts or not, running on its own description).

If $H(< e(G), e(G) >) = 1$, **then** *the computation of $G(e(G))$* halts, **but** *starting from the G's definition $G(e(G))$ the computation halts only* **if** $H(< e(G), e(G) >) = 0$. *Therefore,* **if** $H(< e(G), e(G) >) = 1$, **then** $H(< e(G), e(G) >) \neq 1$.

If $H(< e(G), e(G) >) = 0$, **then** *the computation of $G(e(G))$* runs forever, **but** *starting from the G's definition $G(e(G))$ the computation runs forever only* **if** $H(< e(G), e(G) >) = 1$. *Therefore,* **if** $H(< e(G), e(G) >) = 0$, **then** $H(< e(G), e(G) >) \neq 0$.

The application of function H to the machine G and its description generates a contradiction. Because H is defined to work for any machine description and for any input tape, we must conclude *that the initial assumption is not correct and H is not computable. [?]*

◇

The price for structural simplicity is the limited domain of the computable. See also the minimalization rule in the previous chapter as an example illustrating the HP.

Let us remember the Theorem 2.1 that proves that circuits compute *all* the functions. UTM is limited because it does not compute at least HP. But the advantage of UTM is that the computation has a finite description instead of the circuits that are huge and complex. Circuits are *complex* while the algorithms for TMs are *simple*. *But, the price for the simplicity is the incompleteness*.

2.6 Conclusions

Thesis: The actual structure evolved toward simplicity. In this respect we can promote a thesis:

> **Thesis**: **Digital machines that recognize and generate formal languages can be "infinite" (big sized) machines but must have finite definitions (small complexity)**.

The most important conclusion of this chapter is that there exists a correspondence between:

- $\mathscr{L}_3 \leftrightarrow \mathbf{2-OS}$
- $\mathscr{L}_2 \leftrightarrow \mathbf{3-OS}$
- $\mathscr{L}_1 \leftrightarrow \mathbf{4-OS}$

Turing machine and zero type languages don't have an associated order in structural hierarchy, because the Turing machine is only a theoretical model.

Between context-sensitive languages and zero type languages there are many other types of languages, corresponding to a less restricted production of their grammars. Until now, these languages are out of our interest because most of the programming languages are context-free. Systems having the order more than 4 are, maybe, associated to these hypothetical languages.

The initial evolution of the machine converted the *hardware* complexity into the *software* complexity. Nowadays VLSI technologies can build big sized circuits only if they are simple.

> **The complexity cannot grow with the same speed as the size**.

We must avoid the growing of the complexity in order to built very large circuits, or we must find other ways to make computations. A large complex system has only the chance to balance between *chaos* and *(partial) order* by **self-organizing** processes.

Chapter 3

Loops & Information

One of the most used scientific term is *information*, but we still don't know a wide accepted definition of it. Shannon's theory shows us only *how to measure* information not *what is* information. Many other approaches show us different, but only particular aspects of this full of meanings word used in sciences, in philosophy or in our current language. Information shares this ambiguous statute with others widely used terms such as *time* or *complexity*. Time has a very rigorous quantitative approach and in the same time nobody knows *what the time is*. Also, complexity is used with so many different meanings.

In the *first section* of this chapter we will present three points of view regarding the information:

- a brief introduction of Claude Shannon's definition about what is the *quantity of information*
- Gregory Chaitin's approach: the well known *algorithmic information theory* which offers in the same time a quantitative and a qualitative evaluation
- Mihai Drăgănescu's approach: a *general information theory* built beyond the distinction between artificial and natural objects.

We explain information, in the *second section* of this chapter, as a consequence of a structuring processes in digital systems; this approach will offer only a qualitative image about information as *functional information*.

Between these "definitions" there are many convergences emphasized in the *last section*. I believe that for understanding what is information in computer science these definitions are enough and for a general approach Drăgănescu's theory represents a very good start. In the same time only the scientific community is not enough for validating such an important term. But, maybe a definition accepted in all kind of communities is very hard to be discovered or to be constructed.

3.1 Definitions of Information

3.1.1 Shannon's Definiton

The start point of Shannon was the need to offer a theory for the communication process [Shannon '48]. The information is associated with a *set of events* $E = \{e_1, \ldots, e_n\}$ each having its own probability to come into being $p_1, \ldots, p_n$, with $\sum_{i=1}^{n} p_i = 1$. The quantity of information has the value

$$I(E) = -\sum_{i=1}^{n} p_i log\ p_i$$

.

This quantity of information is proportional with the non-determining removed when an event e_i from E occurs. $I(E)$ is maximized when the probabilities p_i have the same value, because if the events are equally probable any event remove a

big non-determining. This definition does not say anything about the information contained in *each* event e_i. The measure of information is associated only with the set of events E, not with each distinct event.

And,the question remains: what is *Information*? Qualitative meanings are missing in Shannon's approach.

3.1.2 Algorithmic Information Theory

Premises

All big ideas have many starting points. It is the case of *algorithmic information theory* too. We can emphasize three origins of this theory [Chaitin '70]:

- Solomonoff's researches on the inference processes [Solomonoff '64]
- Kolmogorov's works on the string complexity [Kolmogorov '65]
- Chaitin's papers about the length of programs computing binary strings [Chaitin '66].

Solomonoff's researches on prediction theory can be presented using a small story. A physicist makes the next experience: observes at each second a binary manifested process and records the events as a string of 0's and of 1's. Thus obtains an n-bit string. For predicting the $(n+1)$-th events the physicist is driven to the necessity of a *theory*. He has two possibilities:

1. studying the string the physicist *finds* a pattern periodically repeated, thus he can predict rigorously the $(n+1)$-th event
2. studying the string the physicist doesn't find a pattern and can't predict the next event.

In the first situation, the physicist will write a scientific paper with a new theory: the "formula" just discovered, which describes the studied phenomenon, is the pattern emphasized in the recorded binary string. In the second situation, the physicist can publish only the whole string as his own "theory", but this "theory" can't be used to predict anything. When the string has a pattern a formula can be found and a theory can be built. The behavior of the studied reality can be *condensed* and a concise and elegant formalism comes into being. Therefore, there are two kinds of strings:

- patternless or *random* strings that are incompressible, having the same size as its shortest description (i.e., the complexity has the same value as the size)
- compressible strings in which finite substrings, the patterns, are periodically repeated, allowing a shortest description.

Kolmogorov's work starts from the next question: *Is there a qualitative difference between the next two equally probable 16 bits words*:

0101010101010101

0011101101000101

or there does not exist any qualitative difference? Yes, there is, can be the answer. However, what is it? The first has a well-defined generation rule and the second seems to be random. An approach in the classical probability theory is not enough to characterize such differences between binary strings. We need, about Kolmogorov, some additional concepts in order to distinguish the two equally probable strings. If we use a fair coin for generating the previous strings, then we can say that in the second experience all is well, but in the first - the *perfect* alternating of 0 and of 1 - something happens! A strange *mechanism*, maybe an *algorithm*, controls the process. Kolmogorov defines the *relative complexity* (now named *Kolmogorov complexity*) in order to solve this problem.

Definition 3.1 *The complexity of the string x related to the string y is*

$$K_f(x|y) = min\{|p| \mid p \in \{0,1\}^*, f(p,y) = x\}$$

where p is a string that describes a procedure, y is the initial string and f is a function; $|p|$ is the length of the string p. $\diamond$

The function f can be a Universal Turing Machine (says Gregory Chaitin in another context but solving in fact a similar problem) and the relative complexity of x related to y is the length of the shortest description p that computes x starting with y on the tape. Returning to the two previous binary strings, the description for the first binary string can be shorter than the description for the second, because the first is built using a very simple rule and the second has no such a rule.

Theorem 3.1 *There is a partial recursive function f_0 (or an Universal Turing Machine) so as for any other partial recursive function f and for any binary strings x and y the following condition is true:*

$$K_{f_0}(x|y) \leq K_f(x|y) + c_f$$

where c_f is a constant.

◇

Therefore, always there exist a function that generates the *shortest* description for obtaining the string x starting from the string y.

Chaitin's approach starts by simplifying Kolmogorov's definition and by sustiruting with a machine the function f. The teen-eager Gregory Chaitin was preoccupied to study the minimum length of the programs that generate binary strings [Chaitin '66]. He substitutes the function f with a *Universal Turing Machine*, M, where the description p is a *program* and the starting binary string y becomes an *empty string*. Therefore Chaitin's complexity is:

$$C_M(x) = min\{|p| \mid p \in \{0,1\}^*, M(p) = x\}.$$

Chaitin's Definition for Algorithmic Information Content

The definition of algorithmic information content uses a sort of Universal Turing Machine, named M, having some special characteristics.

Definition 3.2 *The machine M (see Figure 3.1) has the following characteristics:*

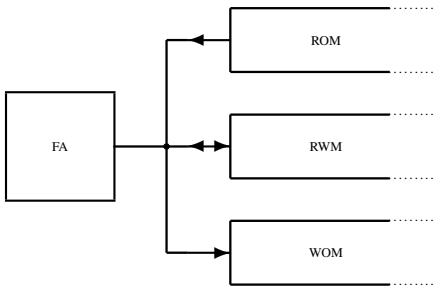

Figure 3.1: The machine M

- *three tapes (memories) as follows:*
  - *a* **read-only program tape** *(ROM) in which each location contains only* 0*'s and* 1*'s, the access head can be moved only in one direction and its content cannot be modified*
  - *a* **read-write working tape** *(RAM) containing only* 0*'s and* 1*'s and* blanks*, having an access head that can be moved to the left or to the right*
  - *a* **write-only output tape** *(WOM) in which each location contains* 0*,* 1 *or* comma*; its head can be moved only in one direction*
- *a finite state strict initial automaton performing eleven possible actions:*
  - **halt**

- **shift** *the work tape to the* left *or to the* right *(two actions)*
- **write 0,1** *or* **blank** *on the read-write tape (three actions)*
- **read** *from the current pointed place of the program tape, write the read symbols on the work tape in the current pointed place and move one place the head of the program tape*
- **write comma, 0** *or* **1** *on the output tape and move one position the access head (three actions)*
- *consult an* **oracle** *enabling the machine M to chose between two possible transitions of the automaton.*

The work tape and the output tape are initially blank. The programming language L associated to the machine M is the following:

```
<instruction> ::= <length of pattern><number of cycles><pattern>
<length of pattern> ::= <1-ary number>
<number of cycles> ::= <1-ary number>
<pattern> ::= <binary string>
<1-ary number> ::= 01 | 001 | 0001 | 00001 | ...
<binary string> ::= 0 | 1 | 00 | 01 | 10 | 11 | 000 | 001 | ...
```

The automaton of the machine M interprets programs written in L and stored on the read-only program memory. ◇

The machine M was defined as an *architecture* because besides structural characteristics it has also defined the language *L*. This language is a very simple one having only theoretical implications. The main feature of this language is that it generates programs using only binary symbols and each program is a self-delimited string (i.e., we don't need a special symbol for indicating the end of the program). Consequently, each program has associated an easy to compute probability to be generated using many tosses of a fair coin.

Example 3.1 *If the program written in L for the machine M is:*

000100000101

then the output string will be:

01010101.

Indeed a pattern having the length 2 (the first 4 bits: 0001) is repeated 4 times (the next 6 bits: 000001) and this pattern is 01 *(the last 2 bits:01).* ◇

Using this simple machine Chaitin defines the basic concepts of algorithmic information theory, as follows.

Definition 3.3 *The* algorithmic probability P(s) *is the probability that the machine M eventually halts with the string* s *on the output tape, if each bit of the program results by a separate toss of an unbiased coin (the program results in a random process).* ◇

Example 3.2 *Let be the machine M. If m is the number of cycles and n is the length of the pattern, then:*

$$P(s) = 2^{-(m+2n+4)}.\diamond$$

Definition 3.4 *The* algorithmic entropy *of the binary string* s *is* $H(s) = -log_2 P(s)$. ◇

Now we are prepared to present the definition of the *algorithmic information*.

Definition 3.5 *The* bf algorithmic information *of the string* s *is* $I(s) = min(H(s))$*, i.e. the shortest program written for the best machine.* ◇

In this approach the machine complexity or the machine language complexity does not matter, only the length of the program measured in number of bits is considered.

Example 3.3 *What is the algorithmic entropy of the two following strings: s_1 a patternless string of n bits and s_2 a string of n zeroes? Using the previous defined machine M results: $H(s_1) \in O(n)$ and $H(s_2) \in O(n)$.*

The question of Kolmogorov remains unsolved because the complexity of the strings *seems* to be the same. What can be the explanation for this situation? It is obvious that the machine M is not performant enough for making the distinction between the complexity of the strings s_1 and s_2. A new better machine must be built.

Definition 3.6 *The machine M from the previous definition becomes M' if the machine language becomes L', defined starting from L modifying only the first line as follows:*

```
<instruction> ::= <the length of <pattern length> in 1-ary>
                  <pattern length in binary>
                  <the length of <number of cycles> in 1-ary>
                  <number of cycles in binary>
                  <pattern> ◇
```

The complexity of the machine M' is bigger than that of the machine M because it must *interpret* programs written in L' that are more complex than programs written in L. Using the machine M' more subtle distinctions can be emphasized in the set of binary strings. Now, we can take back the last exemplu trying to find a difference between the complexity of the strings s_1 and s_2.

Example 3.4 *The program in L' that generates in M' the string s_1 is:*

$$\underbrace{00...0}_{\lceil log_2 n \rceil} 01 \underbrace{1XX...X}_{\lceil log_2 n \rceil} 0011 \underbrace{XX...X}_{n}$$

and for the string s_2 is:

$$0011 \underbrace{00...0}_{\lceil log_2 n \rceil} 01 \underbrace{1XX...X}_{\lceil log_2 n \rceil} 0$$

where $X \in \{0,1\}$. Starting from these two programs written in L' the entropy becomes: $H(s_1) \in O(n)$ and $H(s_2) \in O(\log n)$. Only this new machine makes the difference between a random string and a "uniform" string. ◇

Can we say that $I(s_1) \in O(n)$ and $I(s_2) \in O(log\ n)$? I yes, we can.

Theorem 3.2 *The minimal algorithmic entropy for a certain n-bit string is in $O(\log n)$.* ◇

Proof If the simplest pattern has the length 1, then only the length of the string depends on n and can be coded with $log_2 n$ bits. ◇

According to the algorithmic information theory the amount of information contained in an n-bit binary string has not the same value for all the strings. The value of the information is correlated with the *complexity* of the string, i. e., with the degree of his internal "organization". The complexity is minimal in a high *organized* string. For a quantitative evaluation we must emphasize some basic relationships.

Chaitin extended the previous defined concepts to the *conditioned* entropy.

Definition 3.7 *$H(t|s)$ is the entropy of the process of the t string generation conditioned by the generation of the string s.*◇

We can write: $H(s,t) = H(t|s) + H(s)$, where $H(s,t)$ is the entropy of the string s followed by the string t, because: $P(t|s) = \frac{P(s,t)}{P(s)}$.

Theorem 3.3 $H(s) \leq H(t,s) + c$, $c \in O(1)$.◇

Proof The string s can be generated using a program for the string(t,s) *adding* a constant program as a prefix.◇

Theorem 3.4 $H(s,t) = H(t,s) + c$, $c \in O(1)$. ◇

Proof The program for the string (t,s) can be converted in a program for (s,t) using a constant size program as prefix.$\diamond$

Theorem 3.5 $H(s,t) \leq H(s)+H(t)+c$, $c \in O(1)$.$\diamond$

Proof The "price" of the concatenation of two programs is a constant length program.$\diamond$

Theorem 3.6 $H(t|s) \leq H(t)+c$, $c \in O(1)$.$\diamond$

Proof By definition $H(t|s) = H(s,t) - H(s)$ and using the previous theorem we can write: $H(t|s) \leq H(s)+H(t)+c - H(s) = H(t)+c$, where $c \in O(1)$.$\diamond$

Definition 3.8 *A string s is said to be* random *when $I(s) = n+c$, where n is the length of the string s and $c \in O(1)$.*$\diamond$

Theorem 3.7 *For most of n-bit strings s the algorithmic complexity (information) is: $H(s) = n+H(n)$; or most of the n bits strings are random.* $\diamond$

Proof Each n-bit string has its own distinct program. How many distinct programs have the shorted length $n+H(n)+c-k$ related to the programs having the length $n+H(n)+c$ (where $c \in O(1)$)? The number of the short programs decreases by 2^k. That is, if the length of the programs decreases linearly, then the number of the distinct programs decreases exponentially. Therefore, most of n bits strings are random. $\diamond$

This is a tremendous result because it tells us that almost all of the real processes cannot be condensed in short representations and, consequently, they can not be manipulated with formal instruments or in formal theories. In order to enlarge the domain of formal approach, we must "filter" the direct representations so as the insignificant differences, in comparison with a formal, compact representation, to be eliminated.

Another very important result of algorithmic information theory refers to the complexity of a theorem deduced in a formal system. The axioms of a formal system can be represented as a finite string, also the rules of inference. Therefore, the complexity of a theory is the complexity of the string that contains its formal description.

Theorem 3.8 *A theorem deduced in an axiomatic theory cannot be proven to be of complexity (entropy) more than $O(1)$ greater than the complexity (entropy) of the axioms of the theory. Conversely, "there are formal theories whose axioms have entropy $n+O(1)$ in which it is possible to establish all true propositions of the form "$H(specific\ string) \geq n$"." [Chaitin '77]* $\diamond$

Proof We reproduce Chaitin's proof. "Consider the enumeration of the theorems of the formal axiomatic theory in order of the size of their proof. For each natural number k, let s^* be the string in the theorem of the form "$H(s) \geq n$" with n greater than $H(axioms)+k$ which appears first in this enumeration. On the one hand, if all theorems are true, then $H(s^*) > H(axioms)+k$. On the other hand, the above prescription for calculating s^* shows that $H(s^*) \leq H(axioms)+H(k)+O(1)$. It follows that $k < H(k)+O(1)$. However, this inequality is false for all $k \geq k^*$, where k^* depends only on the rule of inference. The apparent contradiction is avoided only if s^* does not exist for $k = k^*$, i.e., only if it is impossible to prove in the formal theory that a specific string has H greater than $H(axioms)+k^*$. *Proof of Converse*. The set T of all true propositions of the form "$H(s) < k$" is r.e. Chose a fixed enumeration of T without repetitions, and for each natural number n let s^* be the string in the last proposition of the form "$H(s) < n$" in the enumeration. It is not difficult to see that $H(s^*,n) = n+O(1)$. Let p be a minimal program for the pair s^*, n. Then p is the desired axiom, for $H(p) = n+O(1)$ and to obtain all true proposition of the form "$H(s) \geq n$" from p one enumerates T until all s with $H(s) < n$ have been discovered. All other s have $H(s) \geq n$." $\diamond$

Consequences

Many aspects of the reality can be encoded in finite binary strings with more or less accuracy. Because, a tremendous majority of this strings are random, our capacity to do *strict rigorously* forms for all the processes in reality is practically null. Indeed, the formalization is a process of condensation in short expressions, i.e., in programs associated with machines. Some programs can be considered a *formula* for large strings and some not. Only for a few number of strings (realities) a short program can be written. Therefore, we have three solutions:

1. to accept this limit

2. to reduce the accuracy of the representations, making partitions in the set of strings, thus generating a seemingly enlarged space for the process of formalization (many insignificant (?) facts can be "filtered" out, so "cleaning" up the reality by small details (but attention to the small details!))

3. to accept that the reality has deep laws that govern it and these laws can be discovered by an appropriate approach which remains to be discovered.

The last solution says that we live in a subtle and yet unknown Cartesian world, the first solution does not offer us any chances to understand the world, but the middle is the most realistic and optimistic in the same time, because it invites us to "filter" the reality in order to understand it. The effective knowledge implies many subjective options. **For knowing, we must filter out.** The degree of knowledge is correlated with our subjective implication. The objective knowledge is a nonsense.

Algorithmic information theory is a new way for evaluating and mastering the complexity of the big systems.

3.1.3 General Information Theory

Beyond the quantitative (Shannon, Chaitin) and qualitative (Chaitin) aspects of information in formal systems (like digital systems for exemplu) turns up the necessity of a *general information theory* [Drăgănescu '84]. The concept of information must be applied to the non-structured or to the informal defined objects, too. These objects can have an useful function in the future computation paradigms and we must pay attention for them.

To be prepared to understand the premises of this theory we start with two main distinctions:

- between *syntax* and *semantics* in the approach of the world of signs
- between the *signification* and the *sense* of the signs.

Syntactic-Semantic

Let be a set of signs (usually but incorrectly named symbols in most papers), then two types of relations can be defined within the semiotic science (the science of signs):

- an *internal* relation between the elements of the set, named *syntactic relation*
- an *external* relation with another set of objects, named *semantic relation*.

Definition 3.9 *The* syntactic *relation in the set* A *is a subset of the cartesian product* $(A \times A \times \ldots \times A)$. $\diamond$

By the rule, a syntactical relation makes *order* in manipulating symbols to generate useful configurations. These relations emphasize the ordered spaces which have a small complexity. We remind that, according to the algorithmic information theory, the complexity of a set has the order of the complexity of the rule that generates it.

Definition 3.10 *The semantic relation between the set* S *of* signifiers *and the set* O *of* signifieds *is* $R \in (S \times O)$. *The set* S *is a formal defined mathematical set, but the set* O *can be a mathematical set and in the same time can be a collection of physical objects, mental states, ... Therefore, the semantically relation can be sometimes beyond of a mathematical relation.* $\diamond$

Sense and Signification

The semantic relation leads us towards two new concepts: *signification* and *sense*. Both are aspects of the *meaning* associated to a set in which there is a syntactical relation.

Definition 3.11 *The signification can be emphasized using a* formal semantical relation *in which each signifiers has one or more signifieds.* $\diamond$

Definition 3.12 *The sense of an object is a meaning which cannot be emphasized using a formal semantic relation.* $\diamond$

By the above definition, the meaning of the *sense* remains undefined because its meaning may be *suggested* only by an informal approach. We can try an informal definition:

The sense may be the signification in the context of the wholeness.

The sense blows up only in the wholeness. We cannot talk about "the set of senses". Our interest regarding the sense is due to the fact that the senses *act* in the whole reality. A symbol or an object full of senses may have an essential role in the interaction between the technical reality and the wholeness. When an object has sense it overtakes the system, becomes more than a system. By the rule, an object has a signification and sometimes a sense. (*Seldom there is the situation when the object has only sense, but not in the world of the objects.*)

The signification is a formal relation and acts in the structural reality. The sense is an informal connection between an object and the wholeness and acts in a *phenomenological* reality. The structural-phenomenological reality supposes the manifestation of the signification and of the sense. Our limited approach only makes the difference between the structural and the phenomenological. The pure structural reality does not exist, it is created only by our helplessness in understanding the world. On the other hand, the "phenomenological reality" is a pleasantly and motionless dream. Only the play between sense and signification can be a key for dealing with the complexity of the structural-phenomenological reality.

Generalized Information

Starting from the distinctions above presented the **generalized information** will be defined using [Drăgănescu '84].

Definition 3.13 *The* generalized information *is:*

$$N =< S, \mathscr{M} >$$

where: S is the set of objects characterized by a syntactical relation, $\mathscr{M}$ *is the* meaning *of S.* $\diamond$

In this general definition, the meaning associated to S is not a consequence of a relation in all the situations. The meaning must be detailed, emphasizing more distinct levels.

Definition 3.14 *The* informational structure *(or syntactic information) is:*

$$N_0 =< S >$$

where the set of objects S is characterized only by a syntactical (internal) relation.$\diamond$

The informational structure N_0 is the simplest information, we can say that it is a *pre-information* having no meaning. The informational structure can be only a good support for the information.

Example 3.5 *The content of a RAM between the addresses* $0103_H - 53FB_H$ *does not have an informational character without knowing the architecture of the host computer.* $\diamond$

The first actual information is the semantic information.

Definition 3.15 *The* semantic information *is:*

$$N_1 =< S, \mathbf{S} >$$

where: S is a syntactical set, and $\mathbf{S}$ *is the set of significations of S given by a relation in* $(S \times \mathbf{S})$. $\diamond$

Now the meaning exists but it is reduced to the signification. There are two types of significations:

- R, the *referential* signification
- C, the *contextual* signification

thus, we can write:

$$\mathbf{S} =< R, C > .$$

Definition 3.16 *Let us call the* reference information*:* $N_{11} =< S, R >$. $\diamond$

Definition 3.17 *Let us call the* context information*:* $N_{12} =< S, C >$. $\diamond$

If in N_{11} to one significant there are more significats, then adding the N_{12} the number of the significats *can* be reduced, to one in most of the situations. Therefore, the semantic information can be detailed as follows:

$$N_1 =< S, R, C > .$$

Definition 3.18 *Let us call the* phenomenological *information:* $N_2 =< S, \sigma >$*, where:* σ *are senses.* $\diamond$

Attention! The entity σ is not a set.

Definition 3.19 *Let us call the* pure phenomenological *information:* $N_3 =< \sigma >$. $\diamond$

Now, the expression of the information is detailed emphasizing all the types of information:

$$N =< S, R, C, \sigma >$$

from the objects without a specified meaning, $< S >$, to the information without a significant set, $< \sigma >$.

Generally speaking, because all the objects are connected to the whole reality the information has only one form: N. In concrete situations one or another of these forms is promoted because of practical motivations. In digital systems we can not overtake the level of N_1 and in the majority of the situations the level N_{11}. General information theory associates the information with the meaning in order to emphasize the distinct role of this strange ingredient.

3.2 Looping toward Functional Information

Information arises in a natural process in which circuits grow in *size* and in *complexity*. There is a level from which the increasing complexity of the circuits tend to stop and only the circuit size continues to grow. This is a very important moment because the complexity of computation continues to grow based on the increasing of another entity: the *information*. The computational power is distributed from this moment between two main structures:

- a **physical structure** that can grow in size remaining at a moderate or a small complexity
- a **symbolic structure** that has a random structure with the size in the same order with the complexity.

The birth of information is determined by the gap between the size of circuits and their complexity. This gap allows the segregation process, which emphasizes *functional* defined circuits as *simple* circuits. Also, this gap increases the weight of control. Indeed, a small number of well defined functional circuits must do complex computations coordinated by a complex control.

Information assumes the control in the computing systems. It is a way to put together a small number of functional segregated circuits in order to perform complex computations.We usie *simple* machines controlled by *complex* programs. Information comes out in a process in which the *random* part of computation is **segregated** from the *simple* (recursive defined) part of computation. Now, let us explain this process.

The first step towards the definition of information is to emphasize the *informational structure*. In this approach, we will make two distinctions in the class of the automata. The first between automata having *random loops* and *functional loops* and the second between automata with *non-structured* states and with *structured* states. After that, the *informational structure* is defined at the level of the second order digital systems and *information* is defined at the level of the third order digital systems. We end at the level of the 4-OS where information gains a complete control of the function in digital systems.

3.2.1 Random Loop vs. Functional Loop

Let us start with a simple exemplu. Usually we call *half-automaton* a circuit built by a state register R and a combinational circuit *CLC* loop coupled. Most of the circuits designed as half-automaton contain a *CLC* having a "random" structure, i.e., a structure without a simple recursive definition. The minimal definition of a random *CLC* has the size in the same

order with the size of the circuit. On the other hand, there are half-automata with the loop closed over simple, recursive defined *CLC*s having big or small sizes. These *CLC* have well defined functions and in consequence have always a "name", such as: adder, comparator, priority encoder, ... This distinction can be extended over all circuits having internal loops and will have a very important consequences on the structuring process in digital systems. A random structure can not be expanded instead of a recursive defined functional structure that contains in its definition the expansion rule. In the structural developing process the growth of the random circuits stops very soon rather than the same process for functional circuits that is limited only by technological reasons.

Definition 3.20 *The* random *loop of an automaton is a loop on which the actual value assigned for the state has only structural implications on the combinational circuit without any functional consequences on the automaton.* ◇

Any finite automaton has a random loop and the state code can be assigned in many kinds without functional effects. Only the optimization process is affected by the actual binary value assigned to each state.

Definition 3.21 *The* functional *loop of an automaton is a loop on which the actual value of the state is strict related to the function of the automaton.* ◇

A counter has a functional loop and its structure is easy expandable for any number of bits. The same is Bits Eater Automaton (see Figure **??** in Chapter 4). The functional loop will allow us to make an important step towards the definition of information.

If an automaton has a loop closed through uniform circuits (multiplexors, priority encoder, demultiplexor and a linear network of XORs, ...) that all have recursive definitions, then at the input of the state register, the binary configurations have a precise meaning, imposed by the functional circuits. We don't have the possibility to choose the state assignment because of the combinational circuit that has a predefined function.

A final exemplu will illustrate the distinction between the structural loop and the functional loop in a machine that contains both situations.

Example 3.6 *The Elementary Processor (see Figure* **??***) contains two automata. The* **control automaton** *has a* structural loop*: the commands, whatever they are, can be stored in* ROM *in many different orders. The binary configuration stored in ROM is random and the ROM as combinational circuit is then a random circuit. The second automaton is an* **functional automaton** *(R_n & ADD_n & $nMUX_4$) with a* functional loop*: the associated CLC has well defined digital functions (ADD_n & $nMUX_4$)and through the loop we have only binary configurations with* **well-defined meaning***: numbers.*

There is also a third loop, closed over the two previous mentioned automata. The control automaton is loop connected with a system having a well-defined function. The field `<func>` *is used to generate towards ADD_n & $nMUX_4$ binary configurations with a precise meaning. Therefore, this loop is also a functional one.* ◇

On the random loop we are free to use different codes for the same states in order to optimize the associated CLC or to satisfy some external imposed conditions (related to the synchronous or asynchronous connection to the input or to the output of the automaton). The actual code results as a deal with the structure of the circuits that close the loop.

On the functional loop the structure of the circuit and the *meaning* of binary configurations are reciprocally conditioned. The designer has no liberty to choose codes and to optimize circuits. Circuits on the loop are imposed and signals through the loop have well defined meanings.

3.2.2 Non-structured States vs. Structured States

The usual automata have the states coded with a *compact* binary configuration. As we knoow, the size of a combinational circuit depends, in the general case, exponentially by the number of inputs. If the number of bits used for coding the state becomes too large the circuit that implements the loop can grow too much. In order to reduce the size of this combinational circuit the state can be divided in *many fields*, in each clock cycle being modified the value of one field only. So the state gets an *internal structure*.

Definition 3.22 *The* **structured state space automaton** *(S^3A) [Ştefan '91] is:*

$$S^3A = (X \times A, Y, Q_0 \times Q_1 \times \ldots \times Q_q, f, g)$$

where:

- *$X \times A$ is the input set, $X = \{0,1\}^m$ and $A = \{0,1\}^p = \{A_0, A_1, \ldots, A_q\}$ is the selection set, with $q+1 = 2^p$*
- *Y is the output set*
- *$Q_0 \times Q_1 \times \ldots \times Q_q$ is the structured state set*
- *$f : (X \times A \times Q_0 \times Q_1 \times \ldots \times Q_q) \to Q_i$ has the following form:*

$$f(x, P(a, q, q_0, q_1, \ldots, q_q)) = f'(x, q_a)$$

with $x \in X$, $a \in A$, $q_i \in Q_i$, where $f' : (X \times Q_a) \to Q_a$ is the state transition function and P is the projection function (see Chapter 8)

- *$g : (X \times A \times Q_0 \times Q_1 \times \ldots \times Q_q) \to Y$ has the following form:*

$$g(x, P(a, q, q_0, q_1, \ldots, q_q)) = g'(x, q_a)$$

with $x \in X$, $a \in A$, $q_i \in Q_i$, where $g' : (X \times Q_a) \to Y$ is the output transition function.$\diamond$

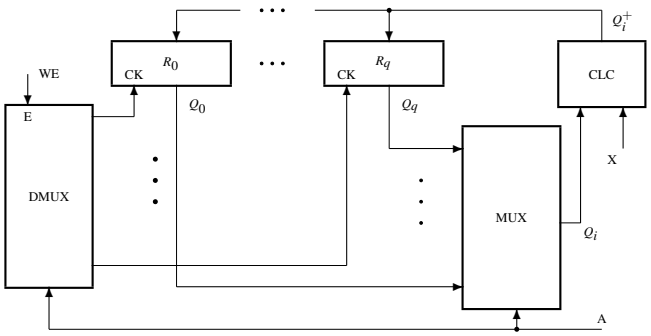

Figure 3.2: The structured state space automaton as a multiple register structure.

The main effect of this approach is the huge reduction of the size of the circuit that closes the loop. Let be $Q_i = \{0,1\}^r$. Then $Q = \{0,1\}^{r\times(q+1)}$. The size of CLC without structured state space should be $S_{CLC} \in O(2^{m+r\times(q+1)})$, but the equivalent variant with structures state space has $S_{CLC'} \in O(2^{m+r})$. Theoretically, the size of the circuit is reduced 2^{q+1} times. The price for this fantastic (only theoretical) reduction is the execution time that is multiplied with $q+1$. **The time increases linearly and the size decreases exponentially**. There is no engineer that dares to ignore this fact. All the time when this solution is possible, it will be applied.

The structure of a S^3A is obtained in a few steps starting from the structure of a standard automaton. In the first step (see Figure 3.2) the state register is divided in $(q+1)$ smaller registers (R_i, $i = 0,1,\ldots,q$) each having its own clock input on which it receives the clock distributed by the demultiplexer DMUX according to the value of the address A. The multiplexer MUX selects, according to A, the content of one of the $q+1$ small registers to be applied to CLC. The output of CLC is stored only in the register that receives the clock.

But, in the structure from Figure 3.2 there are too many circuits. Indeed, each register R_i is build by a *master* latch serial connected with the corresponding *slave* latch. The second stores an element Q_i of the Cartesian product Q, but the first acts only in the cycles in which Q_i is modified. Therefore, in each clock cycle only one *master* latch is active. Starting from this evidence, the second step will be to replace the registers R_i with latches L_i and to add a single *master latch* ML (see Figure 3.3). The latch ML is *shared* by all the slave latches L_i for a proper closing of a non-transparent loop. In each clock cycle the selected L_i and ML form a well structured register that allows to close the loop. ML is triggered by the inverted clock CK' and the selected latch by the clock CK.

The structure formed by $DMUX$, MUX and $L_0, \ldots, L_q$ is obviously a random access memory (RAM) that stores $q+1$ words of r bits. Therefore, the last step in structuring a S^3A is to emphasize the RAM by the structure from Figure 3.4. Each clock cycle allows to modify the content of a word stored at the address A according to the input X and the function performed by CLC.

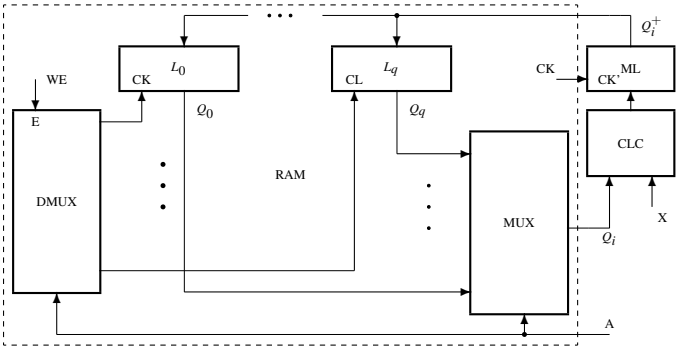

Figure 3.3: The structured state space automaton as a single master-latch.

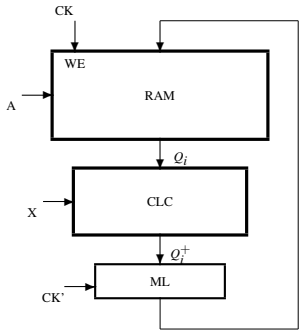

Figure 3.4: The structured state space automaton with RAM and master latch.

Example 3.7 *A very good exemplu of S^3A is the core of each classical processor: the registers (R) and the arithmetic and logic unit (ALU) that form together RALU (see Figure 3.5). The memory RAM has two read ports, selected by* `Left` *and* `Right` *and a write port selected by* `Dest`*. It is very easy to imagine such a memory. In the representation from Figure 3.3 the selections code for DMUX, separated from the selection code of MUX, becomes* `Dest` *and a new MUX is added for the second output port. One output port has the selection code* `Left` *and the other has the selection code* `Right`*. MUX selects (by* `Sel`*) between the binary configuration received from an external device (DIN) and the binary configuration offered to the left output LO of the memory RAM.*

In each clock cycle two words from the memory, selected by `Left` *and* `Right` *(if* `Sel = 1`*), or a word from memory, selected by* `Right`*, and a receiver word (if* `Sel = 0`*), are offered as arguments for the function* `Func` *performed by ALU and the result is stored to the address indicated by* `Dest`*.*

The line of command generated by a control automaton for this device is:

```
<RALU Command> ::= <Left> <Right> <Dest> <Sel> <Func> <Write>
<Left> ::= L0 | L1 | ...  | Lq | - ,
<Right> ::= R0 | R1 | ...  | Rq | - ,
<Dest> ::= D0 | D1 | ...  |Dq | - ,
<Sel> ::= DIN | - ,
<Func> ::= AND | OR | XOR | ADD | SUB | INC | LEFT | SHR,
<Write> ::= W | - .
```

RALU returns to the control automaton some bits as indicators*:*
`Indicators` = {`CARRY, OVFL, SGN, ODD, ZERO`}.
The output AOUT will be used in applications needed to address an external memory. ◇

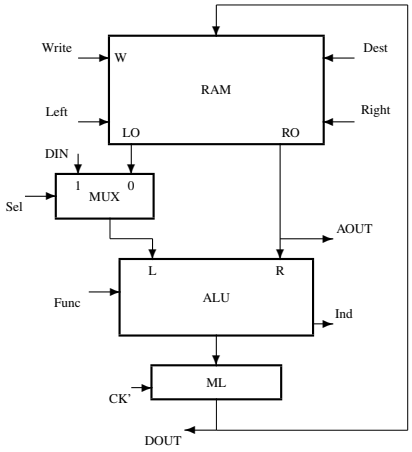

Figure 3.5: An exemplu of structured state space automaton: Registers with ALU (RALU)

The structured state of RALU is modified by a sequence of commands. This sequence is generated by the rule, using a control automaton that works according to its switching functions for the state and for the output, taking into account sometimes the evolution of the indicators.

3.2.3 Informational Structure in Two Loops Circuits (2-OS)

We have seen at the level of the 2-OS appearing a symbolic structure: the *Cartesian product* defining the state space of the automaton. This symbolic structure is very important for two reasons:

1. the RALU, that supports it, is one of the main structure involved in defining and building the *central unit* of a computing machine
2. it is the support for **meanings** that gains, step by step, an important role in defining the function of a digital system; we shall call this new structure *informational structure*.

The Cartesian product $(Q_0 \times Q_1 \ldots \times Q_q)$ stored in the RAM is the state of the automaton. What are the differences between this structured state and the state of a standard finite automaton? The state of a standard automaton has two characteristics:

- it is a whole entity, without an internal structure
- it may be encoded in many equivalent forms and the external behavior of the automaton remains the same; each particular encoding has its own combinational circuit thus the automaton runs in the state space in the same manner; any code changing is compensated by a modification in the structure of the circuit.

In S^3A, RALU for exemplu, the situation is more different:

- the state *has a structure*: the structure of a Cartesian product
- using a *functional loop* (well defined combinational circuit (ALU) closes the loop) we loose the possibility to make any state assignment for Q_i and the concrete form of the state codes have a well defined *meaning*: they are numbers.

Definition 3.23 *The* **informational structure** *is a* structured state *that has a* meaning *correlated with the* functional loop *of an automaton.*$\diamond$

The state of a standard automaton doesn't have any meaning because the loop is closed through a random circuit having a structure "negotiated" with the state assignment. This meaningless of the state code is used for minimizing the combinational circuit of the automaton or to satisfy certain external conditions (asybchronous inputs or free of hazard

outputs). When the state degenerates in informational structure this resource for optimization is lost. What is the gain? I believe that the gain is a new structure - the *informational structure* - that will be used to improve the functional resources of a digital machine and for simplifying its structure.

The *functional loop* and the *structured state* lead us in the neighborhood of information, emphasizing the *informational structure*. The process was stimulated by the segregation of the simple, recursive defined combinational resources of the infinite automata. And now: the main step!

3.2.4 Functional Information in Three Loops Circuits (3-OS)

The functional approach in the structured space *automata* generates the informational structure. Therefore, the second order digital systems offer the context for the birth of the informational structure. The third order digital systems is the context in which the informational structure degenerates in *information*. Therefore, the information is strongly related to the processing function. At the level of processors the informational structure can *act directly* and becomes in this way *information*.

Let's put together the just defined RALU with an improved *control automaton* that was defined as CROM, thus defining a *microprogrammed processor*.

Definition 3.24 *A* **microprogrammed processor** *consists in a RALU, as an* functional automaton*, loop coupled with a CROM, as a* control automaton*. The function of a CROM is given by its* internal structure *and the associated* microprogramming language*. The structure of the simplest CROM is shown in Figure 3.6 (is a variant having the complexity between the structure from Figure* **??** *and the structure from Figure* **??***), where:*

R *is the state register containing the address of the current microinstruction*

ROM *is the combinational random circuit generating ("containing") the current microinstruction having the following fields:*

`<RALU Command>` *containing subfields for RALU (see Exemple 10.7)*

`<Out>` *the field for commanding the external devices (in this exemplu assimilated with the program and data memory)*

`<Test>` *is the field that selects the appropriate indicator for the current switch of CROM*

`<Next>` *is the jump address if the value of T is true (the selected indicator is* 1*)*

`<Mod>` *is the bit that selects together with T the transition mode of the automaton*

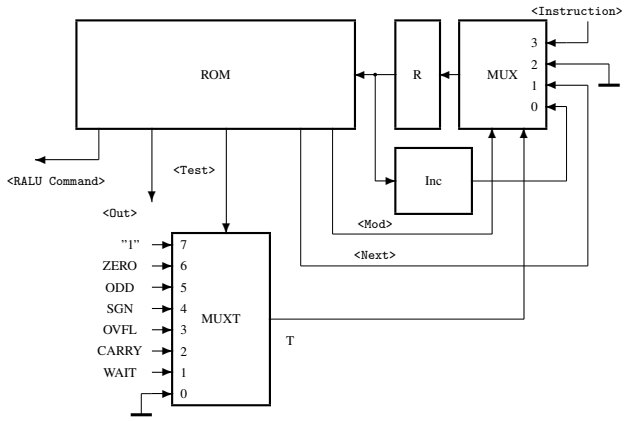

Figure 3.6: CROM

Inc. *is an incrementer realized as a combinational circuit; it generates the next address in current unconditioned transition of the automaton*

MUX *is the multiplexer selecting the next address from:*

- *the incremented current address*
- *the address generated by the microprogram*
- *the address* $00\ldots0$*, for restarting the system*
- *the instruction received from the external memory (the instruction code is constituted by the address from which begin the microprogram associated to the instruction)*

MUXT *is the multiplexer that selects the current indicator (it can be* 0 *or* 1 *for non-conditioned or usual transitions)*

The associated microprogramming language is:

```
<Microinstruction> ::= <RALU Command> <Out> <Mod> <Test> <Next>
<RALU Command> ::= <Left> <Right> <Dest> <Sel> <Func> <Write>
<Left> ::= L0 | L1 | ...  | Lq | - ,
<Right> ::= R0 | R1 | ...  | Rq,
<Dest> ::= D0 | D1 | ...  |Dq | - ,
<Sel> ::= DIN | - ,
<Func> ::= AND | OR | XOR | ADD | SUB | INC | LEFT | SHR,
<Write> ::= W | - ,
<Out> ::= READ | WRITE | - ,
<Mod> ::= INIT | - ,
<Test> ::= - | WAIT | CARRY | OVFL | SGN | ODD | ZERO | TRUE,
<Next> ::= <a label unused before in this definition of maximum six symbols starting with a
letter>.
```

The `WAIT` *signal is received from the external memory.* ◇

In the previous defined machine let be $q = 15$ and $r = 16$, i.e., the machine has 16 register of 16 bits. The register Q_{15} takes the function of the *program counter* (PC) addressing the program space in the memory. The first microprogram must be done for the previous defined machine is the microprogram that initializes the machine resetting the program counter (PC) and, after that, loops forever reading (fetching) an instruction, incrementing PC and giving the access to the microprogram execution. Each microprogram, that *interprets* an instruction, ends with a jump back to the point where a new instruction is fetched, and so on.

Example 3.8 *The main microprogram that drives a microprogrammed machine interpreting a machine language is described by the next procedure.*

Procedure *PROCESSOR*
PC ← *the value zero*
loop
do *READ from PC*
until *not WAIT*
repeat
READ from PC, INIT and PC ← *PC + 1*
repeat
end *PROCESSOR*

The previous procedure has the next implementation as a microprogram.

```
          L15 R15 D15 XOR W // Clear PC //
  LOOP    R15 READ WAIT LOOP // Fetch the current instruction //
          R15 READ TRUE INIT L15 D15 XOR W // ‘‘Jump" to the
          associated microprogram and increment PC //
```

◇

The previous microprogram, or a similar one, is stored starting from the address $00\ldots0$ in any microprogrammed machine. The restart function of CROM facilitates the access to this microroutine.

Definition 3.25 *The* **Processor** *is a third order machine (3-OS) built with two* loop-coupled *2-OS systems, i.e., two distinct automata:*

1. *a* **functional automaton** *receiving commands from a control automaton and returning indicators that characterize the current performed operation (usually is a RALU)*
2. *a* **control automaton** *(CROM in a microprogrammed machine) receiving:*
   - Instructions *that initialize the automaton in order to perform it by* interpretation *(each instruction has an associated microprogram* executed *by the controlled subsystems)*
   - Indicators (flags) *from the functional automaton and from the external devices for decisions within the current microprogram.* $\diamond$

For this subsection one exemplu of instruction is sufficient. The instruction is an exotic one, atypical for a standard processor but very good as an exemplu. The instruction computes in a register the integer part of logarithm from a number stored in another register of the processor. The microprogram implements the *priority encoder* function (see Chapter 2).

Example 3.9 *Let be Q_0 the register that stores the variable and Q_1 the register that will contain the result, if it exists, else (the variable has the value zero) the result will be* $11\ldots1$. *The microprogram is:*

```
        L1 R1 D1 XOR W
        L0 LEFT ZERO ERROR
TEST    L0 D0 SHR ZERO LOOP
        L1 D1 INC W TRUE TEST
ERROR   L0 D0 INC W
        L1 R0 D1 SUB W TRUE LOOP
```

The label `LOOP` *refers in the previous microprogram.* $\diamond$

Each line of microprogram has a binary coded form according to the structure of circuits commanded.
The machine just defined is the typical digital machine for 3-OS: the **processor**. Any processor is characterized by:

- the *behavior* defined by the set of control sequences (in our exemplu microprograms implemented in ROM)
- the *structure* that usually contains many functional segregated simple circuits
- the *flow of the internal loop signals*.

Because the *behavior* (the set of control sequences or of microprograms) and the *structure* (composed by uniform recursive defined circuits) are imposed, we don't have the liberty to choose the actual coding of the *signals* that flows on the loops. In this restricted context there are three types of binary coded sets which "flow" inside the processor:

- *informational structured sets* having elements with a well defined meaning, according to the associated functional loop (for exemplu, the meaning of each Q_i from RALU is that of a number, because the circuit on the loop (ALU) has mainly arithmetic functions)
- the set of *indicators* or *flags* that are signals generated indirectly by the informational structure through an arithmetical or a logical function
- *information*, an informational structured set that generates *functional effects* on the whole system by its flow on a functional loop formed by RALU and CROM.

What is the difference between information and informational structure? Both are informational structures with a well defined *meaning* regarding to the physical structure, but information *acts* having a functional role in the system in which it flows.

Definition 3.26 *The* functional information *is an informational structure which generates strings of symbols specifying the actual function of a digital system.* ◇

The content of ROM can be seen as a Cartesian product of many sets, each being responsible for controlling a physical structure inside or outside the processor. In our example there are 10 fields: six for RALU, one for outside of the machine (for memory) and 3 for the controller. A sequence of elements from this Cartesian product, i.e., a microprogram, *performs* a specific function. We can interpret the information as a *symbolical structure* having a *meaning* through which it *acts* performing a *function*.

The informational structure can be *data* or *microprograms*, but only the *microprograms* belong to the information. At the level of the third order systems (processors) the information is made up only by *microprograms*.

Until now, we emphasized in a processing structure two main informational structures:

- the processed strings that are data (informational structure)
- the strings that lead the processing: microprograms (information).

The informational structure is **processed** by the processor as a whole consisting in two entities:

- a simple and recursive *physical structures*
- the information as a *symbolic complex structure*.

The information is **executed** by the simple functional segregated structures inside the processor.

The information is the random part of the processor. The initial randomness of digital circuits, performing any functions, was converted in the randomness of symbolic structures which meanings are executed by a simple, recursive defined digital circuits. Thus, the processor has two structures:

1. a physical one, consisting in a big size, low complex system
2. a symbolic one, having the complexity related with the performed computation.

According to the sense established for the term information we can say that digital systems do not process the information, they *process through information*.

And now, what is the difference between *flags* and *information*? A flag is *interpreted* through information instead of information that is *executed* by the physical structure (by the hardware). The value of the flag does not have any meaning all the time for the processing. It has meaning only when the information "needs" to know the value of the indicator (the indicator is selected by the field `<Test>`). The flag acts indirectly and suffers a symbolic, informational *interpretation* instead of the hardware *execution* to which the microprogram is submited. The flags are an intermediate stage between the informational structure and information. The flags do not belong to any informational structure.

The loop "closed through" the flags is a weak informational one. The flags classify the huge content of the informational structure in few classes. Only a small part of the meaning contained in data (the informational structure) *acts* having a functional role. Through flags the informational structure manifests with shyness as information. The flags emphasize the small informational content of the informational structure. Thus, between the information and the informational structure there is not a net distinction. The informational structure influence, through the flags only some execution details not the function to be executed.

3.2.5 Controlling by Information in Four Loops Circuits (4-OS)

In the previous subsection, the information interacts directly with the physical structure. All the information is executed or interpreted by the circuits. The next step disconnects partially the information from circuits. In a system, having four loops the information can be interpreted by another information acting to the lower level in the system. The typical 4-OS is the *computer* structure (see Chapter 6). This structure is more than we need for computing. Indeed, as we said in Chapter 8 the partial recursive functions can be computed in 3-OS. Why are we interested in using 4-OS for performing computations? The answer is: *for segregating more the simple circuits from random (complex) informational structure*. In a system having four loops the simple and the complex are maximal segregated, the first in circuits and the second in information.

In order to exemplify how information acts in 4-OS we will use a very simple language: *Extended LOOP* (ELOOP). This language is equivalent with the computational model of partial recursive functions. For this language, an architecture will be defined. The architecture has associated a processor (3-OS) and works on a computer (4-OS).

Definition 3.27 *The LOOP language (LL) is defined as follows [Calude '82]:*
<character>::=A|B|C|...|Z
<number>::=0|1|2|...|9
<name>::=<character>|<name><number>|<name><character>
<instruction>::=<name>=0|<name>=<name>+1|<name>=<name>
<loop>::=LOOP<name>
<end>::=END
<program>::=<loop><program><end>|<program><program>|<instruction>
◇

The LOOP language is devoted to compute primitive recursive functions only. (See the proof in [Calude '82].) A new feature must be added to the LOOP language in order to use it for computing partial recursive functions. The language must **test** sometimes the value resulting in the computation process (see the minimalization rule in 8.1.4).

Definition 3.28 *The Extended LOOP Language (ELOOP) is the LL supplemented with the next instruction:*

$$IF X \neq 0 \ GO TO < label >$$

where $< label >$ is the "name" of an instruction from the current program. ◇

In order to implement a machine able to execute a program written in the ELOOP language we propose two architectures: AL1 and AL2. The two architectures will be used to exemplify different degrees of interpretations. There are two ways in which the information *acts* in digital systems:

- by execution - digital circuits interpret one, more or all fields of an instruction
- by interpretation - another informational structure (by the rule a microprogram) interprets one, more or all fields of the instruction.

In the fourth order systems the ratio between interpretation and execution is modified depending on the architectural approach. If there are fields having associated circuits that directly execute the functions indicated by the code, then these fields are directly *executed*, else these are *interpreted*, usually by microprograms.

Definition 3.29 *The* assembly language one *(AL1), as a minimal architecture associated for the processor that performs the ELOOP language, contains the following instructions:*

LOAD <Register> <Register>*: load the first register with the content of the external memory addressed with the second register*

STORE <Register> <Register>*: store the content of the first register on the cell addressed with the second register*

COPY <Register> <Register>*: copy the content of the first register in the second register*

CLR <Register>*: reset the content of the register to zero*

INC <Register>*: increment the content of the register*

DEC <Register>*: decrement the content of the register*

JMP <Register> <address>*: if the content of the register is zero, then jump to the instruction stored at the indicated address, else execute the next instruction*

NOP *: no operation*

where:

```
<Register> ::= R0 | R1 | ...  | R15
```

The instructions are coded in one 16 bits word. The registers have 16 bits. ◇

There are some difficulties in the previous defined architecture to construct in registers the addresses for *load, store* and *jump*. In order to avoid this inconvenient in the second architecture addresses are generated as values in a special field of the instruction.

Definition 3.30 *The* assembly language two *(AL2), as a minimal architecture associated for the processor that performs ELOOP language, contains the following instructions:*

LOAD `<Register> <Address>`*: load the internal register of the processor with the addressed content of the external memory*

STORE `<Register> <Address>`*: store the content of an internal register on the addressed cell in the external memory*

COPY `<Register> <Register>`*: copy the content of the first register in the second register*

CLR `<Register>`*: reset the content of the register to zero*

INC `<Register>`*: increment the content of the register*

DEC `<Register>`*: decrement the content of the register*

JMP `<Register> <Address>`*: if the content of the register is zero, then jump to the instruction stored at the indicated address, else execute the next instruction*

NOP *: no operation*

where:

```
<Register> ::= R0 | R1 | ...  | R15
<Address> ::= 0H | 1H | ...  | FFFFH.
```

The instructions with the field `<Address>` *are coded in two 16-bits words and the rest in one 16 bits word. The registers have 16-bits.* ◇

The microprogrammed machine previously defined (see Definition 10.24) can be used without any modification to implement the processor associated to these two architectures.

Each instruction in AL1 has the associated microprogram. The reader is invited to make a first exercise implementing this processor using the microprogrammed machine defined in the previous subsection. The exercise consists in writing many microprograms. Each of the six instructions using a register needs 16 microprograms, one for each register. The LOAD, STORE, COPY and JUMP instructions use two registers and we must write 256 microprograms for them. For NOP there is only one microprogram. Therefore, the processor is defined by $3 \times 16 + 4 \times 2073 + 1 =$ microprograms. A big amount of microprogram memory is wasted.

The same machine allows us to implement a processor with the AL2 architecture. In this case, the address is stored in the second word of the instructions: LOAD, STORE and JUMP. The number of needed microprograms decreases to $6 \times 16 + 256 + 1 = 353$.

In order to avoid this big number of microprograms a third exercise can be done. We will modify the internal structure of the processor thus the field `<Register>` is interpreted by the circuits, not by the information as microprogram. (The field `<Register>` accesses direct through a miltiplexer the RALU inputs `<Left>` and `<Dest>`.) Results a machine defined by eight microprograms only, one for each instruction.

Thus, there are many degrees of interpretation at the level of the fourth order systems. In the first implementation the entire information contained by the instruction is interpreted by the microprogram.

The second implementation offers a machine in which the field `<Address>` is executed by the decoder of the external RAM, after its storage in one register of the processor.

The third implementation allows a maximal execution. This variant interprets only the field that contains the name of the instruction. The fields specifying the registers are executed by the RAM from RALU and the address field is stored in RALU and after that is executed by the external memory.

In the first solution, the physical structure has no role in the actual function of the machine. The physical structure has only a potential role, it interprets the basic information: the microprograms.

The third solution generates a machine in which the information, contained by the programs stored in the external RAM, acts in two manners: is *interpreted* by the microprograms (the field containing the name of the instruction) and is *executed* by circuits (the fields containing the register names are decoded by the internal RAM from the RALU and the field containing the value of the address is decoded by the external RAM).

There are processors, which have an architecture in which the information is entirely executed. A pure RISC processor can be designed having circuits that execute all instruction fields. Between complete interpretation and complete execution, the current technologies offer all the possibilities.

Starting from the level of the fourth order systems the functional aspects of a digital system is imposed mainly by the information. The role of the circuits decreases. Circuits become simple even if they gain in size. The complexity of the computation switches from circuits to information.

3.3 Comparing Information Definitions

Ending this chapter about information, we make some comments about the interrelation between the different definitions of this full of meanings term that we discussed here. We want to emphasize that there are many convergences in interpreting different definitions for information.

1. Shannon's theory evaluates the information associated with the set of events instead of Chaitin's approach which emphasizes the information contained in each event. Even with this initial difference, the final results for big sized realities are in the same order for the majority of events. Indeed, according to Theorem 10.7 the most of n-bit strings have information around the value of n bits.

2. The functional information and the algorithmic information offer two very distinct images. The first is an exclusive qualitative approach, instead of the second which is a preponderant quantitative one. Even that, the final point in this two theories is the same: the *program* or a related symbolic structure. The functional way starts from *circuits* instead of the algorithmic approach that starts from the *string of symbols*. Both have in the second plane the idea of *computation* and both are motivated by the relation between the *size* and the *complexity* of the circuits (for functional information) or of the strings (for algorithmic information).

3. The functional information is a particular form of the generalized information defined by Drăgănescu, because the *meaning* (having the form of the *referential signification*) associated to strings of symbols *acts* generating functional effects.

4. The jump from the binary string to the program of a machine that generates the string can be assimilated with the relation between the string and its meaning. This meaning, i.e., the program, is interpreted by the machine generating the string. The *interpretation* is the main function that allows the birth of functional information. Therefore, the *interpretation function*, the *meaning* and the *string* are main concepts that connect the functional information, generalized information and algorithmic information.

5. *The information* **acts** *in a well-defined functional context by its* **meaning** generating a string having the complexity related to the size of its expression. The basic mechanism introduced by information is the interpretation. A string has a meaning and the meaning must be interpreted. Algorithmic information emphasizes the meaning and the functional information emphasizes the functional segregated context in which the meaning is interpreted.

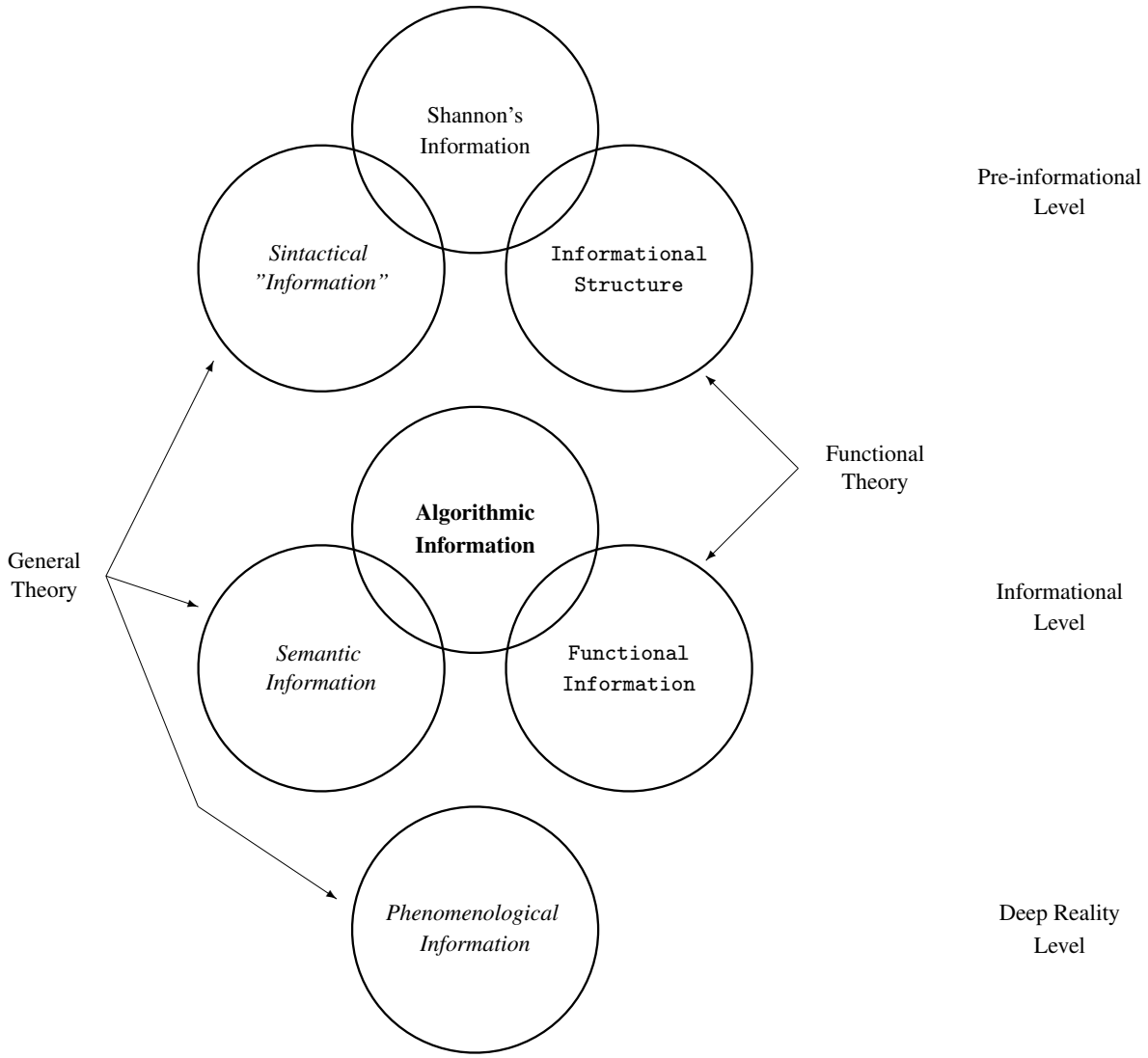

Figure 3.7: The levels of information

Part II

The Parallel Engine

Chapter 4

What Means Parallel Computation?

A clean way to impose a computational system is to go through the following mandatory steps:

- mathematical computational model
- abstract model
- structure
- architecture
- programming model

4.1 From Kleene's model to MapReduce engine

In this section the way from the Kleene's model to MapReduce engine is presented [Ştefan '14]. In Appendix A the Stephen Kleene's model of partial recursive functions is shortly presented. It can be used as a mathematical model for parallel for parallel computation. In the same appendix there are proved two theorems.

First Theorem : *the primitive recursive rule is reducible to repeated applications of specific compositions* (see Theorem A.1).
◇

Second Theorem : *the minimization (least-search) rule is reducible to repeated applications of specific compositions* (see Theorem A.2).
◇

4.1.1 *Kleene Machine*: a Parallel Model of Computation

Because, according to Theorems A.1 and A.2, only the composition rule must be considered in defining what means (parallel) computation, the following definition is based exclusively on the composition rule.

Definition 4.1 ***Kleene Machine****, KM, which computes any function* $f : \mathbb{N}^n \to \mathbb{N}^m$*, is a composition structured as a two-layer construct (see Figure 4.1) with:*

1. ***map*** *level: populated with the functions*

$$h_i(X, l_{i+1}, r_{i-1}) = \langle y_i, l_i, r_i \rangle$$

for $i = 1, 2, \ldots$

2. ***reduction*** *level: the function*

$$g(y_1,\ldots,y_i,\ldots)=\langle z_i,\ldots,z_m\rangle$$

where: the functions h_i and the function g are initial functions or KMs, y_i are arguments for the function g, while l_i, r_i are arguments for h_i generated by h_{i+1} and h_{i-1}, respectively.

◇

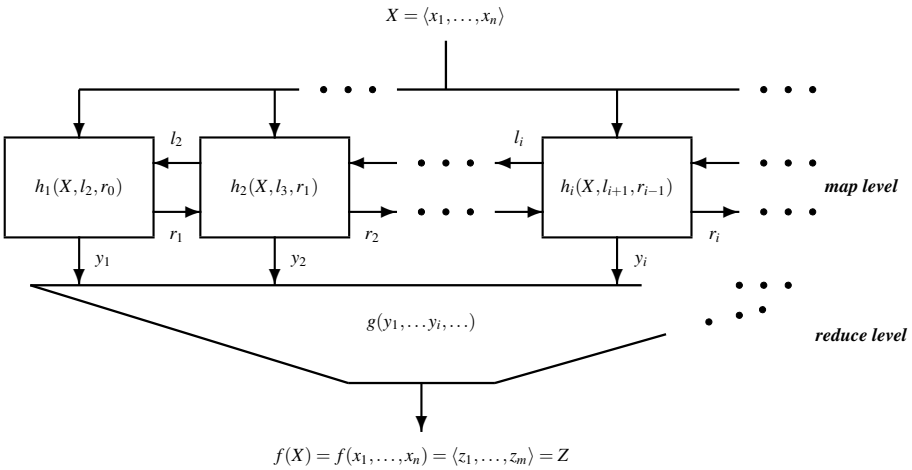

Figure 4.1: **Kleene Machine**. The ***synchronic parallelism*** is performed on the **map** level and the ***diachronic parallelism*** works between the **map** level and the **reduce** level.

The left-right connections between the cells of KM are due to the MOB structure which can be developed in two versions, one already introduced in Definition A.6, and another which can be similarly defined for the left oriented connections.

Because Kleene's model is proved to be mathematically equivalent with the Turing Machine model, the next corollary is true.

Corollary 4.1 ***The Kleene Machine*** *represents a mathematical model for parallel computation with two aspects: the* synchronic *parallelism on the* **map** *level and the* diachronic *(pipelined) parallelism between the two structural levels, the map level and the* **reduce** *level.*

◇

4.1.2 *Universal Kleene Machine*

For each function f there is a KM. As Turing defined [**?**] its Universal Turing Machine, UTM, the concept of KM must be accompanied by the concept of ***Universal Kleene Machine***, UKM. An UKM must provide the possibility (1) to define any KM on the same structure, and (2) to compose KMs.

Definition 4.2 ***Universal Kleene Machine*** *(Figure 4.2) is a finite KM, with p cells on the map-level, loop connected with a Counter-Extended Finite-State Automaton, CFA [**?**] (see Figure **??**), having access to a non-finite Memory addressed by the non-finite counter of CFA. The* **map-level** *of the finite KM contains p identical cells, $C_1,\ldots,C_p$, each having the function:*

$$C_i(h_i,X,r_{i-1},l_{i+1})=H_{h_i}(X,r_{i-1},l_{i+1})=\langle y_i,l_i,r_i\rangle$$

with $h_i\in\{0,1,\ldots,q-1\}$, $X=\langle x_1,\ldots,x_j,\ldots,x_n\rangle$, $x_j\in\mathbb{N}$, for $j=1,\ldots,n$, $l_i,r_i\in\mathbb{N}$ the left and right outputs of the cells, and $y_i\in\mathbb{N}$, for $i=1,\ldots,p$; while the **reduction-level** *performs the function:*

$$G_g:\mathbb{N}^p\to\mathbb{N}$$

selected by:

$$R(g,Y)=SEL(g,G_0(Y),G_1(Y),\ldots,G_{r-1}(Y))$$

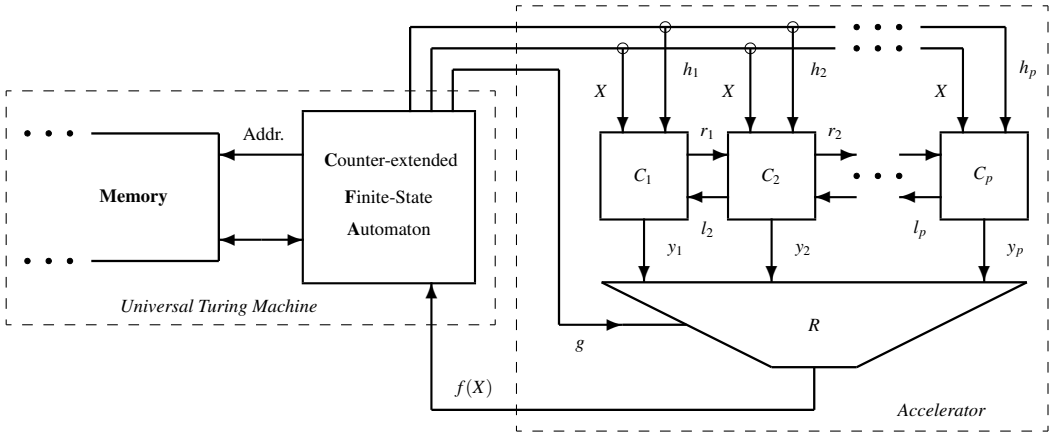

Figure 4.2: **Universal Kleene Machine**: seen as an Accelerated Universal Turing Machine.

with $Y = \langle y_1, y_2, \ldots, y_p \rangle$, $g \in \{0, 1, \ldots, r-1\}$; where h_i and g select functions from the following two finite sets of functions

$$\mathbb{H} = \{H_0, H_1, \ldots, H_{q-1}\}$$

$$\mathbb{G} = \{G_0, G_1, \ldots, G_{r-1}\}$$

representing the characteristic set of functions, $\mathbb{F} = \mathbb{H} \cup \mathbb{G}$, used to compose, starting from the the initial functions of the composition rule, any computable function.

Formally:

$$UKM = (S, \mathbb{A}, S_0, \lambda)$$

where:

- *S is the finite states set of the automaton in CFA*
- *$\mathbb{A} = \mathbb{H} \cup \mathbb{G} \cup \mathbb{N}$ is the alphabet of the UKM*
- *$S_0 \in S$ is the initial state of the automaton in CFA*
- *λ is the transition function*

  $$\lambda : S \times \mathbb{N} \times (\mathbb{N}^p \times \mathbb{N} \times \mathbb{N}^n) \to S \times \mathbb{N}^p \times \mathbb{N} \times \mathbb{N}^n \times \mathbb{N}$$

  which, in each cycle, ***according to****:*

  1. *the current state of the automaton in CFA*
  2. *the output of the reduction function*
  3. *the "instruction" read from Memory having the following fields:*
     (a) *the p indexes for selecting the elements from $\mathbb{H}$ for the map level*
     (b) *the index for selecting one element of $\mathbb{G}$ for the reduction level)*
     (c) *data (the sequence $X \in \mathbb{N}^n$) read from Memory*

  generates*:*

  1. *the next state of the automaton in CFA*
  2. *a sequence of codes for the p functions of the map level, $\langle h_1, h_2, \ldots, h_p \rangle$, provided by Memory or generated by CFA*

3. *the code of the function for the reduce level, g, provided by Memory or generated by CFA*
4. *the X sequence of n integers for the map level, $\langle x_1, x_2, \ldots, x_n \rangle$, provided by Memory or generated by CFA*
5. *the element from $\mathbb{N}$ to be written back in Memory.*

The memory is organized in words of $p+1+n$ integers in read mode and in one-integer words for write mode. The UKM is initialized in the state S_0, and after a finite number of cycles, if the computation is possible, the automaton in CFA stops in a final state.

◇

In the structure of UKM it is easy to segregate the structure of a Turing Machine (which is instantiated as an Universal Turing Machine). Therefore, UKM can be defined also as an UTM working with an accelerator build as a finite KM, because the "infinite"-ness of KM is emulated in the "infinite" Memory, loop connected with CFA.

4.2 *Map-Reduce Abstract Machine* Model for Parallel Computing

From the UKM, as a mathematical model for parallel computation, to an abstract model for parallel computation able to support an actual implementation, few simplifying steps are needed. They are not formally sustained by rigorous proofs. The purpose of this transition is motivated by the transition from a *competent* model to a model which is also able to attain high *performance*.

Definition 4.3 *A computation model is **competent** if the computation it supports ends in a finite number of steps.*

◇

Definition 4.4 *A computation model is **performant** if the computation it supports ends in a minimal number of steps.*

◇

The road from competence to performance requires engineering work. The result is validated by the evaluation of the resulting performance.

4.2.1 Forms of Parallelism

Five forms of simplified parallelism (see Figure 4.3) are emphasized as the meaningful set of particular compositions able to provide the transition from a competent model to a performant one.

Definition 4.5 Data-parallel *computation is defined for MC computation (see Definition A.2) when $n = m$ with $h_i(x_1, \ldots, x_n) = h(x_i)$, for $i = 1, \ldots, n$.*

◇

The same function, h, is applied in parallel to each component, x_i, of the input vector.

Definition 4.6 Reduction-parallel *computation is defined for RC computation (see Definition A.3) when $n = m$ with $h_i(x_1, \ldots, x_n) = h(x_i) = x_i$, for $i = 1, \ldots, n$.*

◇

On the first level of the composition, the map level, all the functions of one variable, x_i, perform the identity function.

Definition 4.7 Speculative-parallel *computation is defined for MC computation when $n = 1$.*

◇

Each function h_i has the same input variable x_1.

Definition 4.8 Thread-parallel *computation is defined for MC computation when $n = m$ with $h_i(x_1, \ldots, x_n) = h_i(x_i)$, for $i = 1, \ldots, n$.*

◇

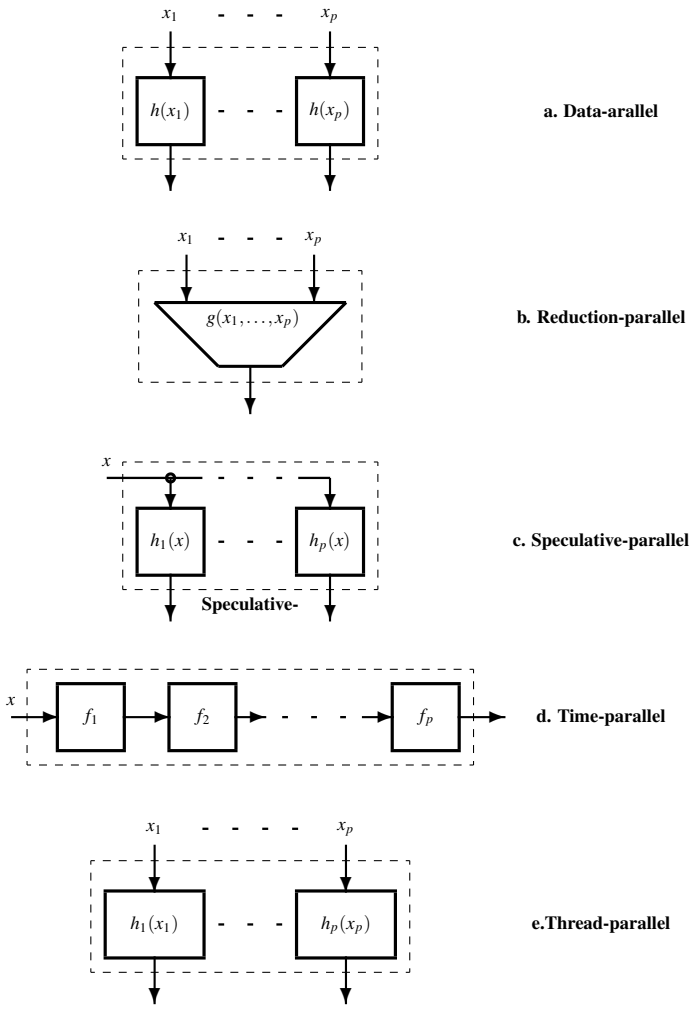

Figure 4.3: Five types of parallelism as particular forms of composition (see Figure 4.2)

Each cell performs a specific function on different data.

Definition 4.9 Time-parallel *computation is defined for repeated application of the composition rule with* $m = n = 1$.
◇

The repeated application of time-parallel computation provides the following pipe of functions:

$$f(x) = f_p(f_{p-1}(f_{p-2}(\ldots f_1(x)\ldots)))$$

4.2.2 Integral Parallelism

We claim that the previous five forms cover efficiently the most frequent parallel computation patterns. Integrating them on a single engine provides the ***parallel abstract model*** for computation. In Figure 4.4, the MapReduce recursive parallel abstract model for parallel computation is presented. It consists of:

- pairs *eng-mem* in the MAP section; they correspond to the cells C_i from UKM, and consist of:
  - *eng*, the engine, which is an execution unit or a processing unit

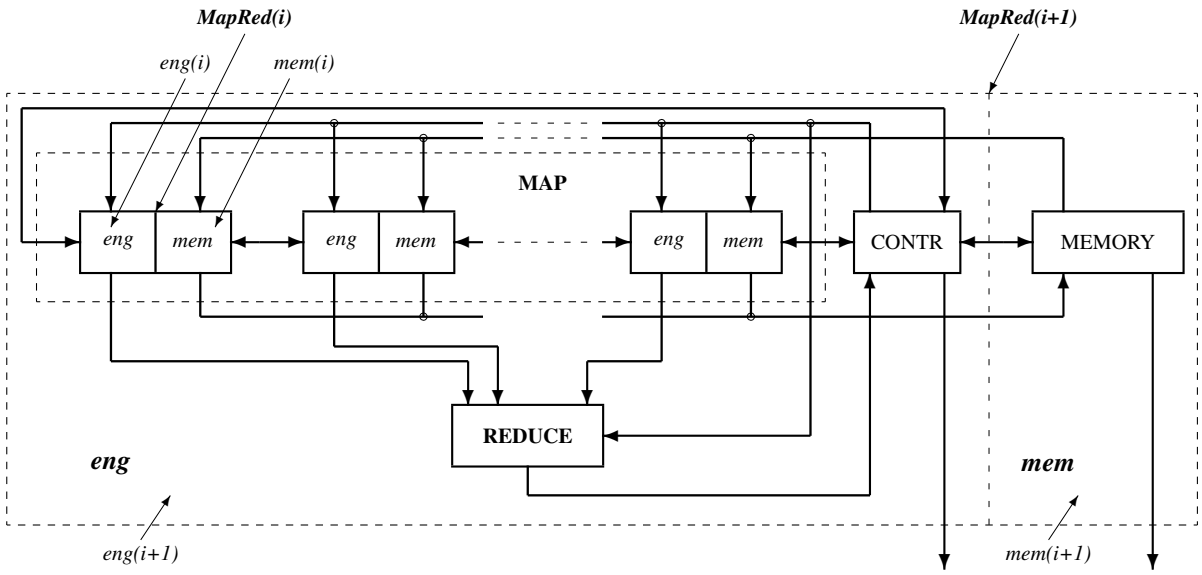

Figure 4.4: ***MapReduce recursive abstract model*** for parallel computation.

  - *mem*, the local memory to store data (when *eng* are execution units) or data and programs (when *eng* are processing units)

- REDUCE unit; it corresponds to the *R* function in UKM
- CONTR, a controller used as sequencer; performs the function of FSM from UKM
- MEMORY, a memory resource for data and programs.

The entire structure from Figure 4.4 can be seen as a two-part entity:

- ***eng***: MAP + REDUCE + CONTR
- ***mem***: MEMORY

which behaves as a cell in a ***recursive hierarchy*** of a map-reduce organization of many-core coomputation.

4.3 A Programming Model

4.3.1 Backus' Functional Forms

Although Backus's concept of *Functional Programming Systems* (FPS) was introduced as an alternative to the *von Neumann style of programming* in [Backus '78], we claim that **they can be seen also as a *low level description* for the parallel computing paradigm**. In the following we use a FPS-like form to provide a low level functional description for the abstract model defined in the previous section. Thus, we obtain the ***virtual machine*** description of a parallel computer, i.e., the description defining the transparent interface between the hardware system and the software system in a real parallel computer. Starting from this virtual machine, the actual ***instruction set architecture*** could be designed for the physical embodiment of various parallel engines.

This section provides, following [Backus '78], the low level description for what we call Integral Parallel Machine (IPM). It contains functions which map objects into objects, where an object could be:

- atom, x; special atoms are: T (true), F (false), ϕ (empty sequence)

- sequence of objects, $< x_1, \ldots, x_p >$, where x_i are atoms or sequences
- $\perp$: undefined object

The set of functions contains:

- **primitive functions**: the functions performed atomically, which manage:
  - atoms, using functions defined on constant length sequences of atoms, returning constant length sequence of atoms
  - p-length sequences, where p is the number of cells of the MANY-CORE section
- **functional forms** for:
  - expanding to sequences the functions defined on atoms
  - defining new functions
- **definitions**: the programming tool used for developing applications.

Primitive Functions

An informal and partial description of a set of primitive functions follows.

- **Atom** : if the argument is an atom, then T is returned, else F is returned.

  $$atom : x \equiv (x \; is \; an \; atom) \rightarrow T; F$$

  The function is performed by the controller or at the level of each c_i cell if the function is applied to each element of a sequence (see *apply to all* in the next subsection).

- **Null** : if the argument is the empty sequence, it returns T, else F.

  $$null : x \equiv (x = \phi) \rightarrow T; F$$

  It is a reduction-parallel function performed by the reduction/loop network, *redLoopNet* (see Figure **??**), which returns a predicate to the controller.

- **Equals** : if the argument is a pair of identical objects, then returns T, else F.

  $$eq : x \equiv ((x =< y, z >) \; \& \; (y = z)) \rightarrow T; F$$

  If the argument contains two atoms, then the function is performed by the controller, else, if the argument contains two sequences, the function is performed in the cells c_i, and the final results is delivered to the controller through *redLoopNet*.

- **Identity** : is a sort of *no operation* function which returns the argument.

  $$id : x \equiv x$$

- **Length** : returns an atom representing the length of the sequence.

  $$length : x \equiv (x =< x_1, \ldots, x_i >) \rightarrow i; (x = \phi) \rightarrow 0; \perp$$

  If the sequence is distributed in the MANY-CELL array, then a Boolean sequence, $< b_1, \ldots, b_p >$, with 1 on each position containing a component x_j is generated and *redLoopNet* provides $\sum_1^p b_j$ for the controller.

- **Selector** : if the argument is a sequence with no less than i objects, then the i-th object is returned.

$$i : x \equiv ((x =< x_1, \ldots, x_p >) \,\&\, (i \leq p)) \rightarrow x_i$$

The function is performed composing an intense speculative-parallel search operation with a data-parallel mask operation and the reduction-parallel OR operation which sends to the controller the selected object.

- **Delete** : if the first argument, k, is a number no bigger than the length of the second argument, then the k-th element in the second argument is deleted.
$del : x \equiv (x =< k, < x_1, \ldots, x_p >>) \,\&\, (k \leq p) \rightarrow$
$< x_1, \ldots, x_{k-1}, x_{k+1}, \ldots >$
The *ORprefix* circuit included in the *redLoopNet* subsystem selects the sequence $< x_k, x_{k+1}, \ldots >$, then the left-right connection in the MANY-CELL array is used to perform a one position left shift in the selected sub-sequence.

- **Insert data** : if the second argument, k, is a number no bigger than the length of the third argument, then the first argument is inserted in the k-th position in the last argument.
$ins : x \equiv (x =< y, k, < x_1, \ldots, x_p >>) \,\&\, (k \leq p) \rightarrow$
$< x_1, \ldots, x_{k-1}, y, x_k, \ldots >$
The *ORprefix* function performed in the *redLoopNet* subsystem selects the sequence $< x_k, x_{k+1}, \ldots >$, then the left-right connection in the MANY-CELL array is used to perform one position right shift in the selected sub-sequence and write y in the freed position.

- **Rotate** : if the argument is a sequence, then it is returned rotated one position left.

$$rot : x \equiv (x =< x_1, \ldots, x_p >) \rightarrow < x_2, \ldots, x_p, x_1 >$$

The *redLoopNet* subsystem and the left-right connection in the MANY-CELL array allows this operation.

- **Transpose** : the argument is a sequence of sequences which can be seen as a two-dimension array. It returns a sequence of sequences which represents the transposition of the argument matrix.
$trans : x \equiv$
$(x =<< x_{11}, \ldots, x_{1m} >, \ldots, < x_{n1}, \ldots, x_{nm} >>) \rightarrow$
$<< x_{11}, \ldots, x_{n1} >, \ldots, < x_{1m}, \ldots, x_{nm} >>$
There are two possible implementations. First, it is naturally solved in the MANY-CELL section because, loading each component of x "horizontally", as a sequence in *Buffer*, we obtain, associated to each cell c_i, the n-component final sequences on the "vertical" dimension (see paragraph 3.2.3):

$$\begin{array}{c} < x_{11}, \ldots, x_{n1} > \text{ accessed by } c_1 \\ < x_{12}, \ldots, x_{n2} > \text{ accessed by } c_2 \\ \ldots \\ < x_{1m}, \ldots, x_{nm} > \text{ accessed by } c_m \end{array}$$

where each initial sequence is a m-variable "line" and each final sequence is n-variable "column" in **Buffer**. Second, using rotate and inter sequence operations.

- **Distribute** : returns a sequence of pairs; the i-th element of the returned sequence contains the first argument and the i-th element of the second argument.
$distr : x \equiv (x =< y, < x_1, \ldots, x_p >>) \rightarrow$
$<< y, x_1 >, \ldots, < y, x_p >>$
The function is performed in two steps: (1) generates the p-length sequence $< y, \ldots, y >$, then (2) performs $trans << y, \ldots, y >, < x_1, \ldots, x_p >>$.

- **Permute** : the argument is a sequence of two equally length sequences; the first defines the permutation, while the second is submitted to the permutation.
$perm : x \equiv$

$(x =<< y_1,\ldots,y_p >,< x_1,\ldots,x_p >>) \rightarrow$
$< x_{y_1},\ldots,x_{y_p} >$
With no special hardware support it is performed in time $O(p)$. An optimal implementation, in time belonging to $O(log\, p)$, involves a *redLoopNet* containing a Waksman permutation network, with $< y_1,\ldots,y_p >$ used to program it.

- **Search** : the first argument is the searched object, while the second argument is the target sequence; returns a Boolean sequence with T on each match position.
$src : x \equiv (x =< y,< x_1,\ldots,x_p >>) \rightarrow$
$< (y = x_i),\ldots,(y = x_p) >$
It is an intense speculative-parallel operation. The scalar y is issued by the controller and it is searched in each cell generating a Boolean sequence, distributed along the cells c_i in MANY-CELL, with T on each match position and F on the rest.

- **Conditioned search** : the first argument is the searched object, the second argument is the target sequence, while the third argument is a Boolean sequence (usually generated in a previous search or conditioned search); the search is performed only in the positions preceded by T in the Boolean sequence; returns a Boolean sequencer with T on each conditioned match position.
$csrc : x \equiv$
$(x =< y,< x_1,\ldots,x_p >,< b_1,\ldots,b_p >>) \rightarrow$
$< c_1,\ldots,c_p >$
where: $c_i = ((y = x_i)\ \&\ b_{i-1})\ ?\ T : F$.
The combination of `src` or `csrc` allows us to define a `sequence_search` operation (an application is described in [**?**]).

- **Arithmetic & logic operations** :

$$op2 : x \equiv ((x =< y,z >)\ \&\ (y,z\ atoms)) \rightarrow y\, op2\, z$$

where: $op2 \in \{add, sub, mult, eq, lt, gt, leq, and, or, ...\}$
or

$$op1 : x \equiv ((x = y)\ \&\ (y\ atom)) \rightarrow op1\, y$$

where: $op1 \in \{inc, dec, zero, not\}$. These operations will be applied on sequences of any length using the functional forms defined in the next sub-section.

- **Constant** : generates a constant value.

$$\bar{x} : y \equiv x$$

Functional Forms

A functional form is made of functions that are applied to objects. They are used to define complex functions, for an IPM, starting from the set of primitive functions.

- **Apply to all** : represents the ***data-parallel*** computation. The same function is applied to all elements of the sequence.

$$\alpha f : x \equiv (x =< x_1,\ldots,x_p >) \rightarrow < f : x_1,\ldots,f : x_p >$$

Example:
$\alpha\, add :<< x_1,y_1 >,\ldots,< x_p,y_p >> \rightarrow$
$< add :< x_1,y_1 >,\ldots,add :< x_p,y_p >>$
expands the function add, defined on atoms, to be applied on sequences, $<< x_1,\ldots,x_p >< y_1,\ldots,y_p >>$, transposed in a sequence of pairs $< x_i,y_i >$.

- **Insert** : represents the ***reduction-parallel*** computation. The function f has as argument a sequence of objects and returns an object. Its recursive form is:
  $/f : x \equiv ((x =< x_1, \ldots, x_p >) \& (p \geq 2)) \rightarrow$
  $f :< x_1, /f :< x_2, \ldots, x_p >>$
  The resulting action looks like a sequential process executed in $O(p)$ cycles, but on the Integral Parallel Abstract Model (see Figure **??**) it is executed as a reduction function in $O(log\ p)$ steps in the *redLoopNet* circuit.

- **Construction** : represents the ***speculative-parallel*** computation. The same argument is used by a sequence of functions.
  $$[f_1, \ldots, f_n] : x \equiv < f_1 : x, \ldots, f_n : x >$$

- **Composition** : represents ***time-parallel*** computation if the computation is applied to a stream of objects. By definition:
  $(f_q \circ f_{q-1} \circ \ldots \circ f_1) : x \equiv$
  $f_q : (f_{q-1} : (f_{q-2} : (\ldots : (f_1 : x) \ldots)))$
  The previous form is:

  - sequential computation, if only one object x is considered as input variable
  - pipelined *time-parallel* computation, if a ***stream*** of objects, $|x_n, \ldots, x_1|$, are considered to be inserted, starting with x_1, in c_1 in the MANY-CORE section (see Figure **??**) so as in each successive two cells, c_i and c_{i+1}, are performed
    $$f_i(f_{i-1} : (f_{i-2} : (\ldots : (f_1 : x_j) \ldots)))$$
    $$f_{i+1}(f_i : (f_{i-1} : (\ldots : (f_1 : x_{j-1}) \ldots)))$$
    Thus, the array of cells $c_1, \ldots, c_p$ can be involved to compute in parallel the function
    $$f(x) = (f_q \circ f_{q-1} \circ \ldots \circ f_1) : x$$
    for maximum q values of x.

- **Threaded construction** : is a special case of construction for: $f_i = g_i \circ i$ which represents the ***thread-parallel*** computation:
  $\theta[f_1, \ldots, f_p] : x \equiv$
  $(x =< x_1, \ldots, x_p >) \rightarrow < g_1 : x_1, \ldots, g_p : x_p >$
  where: $g_1 : x_1$ represents an independent thread.

- **Condition** : represents a conditioned execution.
  $(p \rightarrow f; g) : x \equiv$
  $((p : x) = T) \rightarrow f : x; ((p : x) = F) \rightarrow g : x$

- **Binary to unary** : is used to express any function as an unary function.
  $$(bu\ f\ x) : y \equiv f :< x, y >$$
  This function allows the algebraic manipulation of programs.

Definitions

Definitions are used to write programs conceived as functional forms.

$$\textbf{Def}\ new\_function\_symbol \equiv functional\_form$$

Example : Let be the following definitions used to compute the *sum of absolute difference* (SAD) of two sequence of numbers:

Def $SAD \equiv (/+) \circ (\alpha ABS) \circ trans$
Def $ABS \equiv lt \rightarrow (sub \circ REV); sub$
Def $REV \equiv (bu\ perm < \bar{2}, \bar{1} >)$

4.3.2 Kleene – Backus Synergy

The beauty of the relation between the abstract machine components resulting from Kleene's model and the FPS proposed by Backus is that all the five meaningful forms of composition correspond to the main functional forms, as follows:

Kleene's parallelism $\leftrightarrow$ **Backus's functional forms**
data-parallel $\leftrightarrow$ `apply to all`
reduction-parallel $\leftrightarrow$ `insert`
speculative-parallel $\leftrightarrow$ `construction`
time-parallel $\leftrightarrow$ `composition`
thread-parallel $\leftrightarrow$ `threaded construction`

Let us agree that Kleene's model, and the FPS proposed by Backus represent a solid foundation for parallel computing, avoiding risky *ad hoc* constructs. The generic parallel structure proposed in the next section is a promising start in saving us from saying "Hail Mary" (see [**?**]) when we decide what to do in order to improve our computing machines with parallel features.

4.3.3 Lisp-like MapReduce Functional Language

A low level programming environment, called Backus-Connex Parallel FP system – BC for short –, was defined in Scheme for this generic parallel engine (see [**?**]). Some of the most used functions working on the previously defined array A are listed below:

```
(SetVector a v); a: address, v: vector content
(UnaryOp x)    ; x: scalar|vector
(BinaryOp x y) ; (x,y): scalar | vector
(Cond x y)     ; (x,y): scalar | vector
(RedOp v)      ; RedOp = {RedAdd, RedMax,...}
(ResetActive)  ; activate all cells
(Where b)      ; active where vector b is 1
(ElseWhere)    ; active where vector b was 0
(EndWhere)     ; return to previous active
```

Let us take as example the function *conditioned reduction add*, *CRA*, which returns the sum of all the components of the sequence $s_1 =< x_{11},\ldots,x_{1p} >$ corresponding to the positions where the element in the sequence $s_2 =< x_{21},\ldots,x_{2p} >$ is *less or equal than* the element of the sequence $s_3 =< x_{31},\ldots,x_{3p} >$:

$$CRA(s_1,s_2,s_3) = \sum_{i=1}^{p} (x_{2i} \leq x_{3i})\,?\,x_{1i} : 0$$

The computation of this function is expressed as follows:

$$\textbf{Def}\, CRA \equiv (/+) \circ (\alpha((leq \circ (bu\ del 1)) \rightarrow (id \circ 1); \bar{0})) \circ trans$$

where the argument must be a sequence of three sequences:

$$x =< s_1, s_2, s_3 >$$

and the result is returned as an atom. For

$$x =<< 1,2,3,4 >, < 5,6,7,8 >, < 8,7,6,5 >>$$

the evaluation is the following:
$CRA : x \Rightarrow$
$(/+) \circ (\alpha((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1); \bar{0})) \circ trans :$
$<< 1,2,3,4 >, < 5,6,7,8 >, < 8,7,6,5 >> \Rightarrow$
$(/+) \circ (\alpha((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1); \bar{0})) :<< 1,5,8 >, < 2,6,7 >, < 3,7,6 >< 4,8,5 >> \Rightarrow$
$(/+) :<$

$((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1);\bar{0}) :< 1,5,8 >,$
$((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1);\bar{0}) :< 2,6,7 >,$
$((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1);\bar{0}) :< 2,6,7 >,$
$((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1);\bar{0}) :< 4,8,5 >> \Rightarrow$
$(/+) :< ((leq :< 5,8 >) \rightarrow (id : 1);\bar{0}), \ldots, ((leq :< 8,5 >) \rightarrow (id : 4);\bar{0}) > \Rightarrow$
$(/+) :< ((leq :< 5,8 >) \rightarrow 1;\bar{0}), \ldots, ((leq :< 8,5 >) \rightarrow 4;\bar{0}) > \Rightarrow$
$(/+) :< (T \rightarrow 1;0), (T \rightarrow 2;0), (F \rightarrow 3;0), (F \rightarrow 4;0) > \Rightarrow$
$(/+) :< 1,2,0,0 > \Rightarrow 3$

At the level of machine language the previous program is translated into the following BC code:

```
(define (CRA v0 v1 v2 v3)
    (Where (Leq (Vec v2) (Vec v3)))
        (SetVector v0 (Vec v1))
      (ElseWhere)
        (SetVector v0 (MakeAll 0))
    (EndWhere)
    (RedAdd (Vec v0))
)
```

The function `CRA` returns a scalar and has as side effect the updated content of the vector `v0`.

4.3.4 Backus-type MapReduce Functional Language

The language describe the computation of an accelerator in a hybrid computing system. The system consists of HOST and ACCELERATOR interconnected by INTERFACE. The program runs mainly on ACCELERATOR. Only the transfer functions are controlled by HOST.

$$\textbf{Def}\ FUNC \equiv OP1 \circ OP2 \circ \ldots \circ OPn$$

If, `FUNC<<parameter_1>...<parameter_m>>` then, the function `OPn` must be defined on `<<parameter_1>...<parameter_m>>`, it must let, for the next function `OP(n-1)`, an appropriate number of parameters, and so on.

Example 4.1 *Functions belonging to the matrix subset:*

```
// on ACCELERATOR
MATMULT<<source><source><dest>>

<source> | <dest>:  <MAT<lines, columns, vectorAddress>> |
                    <EXTMAT<lines, columns, scalarAddress>>
// on HOST
LOADMAT<lines, columns, scalarAddress> // HOST loads inFifo from scalarAddress,
    // INTERFACE loads the matrix from inFifo in ACCELERATOR
STOREMAT<lines, columns, scalarAddress> // INTERFACE load outFifo
    // HOST store the matrix at scalarAddress in external memory
```

The program which multiplies an internally stored matrix with an externally stored matrix and stores back the result in the external memory:

```
MATMULT<MAT<lines, columns, vectorAddress>
        EXTMAT<lines, columns, vectorAddress>
        EXTMAT<lines, columns, vectorAddress>
       >
LOADMAT<lines, columns, scalarAddress>  // load the second operand
STOREMAT<lines, columns, scalarAddress> // store the result
```

$\diamond$

Chapter 5

The Generic Parallel Engine

5.1 The General Description of the Hybrid System

The structure of the hybrid system we consider (see Figure 5.1) consists of:

- HOST SYSTEM: a general purpose computing system with Harvard architecture
- ACCELERATOR: a parallel engine

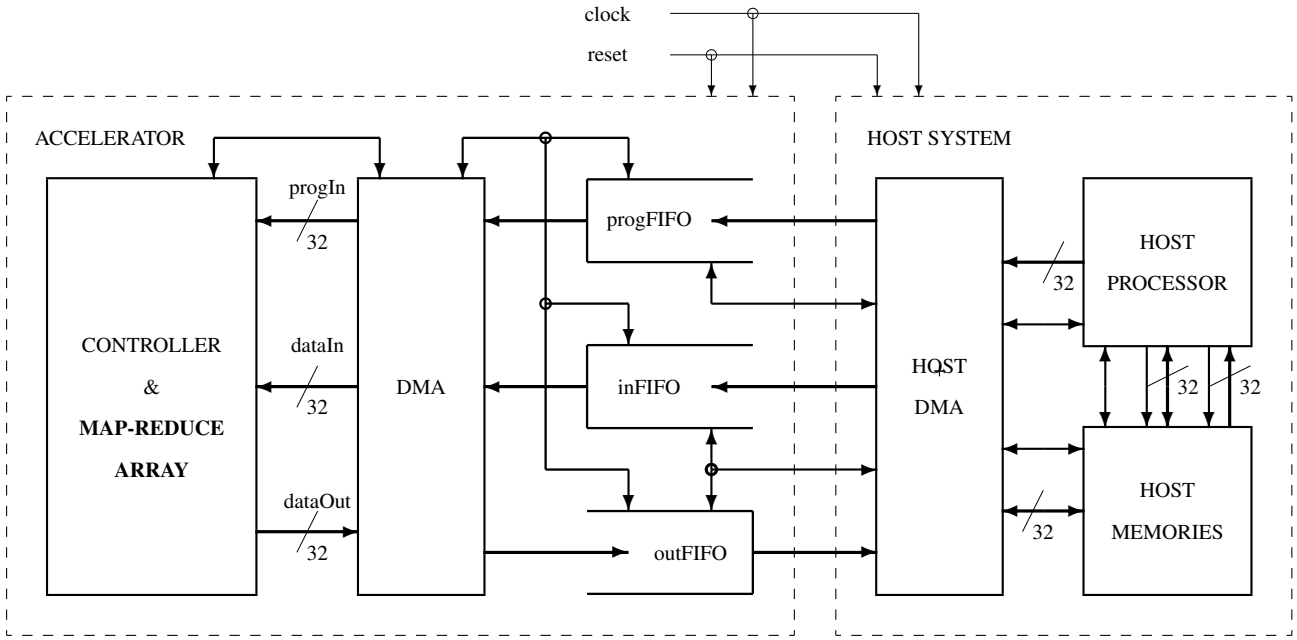

Figure 5.1: The hybrid system: HOST SYSTEM & ACCELERATOR.

The HOST SYSTEM is supposed to run a complex part of the program stored in its program memory, part of MEMORY, while the intense part of the program runs in ACCELERATOR. The intense part of the program and the associated data, stored in the data section of MEMORY, are transferred between HOST SYSTEM and ACCELERATOR.

5.2 Host System

5.2.1 Host System's Structure

The host system (see Figure 5.1) consists of:

- HOST PROCESSOR, a simple, general purpose RISC processor with Harvard architecture.
- HOST DMA, an interface with ACCELERATOR having two counter-extended automata:
  - one to control the program and commands transfer from HOST PROCESSOR system to ACCELERATOR
  - another to manage data transfer between ACCELERATOR and data section of MEMORY
- MEMORY, the memory system containing two memories, one for programs and another for data.

5.2.2 The Host's Instruction Set Architecture

The storage resources of the host system are:

- host program memory (part of MEMORY):
  `reg [31:0] hostProgMem[0:1023]`
- host data memory (part of MEMORY):
  `reg [31:0] hostDataMem[0:1023]`
- register file:
  `reg [31:0] rf[0:31]`
- program counter:
  `reg [31:0] pc`
- cycles counter:
  `reg [31:0] hostCounter`

The instruction set architecture is a typical RISC one, as follows:

```
/*****************************************************************************
File name: 01_hostISA.v
INSTRUCTION SET ARCHITECTURE
    reg [15:0]  pc              ; // program counter
    reg [31:0]  rf[0:31]        ; // register file

    inst[31:0] =
        {0, opCode[4:0], dest[4:0], left[4:0], value[15:0]} |
        {1, opCode[4:0], dest[4:0], left[4:0], right[4:0], noUse[10:0]};

    rigthOp = inst[31] ? rf[right] : {{16*{value[15]}}, value[15:0]}
*****************************************************************************/
parameter   hval    = 1'b0  , // value is the right operand
            rfr     = 1'b1  ; // rf[right] is the right operand

parameter
                        // for inst[31] = x
    hadd        = 5'b00000, // rf[dest] = (rf[left] + rigthOp)[31:0]
    hsub        = 5'b00001, // rf[dest] = (rf[left] - rigthOp)[31:0]
    haddcr      = 5'b00010, // rf[dest] = (rf[left] + rigthOp)[32]
```

```
    hsubcr      = 5'b00011, // rf[dest] = (rf[left] - rigthOp)[32]
    hmult       = 5'b00100, // rf[dest] = rf[left][15:0] * rigthOp[15:0]
    hrsh        = 5'b00101, // rf[dest] = rf[left] >> 1
    harsh       = 5'b00110, // rf[dest] = {rf[left][31], rf[left][31:1]}
    hersh       = 5'b00111, // rf[dest] = {sRigthOp[0], rf[left][31:1]}
    helsh       = 5'b01000, // rf[dest] = {rf[left][30:0], rigthOp[31]}
    hbwand      = 5'b01001, // rf[dest] = rf[left] & rigthOp
    hbwor       = 5'b01010, // rf[dest] = rf[left] | rigthOp
    hbwxor      = 5'b01011, // rf[dest] = rf[left] ^ rigthOp
                            // DATA MOVE
    hsval       = 5'b01100, // rf[dest] = {{16*{rigthOp[15]}}, rigthOp[15:0]}
    huval       = 5'b01101, // rf[dest] = {{16'b0}, rigthOp[15:0]}
    hinsval     = 5'b01110, // rf[dest] = {rf[left][15:0], rigthOp[15:0]}
    hstore      = 5'b01111, // dataMem[rigthOp] = rf[left]
    hget        = 5'b10000, // dataMemOut = dataMem[rigthOp]
    //          = 5'b10001, //
    //          = 5'b10010, //
    hstart      = 5'b10011, // enable hostCounter
    hstop       = 5'b10100, // disable hostCounter
    hload       = 5'b10101, // rf[dest] = dataMemOut
    hpush       = 5'b10110, // send parameter to controller
    htrans      = 5'b10111, // from dataMem[rigthOp] transfer program
    hrjmp       = 5'b11000, // pc = pc + rigthOp[15:0];
    hbrz        = 5'b11001, // pc = (rf[left]=0) ? pc + rigthOp[15:0] : pc+1
    hbrnz       = 5'b11010, // pc = !(rf[left]=0) ? pc + rigthOp[15:0] : pc+1
    hjmp        = 5'b11011, // pc = rigthOp;
    hcall       = 5'b11100, // pc = rigthOp[15:0]; rf[dest] = pc+1;
    hrun        = 5'b11101, // run accelerator's program from rigthOp
    hwaitacc    = 5'b11110, // wait accelerator to enter in idle state
    hwaitint    = 5'b11111; // wait hostInterface to enter idle state
/*****************************************************************************
pseudoInstructions:
    NOP             : 32'b0;
    NOT(dest, left) : {1'b0, bwxor, dest, left, 16'b1111_1111_1111_1111}
    HALT            : {1'b0, rjmp, 26'b0}
    RET(right)      : {1'b1, jmp, 10'b0, right, 11'b0}
    MOVE(dest,left) : {1'b0, add, dest, left, 16'b0}
*****************************************************************************/
```

5.3 Accelerator

5.3.1 Accelerator's Structure

The accelerator (see Figure 5.1) consists of:

- DMA: Direct Memory Access controller which receives programs and commands from Host, through progFIFO, or manages data transfers between MEMORY and ACCELERATOR; it consists of two counter-extended finite automata:
  - one for managing program and commands transfer from HOST to the ACCELERATOR
  - another to manage data transfer between ACCELERATOR and MEMORY and to send messages to HOST
- progFIFO: used to transfer the program which run on ACCELERATOR and to trigger the functions (programs) programmed on ACCELERATOR

- inFIFO: used to receive data from the External Memory
- outFIFO: used to
  - send back to MEMORY the result of computation
  - send requests of data from ACCELERATOR to the external memory MEMORY
  - to send simple messages to HOST (such as end of running the requested function)
- CONTROLLER & MAP-REDUCE ARRAY: is in fact the accelerator.

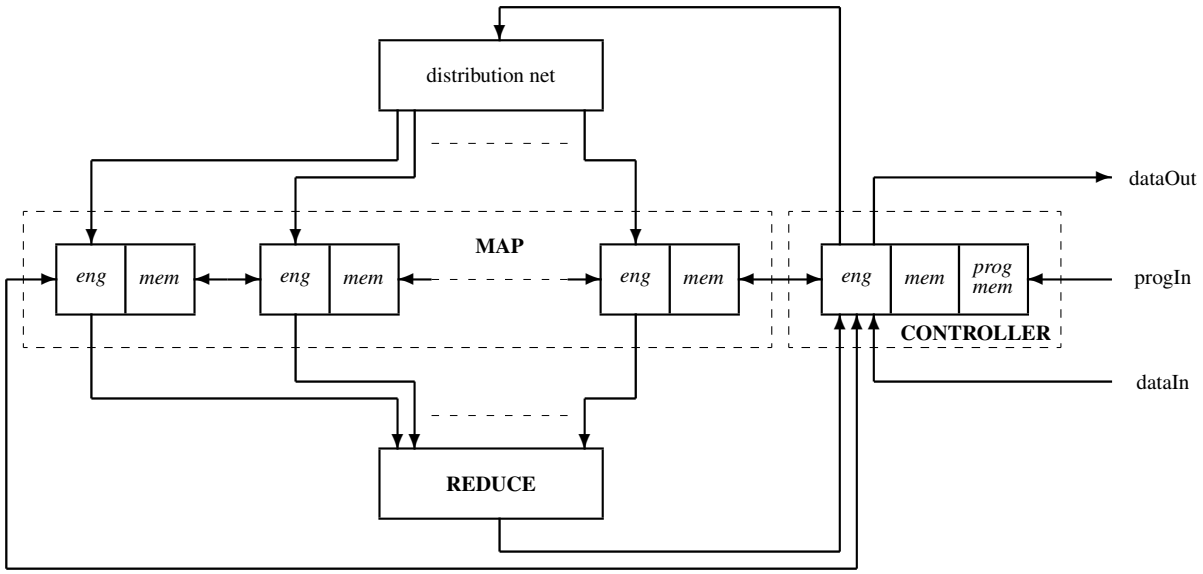

Figure 5.2: The functional organization of the accelerator's core.

The computational part of the accelerator (see Figure 5.2) performs functions dealing with scalar or vectors and consists of a four parts:

- CONTROLLER: performing functions defined on scalars with values in scalars; it has a Hardware RISC architecture with its program memory (*prog mem*), data memory (*mem*) and execution unit (*eng*)
- distribution net: is a *log*-depth pipe-lined network used to distribute instruction, data and address form CONTROLLER to the cells of the MAP section
- MAP section: performing functions defined on vectors with values in vectors; it is a linear array of cells each with its own data memory and execution unit similar with those of the controller
- REDUCTION network: performing functions defined on vectors with values in scalars; it is a *log*-depth circuit.

The parameters used to configure the ACCELERATOR are the following:

```
parameter
    n = 32   , // word size
    x = 10   , // index size -> 2^x = 1024 cells
    v = 11   , // vector memory address size -> 2048 1024-component vectors
    s = 9    , // scalar memory address size -> 512 32-bit scalars
    p = 8    , // program memory address size -> 256 pairs of instructions
    c = 8    , // value size in instruction
    a = 5      // (size of activation counter -> 32 embedded WHEREs)
```

for an engine characterized by:

- 32-bit word
- 1024-cell array
- 2048-word local memory in each cell, which translates in a Vector Memory of 2048 vectors of 1024 32-bit scalar each
- 512-word Data Memory in CONTROLLER
- 256-instruction Program Memory in CONTROLLER
- 8-bit immediate in instruction for both, array and controller.

5.3.2 The User's Architectural Image

The user's image of the accelerator system is presented in Figure 5.3. It consists of the memory resources accessible at the level of the assembly language. There are two levels of storage in the system we simulate:

- Controller's Memory resources are:
  - Accumulator Register: is a 32-bit register in the accumulator-based execution unit; it provides one of the operand and and stores the result of the unary and binary operations performed by the execution unit
    `reg [n-1:0] acc`
  - Carry Bit: is a 1-bit register whose content is actualized at each arithmetic operation (shifts are arithmetic operations)
    `reg cr`
  - Scalar Memory: is the data memory of the controller; it provides, by the rule,, the second operand for binary operations.
    `reg [n-1:0] mem[0:(1<<s)-1]`
  - Address Register: is a register used to form the address for Scalar Memory when relative addressing mode is used; its content is added with the immediate value provided by controller's instruction
    `reg [s-1:0] addr`
  - Programm Memory: contains at each location a pair of instructions, one for CONTROLLER and another for MAP-REDUCE array; it is loaded under the control of DMA unit
    `reg [31:0] progMem[0:(1<<p)-1]`
- Array's Memory resources are:
  - Boolean Vector: is a $p = 2^x$ one-bit words vector; if $b_i = 1$ the $cell_i$ is active, i.e., the instruction received from controller is executed, else, if $b_i = 0$ the $cell_i$ is inactive
    `reg boolVect[0:(1<<x)-1]`

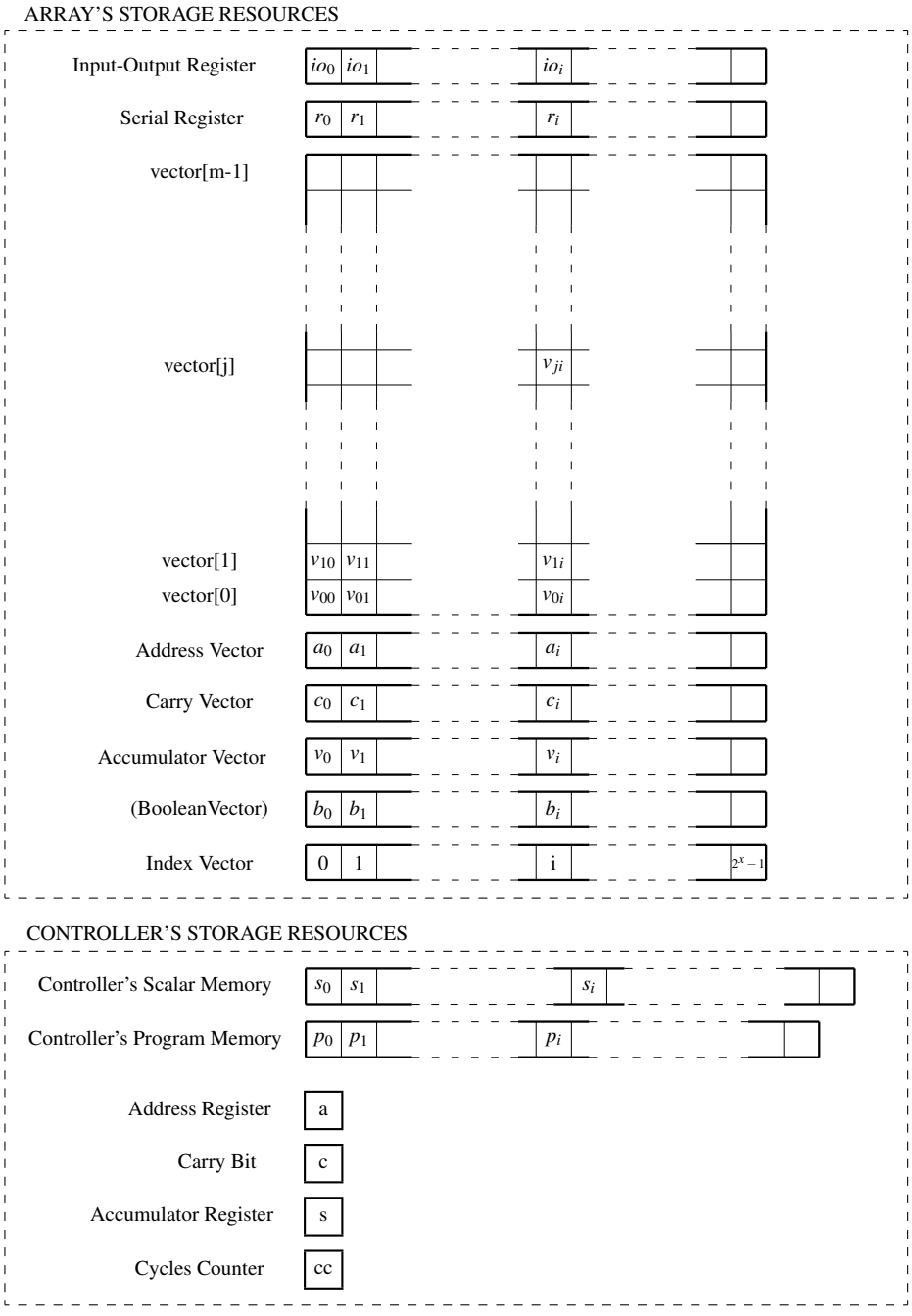

Figure 5.3: The users view of the architecture.

- Accumulator vector: is a p n-bit words vector distributed along the p cells of the MAP section; its components are used as accumulators in the execution units of each cell
  `reg [n-1:]  accVect[0:(1<<x)-1]`
- Carry Vector: is a p one-bit words vector distributed along the p cells of the MAP section; its content is updated at each arithmetic and shift operation
  `reg crVect[0:(1<<x)-1]`
- Address Vector: is a vector distributed along the p cells of the MAP section; it is used for relative addressing the data memory of each cell
  `reg [v-1:0] addrVect[0:(1<<x)-1]`
- Vector Memory: contains $m = 2^v$ p-component vectors
  `reg [n-1:0] vectMem[0:(1<<x)-1][0:(1<<v)-1]`
  as follow
  - vector[0]: `reg [n-1:0] vectMem[0:(1<<x)-1][0]`
  - vector[1]: `reg [n-1:0] vectMem[0:(1<<x)-1][1]`
  - ...
  - vector[i]: `reg [n-1:0] vectMem[0:(1<<x)-1][j]`
  - ...
  - vector[p-1]: `reg [n-1:0] vectMem[0:(1<<x)-1][p-1]`
- Serial Register: is a serial-parallel register distributed along the MAP's cells; each of the p cells contains a n-bit parallel register serially connected in the previous and in the next cell
  `reg [n-1:0] serialReg[0:(1<<x)-1]`
- Index Vector: is a constant vector used to index the p cells of the MAP section
  `reg [x-1:0] ixVect[0:(1<<x)-1]`
- Input-Output Register: is used to insert `inData` or to extract `outData` (see Figure 5.1) in/from array of cells
  `reg [n-1:0] ioReg [0:(1<<x)-1]`

There are the following five operation modes in the storage space just described:

1. ***vector to scalar mode***: is performed in REDUCTION section starting from `accVect` and providing a value in `acc` or back to the MAP section.
   Important note: the REDUCTION unit is a *log*-depth circuit with a latency $\lambda(p) = 1 + 0.5 log_2 p$. Therefore, any scalar generated at the output of the REDUCTION unit is valid with a λ cycles delay, i.e., between the instruction which set the content of `accVect` submitted to a reduction operation and the instruction which uses the result of the reduction operation whatever λ instructions must be inserted; if nothing to do, then no operation instructions are welcome.

2. ***scalar-scalar to scalar mode***: is performed in CONTROLLER between `acc` and `mem[i]` or immediate value contained in instruction or `coOperand` with result in `acc`; `coOperand` is the scalar value received, with λ cycles latency, through REDUCTION unit from MAP section

3. ***vector-scalar to vector***: is performed in MAP section between `accVect` and immediate value contained in instruction or `coOperand` with result in `accVect`; `coOperand` is the scalar value received from CONTROLLER or, with λ cycles latency, from the REDUCTION unit

4. ***vector-vector to vector mode***: is performed in MAP section between `accVect` and `vectMem[j]`

5. ***vector to vector mode***: is performed in MAP section on `accVect`

5.3.3 The Accelerator's Instruction Set Architecture

The initial, generic instruction set is described.

Because the structure of the MapReduce generic engine consists of two programmable parts – the Controller and the Array –, the instruction set architecture, $ISA_{mapReduce}$, is a dual one:

$$ISA_{mapReduce} = (ISA_{controller} \times ISA_{array})$$

where:

- $ISA_{controller} = SS_{arith\&logic} \cup SS_{control} \cup SS_{communication}$ is the ISA associated to the Controller, with three subsets of instructions
- $ISA_{array} = SS_{arith\&logic} \cup SS_{spatialControl} \cup SS_{transfer}$ is the ISA associated to the cellular array, with three subsets of instructions

In each clock cycle from the program memory of the controller a pair of instructions is read: one from $ISA_{controller}$, to be executed by Controller, and another from ISA_{array} to be executed by Array.

The subset $SS_{arith\&logic}$ in the two ISAs – $ISA_{controller}$ and ISA_{array} – are identical. The $SS_{communication}$ subset controls the internal communication between array and controller and the communication of the MapReduce system with the host computer. The $SS_{transfer}$ subset controls the data transfer between the distributed along the cells local memory of the array – `reg [n-1:0] vectMem[0:(1<<x)-1][0:(1<<v)-1]` – and the external memory of the system. The $SS_{control}$ subset consists of conventional control instruction is a standard processor. We must pay more attention to the $SS_{spatialControl}$ subset used to perform the specific spatial control in an array of cells containing execution units or processing units. The main instructions in $SS_{spatialControl}$ subset are:

activate : all the cells of the array are activated for executing the next instructions

where : maintains active only the active cells where the condition `cond` is fulfilled; example: `where (zero)` maintains active only the active cells where the accumulator is zero (such an instruction corresponds to the `if (cond)` instruction form the $SS_{control}$ subset)

elsewhere : activates the cells inactivated by the associated `where (cond)` instruction (it corresponds to the `else` action form the $SS_{control}$ subset)

endwhere : restores the activations existed before the previous `where (zero)` instruction (it corresponds to the `endif` instruction form the $SS_{control}$ subset)

The instruction format for the MapReduce engine allows issuing two instruction at a time, as follows:

```
mrInstruction[31:0] = {controllerInstr, arrayInstr} =
                      {{instr[4:0], operand[2:0], value[7:0]},
                       {instr[4:0], operand[2:0], value[7:0]}}
```

where:

`instr[4:0]` : codes the instruction

`operand[2:0]` : codes source of the second operand used in instruction

`value[7:0]` : is mainly the immediate value or the address

The field `operand[2:0]` is specific for our accumulator centered architecture. It mainly specifies the second n-bit operand, `op`, and has the following meanings:

```
val = 3'b000 : immediate value
     op = {{(n-8){value[7]}}, value[7:0]}
```

`mab = 3'b001` : absolute, from local memory
`op = mem[value]`

`mrl = 3'b010` : relative, from local memory
`op = mem[value + addr]`

`mri = 3'b011` : relative, from local memory and increment the address pointer
`op = mem[value + addr];`
`addr <= value + addr`

`cop = 3'b100` : immediate, with co-operand – `coop`
`op = coop`

`mac = 3'b101` : absolute, from the local memory of each cell, addressed with acc or the controller's operand selected by the send instruction `op[i] = vectMem[(contrOpCode==send) ? op : acc]`

`mrc = 3'b110` : relative, from local memory, using acc or the controller's operand selected by the send instruction `op[i] = vectMem[addr[i] + ((contrOpCode==send) ? op : acc)]`

`ctl = 3'b111` : control instructions

where the co-operand of the array is the accumulator of the controller: `acc`, while the co-operand of the controller is provided by the four outputs of reduction section of the array:

`redSum` : the sum of the accumulators from the active cells: $\Sigma_0^p\ acc_i$

`redMin` : the minimum value of the accumulators from the active cells: $Min_0^p\ acc_i$

`redMax` : the maximum value of the accumulators from the active cells: $Max_0^p\ acc_i$

`redBool` : the sum of the Boolean variable from the active cells: $\Sigma_0^p\ bool_i$

The instruction set architecture is the machine language used to put to work each engine of our computational structure. The codes associated to the fields `instr[4:0]` in the instruction format are listed below:

```
/*****************************************************************************
File name: 01_isa.v
*****************************************************************************/
parameter
    // operand = 000, ..., 110
    // #: for controller only
    // $: for array only
    add         = 5'b00000, //  {cr, acc} <= acc + op;
    addc        = 5'b00001, //  {cr, acc} <= acc + op + cr;
    sub         = 5'b00010, //  {cr, acc} <= acc - op;
    rsub        = 5'b00011, //  {cr, acc} <= op - acc;
    subc        = 5'b00100, //  {cr, acc} <= acc - op - cr;
    rsubc       = 5'b00101, //  {cr, acc} <= op - acc - cr;
    div         = 5'b00110, //  acc <= acc/op;
    rdiv        = 5'b00111, //  acc <= op/acc;
    mult        = 5'b01000, //  acc <= acc * op;
    bwand       = 5'b01001, //  acc <= acc & op;
    bwor        = 5'b01010, //  acc <= acc | op;
    bwxor       = 5'b01011, //  acc <= acc ^ op;
    load        = 5'b01100, //  acc <= op;
    store       = 5'b01101, //  op <= acc;
```

```
pushl       = 5'b01110, //# push left op in the global shift register
pushr       = 5'b01111, //# push right op in the global shift register
send        = 5'b10000, //# send op as coOperand to array (NF2)
compare     = 5'b10001, // cr <= (acc - op)[n]
fadd        = 5'b10010, // float add
madd        = 5'b10011, // float add1
apack       = 5'b10100, // float add2
fmult       = 5'b10101, // float multiply
mpack       = 5'b10110, // float multiply1
fdiv        = 5'b10111, // float divide
mdiv        = 5'b11000, // float divide1
dpack       = 5'b11001, // float divide2
//          = 5'b11010, //
//          = 5'b11011, //
srcall      = 5'b11100, //$ search in all cells
search      = 5'b11101, //$ search is selected cells
csearch     = 5'b11110, //$ conditioned search
insert      = 5'b11111, //$ insert op ar first

// operand = 111; ctl
jmp         = 5'b00000, //# pc <= pc + scalar;
brz         = 5'b00001, //# pc <= acc=0 ? pc + scalar : pc + 1;
brnz        = 5'b00010, //# pc <= acc=0 ? pc + 1 : pc + scalar;
brzdec      = 5'b00011, //# pc <= acc=0 ? pc + scalar : pc + 1; acc <= acc - 1;
brnzdec     = 5'b00100, //# pc <= acc=0 ? pc + 1 : pc + scalar; acc <= acc - 1;
brcr        = 5'b00101, //# pc <= cr ? pc + scalar : pc + 1;
brncr       = 5'b00110, //# pc <= cr ? pc + 1 : pc + scalar;
iowait      = 5'b00111, //# pc <= idleState ? pc + 1 : pc ;
skipeq      = 5'b01000, //# pc <= acc=op ? pc + 2 : pc + 1;
skipneq     = 5'b01001, //# pc <= acc=op ? pc + 1 : pc + 2;
brsgn       = 5'b01010, //# pc <= acc[n-1]=1 ? pc + scalar : pc + 1;
brnsgn      = 5'b01011, //# pc <= acc[n-1]=0 ? pc + 1 : pc + scalar;
brzinc      = 5'b01100, //# pc <= acc+1=0 ? pc + scalar : pc + 1; acc <= acc + 1;
brnzinc     = 5'b01101, //# pc <= acc+1=0 ? pc + 1 : pc + scalar; acc <= acc + 1;
start       = 5'b01110, //# start cycle counter
stop        = 5'b01111, //# stop cycle counter

where       = 5'b00000, //$ &boolVect <= (boolVect & cond[scalar[1:0]]) ? 1 : 0;
wheren      = 5'b00001, //$ &boolVect <= (boolVect & cond[scalar[1:0]]) ? 0 : 1;
elsew       = 5'b00010, //$ boolVect <= ~boolVect;
endwhere    = 5'b00011, //$ boolVect <= 1;
actwhere    = 5'b00100, //$ boolVect <=  (NF3)
saveact     = 5'b00101, //$&act[i] <= act[i] - 1; save activation (NF3)
activate    = 5'b00110, //$ actVect[i] = 0 (NF4)
restact     = 5'b00111, //$&act[i] <= act[i] + 1; restore activation (NF3)
grotate     = 5'b01000, //$ global rotate
glshift     = 5'b01001, //$ global left shift
grshift     = 5'b01010, //$ global right shift
brshift     = 5'b01011, //$ Boolean vector right shift; for read from array
delete      = 5'b01100, //$ delete first
ixload      = 5'b01101, //$ index load: acc[i] <= i
srload      = 5'b01110, //$ serial register load: acc[i] <= serialReg[i]
srstore     = 5'b01111, //$ serial register store: serialReg[i] <= acc[i]

insval      = 5'b10000, // acc <= {acc[23:0], scalar}
```

```
    shrightc    = 5'b10001, // {cr, acc} <= {acc[0], cr, acc[n-1:1]}
    shright     = 5'b10010, // {cr, acc} <= {acc[0], 1'b0, acc[n-1:1]}
    sharight    = 5'b10011, // {cr, acc} <= {acc[0], acc[n-1], acc[n-1:1]}
    penc        = 5'b10100, // {cr, acc} <= {(acc==0), 27'b0, pe(acc)}
    srleft      = 5'b10101, // shift left the content of serial register
    ioload      = 5'b10110, //$ input-output register load: acc[i] <= ioReg[i]
    iostore     = 5'b10111, //$ input-output register store: ioReg[i] <= acc[i]
    insertio    = 5'b11000, //$ insert in ioReg: ioReg <=
                            //                    <= <acc, ioReg[0],..., ioReg[2^x-2]
    iogrsh      = 5'b11001, //$ io right shift: <acc[0],..., acc[2^x-1]> <=
                            //                  <= <ioReg[2^x-1],acc[0],..., acc[2^x-2]>
    popfifo     = 5'b10110, //# acc <= fromInFifo; readInFifo
    //  = 5'b10111, //#
    //  = 5'b11000, //#
    //  = 5'b11001, //#
                            // DATA & PROGRAM TRASNFER INSTRUCTIONS
                            // If DATA TRANSFER codes come unrequested in inFIFO,
                            //   then they are executed by DMA
                            // transferType 3'b000: pop parameter
                            //              3'b001: load
                            //              3'b010: store
                            //              3'b111: accelerator runs halt
                            // PROGRAM TRANSFER codes come always unrequested and
                            //   is executed by DMA
    trun        = 5'b11000, //# transfer run
    lsize       = 5'b11001, //# load the size of vector
    laddr       = 5'b11011, //# load address in external scalar memory
    prun        = 5'b11110, //# program run
    pload       = 5'b11111; //# program load
//*//=============================================================================
```

Part III

Berkeley view of parallel computing

Chapter 6

Berkeley's View

The efficiency of ***Connex System*** in performing all the aspects of intense computation remains to be proved. In this subsection we sketch only the complex process of evaluation using the report "A View from Berkeley" [**?**]. Many decades just an academic topic, "parallelism" becomes an important actor on the market after 2001 when the clock rate race stopped. This research report presents 13 computational motifs which cover the main aspects of parallel computing. Short comments follows about how the proposed architecture and generic parallel engine work for all of the 13 motifs.

For **dense linear algebra** the most used operation is the inner product (IP) of two vectors. It is expressed in FP System as follows:

$$\textbf{Def}\ IP \equiv (/+) \circ (\alpha\times) \circ trans$$

while the BC code is:

```
(define (IP v0 v1)
    (RedAdd (Mult v0 v1))
)
```

allowing a linear acceleration of the computation.

For **sparse linear algebra** the band arrays are first transposed using the function `Trans` in a number of vectors equal with the width w of the band. Then the main operations are naturally performed using the appropriate `RotLeft` and `RotRight` operations. Thus, the multiplication of two band matrices is done on Connex System in $O(w)$.

For **spectral methods** the typical example is FFT. The vertical and horizontal vectors defined in the array A help the programmer to adapt the data representation to obtain an almost linear acceleration [**?**], because the **Scan** module is designed to hide the performance of the matrix transpose operation. In order to eliminate the slowdown caused by the rotate operations, the stream of samples are operated as vertical vectors (see also [**?**], where for example: FFT for 1024 floating point samples is done in less than 1 clock cycle per sample).

***N*-Body method** fits perfect on the proposed architecture, because for $j = 0$ to $j = n-1$ the following equation is computed:

$$U(x_j) = \sum_i F(x_j, X_i)$$

using one cell for each function $F(x_j, X_i)$, followed b the sum (a *reduction* operation).

Structured grids are distributed on the two dimensions of the array A. Each processor is assigned a column of nodes. Each node has to communicate only with a small, constant number of neighbor nodes on the grid, exchanging data at the end of each step. The system works like a cellular automaton.

Unstructured grids problems are updates on an irregular grid, where each grid element is updated from its neighbor grid elements. Parallel computation is disturbed by the non-uniformity of the data distribution. In order to solve the non-uniformity problem a preprocessing step is required to generate an easy manageable representation of the grid. Slow-downs are expected compared with the structured grid.

The typical example of **mapReduce** computation is the Monte Carlo method. This method is highly parallel because it consists in many completely independent computations working on randomly generated data. It requires the add reduction function. The computation is linearly accelerated.

For **combinational logic** a good example is AES encryption which works in 4×4 arrays of bytes. If each array is loaded in one cell, then the processing is pure data-parallel with linear acceleration.

For **graph traversal** in [**?**] are reported parallel algorithms achieving asymptotically optimal $O(|V| + |E|)$ work complexity. Using sparse linear algebra methods, the breadth-first search for graph traversal is expected to be done on a Connex System in time belonging to $O(|V|)$.

For **dynamic programming** the Viterbi decoding is a typical example. The parallel strategy is to distribute the states among the cells. Each state has its own distinct cell. The inter-cell communication is done in a small neighborhood. Each cell receives the stream of data which is thus submitted to a speculative computation. The work done on each processor is similar. The last stage is performed using the reduction functions. The degree of parallelism is limited to the number of states considered by the algorithm.

Parallel **back-track** is exemplified by the SAT algorithm which runs on a p-cell engine by choosing $log_2\ p$ literals, instead of one on a sequential machine, and assigning for them all the values from $00\ldots0$ to $11\ldots1 = p-1$. Each cell evaluates the formula for one value. For parallel **branch & bound** we use the case of the Quadratic Assignment Problem. The problem deals with two $N \times N$ matrices: $A = (a_{ij})$, $B = (b_{kl})$. The global cost function:

$$C(p) = \sum_{i}^{n} \sum_{j}^{n} a_{ij} \times b_{p(i)p(j)}$$

must be minimized finding the permutation p of the set $N = \{1, 2, \ldots, n\}$. Dense linear algebra methods, efficiently running on our architecture, are involved here.

Graphical models are well represented by parallel hidden Markov models. The architectural features reported in research papers refers to fine-grained data-parallel processor arrays connected to each node of a coarse-grained PC-cluster. Thus, our engine can be used efficiently as an accelerator for general purpose sequential engines.

For **finite state machine** (FSM) the authors of [**?**] claim that "nothing helps". But, we consider that the array of cells with their local memory loaded with non-deterministic FSM descriptions work very efficient as a speculative engine for applications such as deep packet inspection, for example.

At the end of this superficial introductory analysis, *which must be deepened by future investigations*, we claim that for almost all the computational motifs the ***Connex System***, in its simple generic form, perform at least encouraging if not pretty well.

Chapter 7

The First Dwarf: Dense Linear Algebra

The problems approached in this chapter are:

- matrix transpose
- matrix-vector multiplication
- matrix-matrix multiplication
- matrix move
- ... (TBD)

7.1 Matrix Transpose

7.1.1 The Algorithm

The algorithm:

```
================================================================================
    size    = N; // the matrix size
    source  = S; // the address of the first vector
    dest    = D; // the address of the first vector

    ixModN[i] = ix - (ix/N)*N;
    cycles = size - 1;
    sAddr[i] = (ixModN[i] - cycles)modN         // compute "diagonal"
    dAddr[i] = (ixModN[i] + cycles)modN         // compute the "opposite diagonal"
    while (cycles > 0) {
        acc[i] <= mem[i][sAddr[i] + source];    // read on "diagonal"
        glshift(cycles);                        // global left shift
        mem[i][dAddr[i] + dest] <= acc;         // store on the "opposite diagonal"
        acc[i] <= mem[i][sAddr[i] + source];    // read on "diagonal"
        grshift(N-cycels);                      // global right shift
        where (ixModN >= N-cycels)
            mem[i][dAddr[i] + dest] <= acc;     // store on the "opposite diagonal"
        endwhere
        sAddr <= (sAddr + 1)modN;               // "increment diagonal"
        dAddr <= (dAddr - 1)modN;               // "decrement diagonal"
        cycles <= cycels - 1;                   // decrement cycles
    }
    move the diagonal;
================================================================================
```

7.1.2 The Program

The program has the following parameters:

```
mem[0] = N: matrix size
mem[1] = S: source matrix (the address of the first vector)
mem[2] = D: destination matrix (the address of the first vector)
```

and the file `matrixTranspose.v` is:

```
/*****************************************************************************************
MATRIX TRANSPOSE

M: the matrix to be transposed stored starting from the addres stored in mem[4]
MT: the transposed matrix stored starting from the addres stored in mem[1]
N: the size of the square matrix stored in mem[0]

Example: the final state of vector memory for:
    - mem[0] = 5;
    - mem[1] = 25;
    - mem[4] = 5
is:

vect[5]  = 0  1  2  3  4  -  - ...
vect[6]  = 0  1  2  3  4  -  - ...
vect[7]  = 0  1  2  3  4  -  - ...
vect[8]  = 0  1  2  3  4  -  - ...
vect[9]  = 0  1  2  3  4  -  - ...
...                     -  - ...
vect[25] = 0  0  0  0  0  -  - ...
vect[26] = 1  1  1  1  1  -  - ...
vect[27] = 2  2  2  2  2  -  - ...
vect[28] = 3  3  3  3  3  -  - ...
vect[29] = 4  4  4  4  4  -  - ...

*****************************************************************************************/
                              // mem[0][i] <= ixModN[i]
        cLOAD(0);   IXLOAD;   // acc <= N; acc[i] <= index
        cNOP;       CDIV;     // acc[i] <= index/N
        cNOP;       CMULT;    // acc <= N*(index/N) in integers
        cNOP;       STORE(0); // mem[0][i] <= acc[i]
        NOP;        IXLOAD;   // acc[i] <= index
        cVSUB(1);   SUB(0);   // acc <= acc - 1; acc[i] <= index - N*(index/N) = ixModN
        cSTORE(3);  STORE(0); // mem[3] <= size - 1 = cycles; mem[0][i] <= ixModN[i]

                              // mem[1][i] <= sAddr[i] = (ixModN[i] - cycles)modN
        cNOP;       CSUB;     // acc <= ixModN - cycles
        cLOAD(0);   WHERECARRY; // acc <= N; select where carry
        cNOP;       CADD;     // acc[i] <= acc[i] + acc
        cNOP;       ENDWHERE; // reselect all cells
        cNOP;       STORE(1); // store at mem[1][i]

                              // mem[2][i] <= dAddr[i] = (ixModN[i] + cycles)modN
        cLOAD(3);   LOAD(0);  // acc <= cycles; acc[i] <= ixModN[i]
        cLOAD(0);   CADD;     // acc <= N; acc[i] <= ixModN[i] + cycles
        cNOP;       CCOMPARE; // compare with N (acc - N)
        cNOP;       WHERENCARRY;// select where not carry
        cNOP;       CSUB;     // acc[i] <= acc - N;
        cNOP;       ENDWHERE; // reselect all cells
        cNOP;       STORE(2); // store at mem[2][i]

                              // read on "diagonal"
LB(4);  cLOAD(1);   LOAD(1);  // load source; load (ixModN[i] - cycles)modN
        cNOP;       ADDRLD;   // addr[i] <= (ixModN[i] - cycles)modN
        cLOAD(3);   CRLOAD;   // acc[i] <= mem[i][S + (ixModN[i] - cycles)modN]

                              // local, modN rotate with cycles
```

```
        cVSUB(1);   STORE(3);   // save "diagonal" (!!! a register should be good)
LB(2);  cBRNZDEC(2);GLSHIFT;    // global left shift cycle times
        cNOP;       STORE(4);   // save the left shifted "diagonal" (!!! the same note)
                                // write on "diagonal"
        cLOAD(2);   LOAD(2);    // load dest; load (ixModN[i] + cycles)modN
        cNOP;       ADDRLD;     // addr[i] <= (ixModN[i] + cycles)modN
        cNOP;       LOAD(4);    // reload the shifted "diagonal"
        cLOAD(0);   CRSTORE;    // acc[i] <= mem[i][D + (ixModN[i] + cycles)modN]
        cSUB(3);    LOAD(3);    // reload the "diagonal"
        cVSUB(1);   NOP;        //
LB(3);  cBRNZDEC(3);GRSHIFT;    // global right shift N-cycles times
        cLOAD(0);   STORE(4);   // save the right shifted "diagonal"
        cSUB(3);    LOAD(0);    // acc <= cycles; acc[i] <= ixModN[i]
        cNOP;       CCOMPARE;   // compare ixModN[i] with cycles
        cNOP;       WHERENCARRY;// where not carry (select where load the right shift)
        cLOAD(2);   LOAD(4);    // restore the right shifted "diagonal"
        cNOP;       CRSTORE;    // acc[i] <= mem[i][D + (ixModN[i] + cycles)modN]
        cLOAD(0);   ENDWHERE;   // acc <= N; reselect all cells

                                // "increment source diagonal"
        cVSUB(1);   LOAD(1);    // acc <= N-1; load source diagonal addresses
        cNOP;          CSUB;        // acc[i] <= (ixModN[i] + cycles)modN - (N-1)
        cLOAD(0);   WHERENZERO; // select where not carry
        cNOP;       CADD;       // acc[i] <= acc[i] + acc
        cNOP;       ENDWHERE;   // reselect all cells
        cNOP;       STORE(1);   // store back aAddr[i]

                                // "decrement dest diagonal"
        cVSUB(1);   LOAD(2);    // acc <= N-1; acc[i] <= dAddr[i] = (ixModN[i] + cycles)modN
        cNOP;       WHEREZERO;  // select where zero
        cNOP;       CLOAD;      // where 0 acc <= N-1
        cLOAD(5);   ELSEWHERE;  // select where not zero
        cVSUB(1);   VSUB(1);    // acc <= cycles; acc[i] <= acc[i] - 1
        cSTORE(5);  ENDWHERE;   // acc <= acc - 1; reselect all cells
        cBRNZ(4);   STORE(2);   // mem[5] <= cycles; store back dAddr[i]

                                // move diagonal
        cLOAD(1);   LOAD(0);    // acc <= S; acc[i] <= ixModN[i]
        cNOP;       ADDRLD;     // addr[i] <= ixModN[i]
        cLOAD(2);   CRLOAD;     // acc <= D; acc[i] <= mem[i][S + ixModN[i]]
        cNOP;       CRSTORE;    // mem[i][D + ixModN[i]] <= ac[i]
//=============================================================================================
```

7.1.3 The Verification

The execution time is:

$$T_{matrixTranspose} = N^2 + 30N - 7 \in O(N^2)$$

For $N = 1024$, $T_{matrixTranspose} = 1.03 \times N^2$.

Example 7.1 *The matrix is of* 13×13 *elements, it is stored starting with the vector 5, and the result will be stored starting with the vector 25. The matrix contains on each line the index vector.*

```
INITIAL
mem[0]   = 13 // N=13: the size
mem[1]   = 5  // S=5: where starts source matrix
mem[2]   = 25 // D=25: where starts destination matrix

//source matrix
vect[5]  = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[6]  = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[7]  = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[8]  = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[9]  = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[10] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

```
vect[11] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[12] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[13] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[14] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[15] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[16] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[17] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

The program for this example is:

```
/*****************************************************************************
TESTING MATRIX TRANSPOSE
*****************************************************************************/
            cVLOAD(13);     NOP;
            cSTORE(0);      NOP;        // mem[0] = N: matrix size (13)
            cVLOAD(5);      NOP;
            cSTORE(1);      NOP;        // mem[1] = S: source (5)
            cVLOAD(25);     NOP;
            cSTORE(2);      NOP;        // mem[2] = D: destination (25)
            cNOP;           ENDWHERE;   // activate all cells
                                        // SET MATRIX
                                        // v(S)     = 0 1 2 ...
                                        // v(S+1)   = 0 1 2 ...
                                        // ...
            cVLOAD(13);     VLOAD(4);
            cNOP;           ADDRLD;
            cNOP;           IXLOAD;
    LB(1);  cVSUB(1);       VADD(0);
            cBRNZ(1);       RISTORE(1);
            cSTART;         NOP;

    `include "matrixTranspose.v"

            cSTOP;          NOP;        // stop cycles counter
            cHALT;          NOP;        // halt
//============================================================================

FINAL
vect[5]  = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[6]  = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[7]  = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[8]  = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[9]  = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[10] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[11] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[12] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[13] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[14] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[15] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[16] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
vect[17] = 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

vect[25] = 0  0  0  0  0  0  0  0  0  0  0  0  0  13 13 13
vect[26] = 1  1  1  1  1  1  1  1  1  1  1  1  1  14 14 14
```

```
vect[27] = 2  2  2  2  2  2  2  2  2  2  2  2  2  15 15 15
vect[28] = 3  3  3  3  3  3  3  3  3  3  3  3  3  0  0  0
vect[29] = 4  4  4  4  4  4  4  4  4  4  4  4  4  0  0  0
vect[30] = 5  5  5  5  5  5  5  5  5  5  5  5  5  0  0  0
vect[31] = 6  6  6  6  6  6  6  6  6  6  6  6  6  0  0  0
vect[32] = 7  7  7  7  7  7  7  7  7  7  7  7  7  0  0  0
vect[33] = 8  8  8  8  8  8  8  8  8  8  8  8  8  0  0  0
vect[34] = 9  9  9  9  9  9  9  9  9  9  9  9  9  0  0  0
vect[35] = 10 10 10 10 10 10 10 10 10 10 10 10 10 0  0  0
vect[36] = 11 11 11 11 11 11 11 11 11 11 11 11 11 0  0  0
vect[37] = 12 12 12 12 12 12 12 12 12 12 12 12 12 0  0  0
```

Execution time on simulation: 552 cycles.

◇

7.2 Matrix-Vector Multiplication

The algorithm is standard.

7.2.1 The Program

```
/*****************************************************************************
Matrix-Vector Multiplication Algorithm

The 2-line inner loop (labeled 6) performs:
    - load the line: RILOAD(127)
    - multiplication, in map section of the array: MULT(0)
    - reduction add, in reduction section with result in the global shift
        register: cCPUSHL(0)
    - decrement test and decrement the counter: cBRNZDEC(6)
The main loop is repeated N times.
*****************************************************************************/
        cSEND(6);     CADDRLD;    // addr[i] <= mem[6]
        cLOAD(0);     RLOAD(0);   // acc <= N;  load last matrix M1 line
        cVSUB(1);     MULT(0);    // acc <= N-1;add line with vector
                                  // MAIN LOOP
                                  // INNER LOOP
LB(6);  cCPUSHL(1);   RILOAD(127);// push reduction min; load next  line
        cBRNZDEC(6);  MULT(0);    // test end of loop; add line with vector
                                  // END OF INNER LOOP
                                  // latency = 1 + 0.5 log N
        cNOP;         NOP;        // latency
        cNOP;         NOP;        // latency
        cLOAD(9);     SRLOAD;     // acc <= mem[9]; load result in acc
        cVADD(1);     CSTORE;     // acc <= mem[9]+1; mem[i][mem[9]] <= acc[i]
                                  // END OF MAIN LOOP
//===========================================================================
```

The execution time for a $N \times N$ matrix in a system with P cells, where $N \leq P$, is:

$$T_{vm}(N) = 2N + 5 + 0.5 log P \in O(N)$$

where $0.5 log N$ is due to the latency introduced by the reduction network.

7.3 Matrix-Matrix Multiplication

The algorithm is standard.

7.3.1 The Program

The program has the following parameters:

```
        cSTORE(0);      NOP;        // mem[0] <= N
        cVLOAD(24);     NOP;        //
        cSTORE(1);      NOP;        // mem[1] <= M2T
        cVLOAD(32);     NOP;        //
        cSTORE(2);      NOP;        // mem[2] <= R
        cVLOAD(8);      NOP;        //
        cSTORE(3);      NOP;        // mem[3] <= M1
        cVLOAD(16);     NOP;        //
        cSTORE(4);      NOP;        // mem[4] <= M2
```

The program is stored as the file `matrixMatrixMult.v` of form:

```
/*****************************************************************************
MATRIX-MATRIX MULTIPLICATION

R: starting vector for result
M1: starting vector for the multiplicand matrix
M2: starting vector for the multiplier matrix

Main steps:
    - compute starting with M2T the transposed multiplier
    - multiply each line of the multiplicand with the transposed multiplier
*****************************************************************************/
        `include "03_matrixTranspose.v"

                                    // seleect the first N cells only
        cLOAD(0);     IXLOAD;     // acc <= N; acc[i] <= index
        cLOAD(0);     CSUB;       // acc[i] <= index - N
        cSTORE(5);    WHERECARRY; // select only the first N cells
        cLOAD(1);     NOP;        //
        cADD(0);      NOP;        //
        cVSUB(1);     NOP;        //
        cSTORE(6);    NOP;        // mem[6] <= last line in M1
                                  // M1 "x" M2T
LB(7);  cLOAD(3);     NOP;        // acc = pointer in M1
        cVADD(1);     CALOAD;     // increment pointer; acc[i] <= mem[i][mem[3]]
        cSTORE(3);    STORE(0);   // save the pointer; load line at 0

        `include "03_matrixVectMult.v"

        cSTORE(2);    NOP;        //
        cLOAD(5);     NOP;        // acc = loopCounter
        cVSUB(1);     NOP;        // decrement loopCounter
        cSTORE(5);    NOP;        // store back loopCounter
        cBRNZ(7);     NOP;
                                    // END MATRIX-MATRIX MULTIPLICATION
//============================================================================
```

The execution time, for $N \leq P$, is $T_{matrixMatrixMult} = 3N^2 + 0.5Nlog_2P + 43N \in O(N^2)$.
For $N = 1024$ results: $T_{matrixMatrixMult} = 3.046N^2$, out of which $3N^2$ are consumed in the following lines as follows:

- from the file `03_matrixTranspose.v` the following two lines are executed in N cycles

```
LB(2);  cBRNZDEC(2);GLSHIFT;    // global left shift cycle times
...
LB(3);  cBRNZDEC(3);GRSHIFT;    // global right shift N-cycles times
```

- from the file `03_matrixVectMult.v.v` the following two lines are executed in 2N cycles

```
LB(2); cCPUSHL(0);  RILOAD(63); // push reduction sum; load matrix line
       cBRNZDEC(2); MULT(0);    // test end of loop; line-vector multiply
```

7.3.2 The Verification

Example 7.2 *Let us multiply the two matrix, M1 and M2, with $N = 7$, stored in vector memory starting with* `vectMem[8]` *for M1, and* `vectMem[16]` *for M2. The result will be stored starting with* `vectMem[32]`. *The memory space starting from* `vectMem[24]` *is used to store the transposed from of the matrix M2.*

The matrix M1 contains on the line `i` *the vector* `index + i`, *while M2 contains on each line the vector $\langle i, i, \ldots, i \rangle$, for $i = 1, \ldots, 7$.*

```
/*************************************************************************
TESTING MATRIX-MATRIX MULTIPLICATION
*************************************************************************/
        cVLOAD(7);      ENDWHERE;   // activate all cells
        cSTORE(0);      NOP;        // mem[0] <= N
        cVLOAD(24);     NOP;        //
        cSTORE(1);      NOP;        // mem[1] <= M2T
        cVLOAD(32);     NOP;        //
        cSTORE(2);      NOP;        // mem[2] <= R
        cVLOAD(8);      NOP;        //
        cSTORE(3);      NOP;        // mem[3] <= M1
        cVLOAD(16);     NOP;        //
        cSTORE(4);      NOP;        // mem[4] <= M2
                                    // mem[5] reserved for cycles
                                    // SET MATRIX M1
                                    // v8  = index + 1
                                    // v9  = index + 2
                                    // ...
                                    // v14 = index + 7
        cLOAD(3);       NOP;
        cVSUB(1);       NOP;
        cLOAD(0);       CADDRLD;
        cNOP;           IXLOAD; // VLOAD(3);
LB(8);  cVSUB(1);       VADD(1);
        cBRNZ(8);       RISTORE(1);

                                    // SET MATRIX M2
                                    // v16 = 0 0 ... 0
                                    // v17 = 1 1 ... 1
                                    // ...
                                    // v22 = 6 6 ... 6
```

```
        cLOAD(4);       NOP;
        cVSUB(1);       NOP;
        cLOAD(0);       CADDRLD;
        cNOP;           VLOAD(255);
LB(9);  cVSUB(1);       VADD(1);
        cBRNZ(9);       RISTORE(1);

        cSTART;         NOP;        // start cycle counter

        `include "03_matrixMatrixMult.v"

        cSTOP;          NOP;        // stop cycle counter
        cHALT;          NOP;
//============================================================================
```

The initial state of the vector memory is:

```
vect[8]      = 1 2 3 4  5  6  7  8  9  10 11 12 13 14 15 16
vect[9]      = 2 3 4 5  6  7  8  9  10 11 12 13 14 15 16 17
vect[10]     = 3 4 5 6  7  8  9  10 11 12 13 14 15 16 17 18
vect[11]     = 4 5 6 7  8  9  10 11 12 13 14 15 16 17 18 19
vect[12]     = 5 6 7 8  9  10 11 12 13 14 15 16 17 18 19 20
vect[13]     = 6 7 8 9  10 11 12 13 14 15 16 17 18 19 20 21
vect[14]     = 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
vect[15]     = x x x x  x  x  x  x  x  x  x  x  x  x  x  x
vect[16]     = 0 0 0 0  0  0  0  0  0  0  0  0  0  0  0  0
vect[17]     = 1 1 1 1  1  1  1  1  1  1  1  1  1  1  1  1
vect[18]     = 2 2 2 2  2  2  2  2  2  2  2  2  2  2  2  2
vect[19]     = 3 3 3 3  3  3  3  3  3  3  3  3  3  3  3  3
vect[20]     = 4 4 4 4  4  4  4  4  4  4  4  4  4  4  4  4
vect[21]     = 5 5 5 5  5  5  5  5  5  5  5  5  5  5  5  5
vect[22]     = 6 6 6 6  6  6  6  6  6  6  6  6  6  6  6  6
```

The final sate of the vector memory is:

```
vect[0]      = 0   1   2   3   4   5   6   0   1   2   3   4   5   6   0   1
vect[1]      = 0   1   2   3   4   5   6   0   1   2   3   4   5   6   0   1
vect[2]      = 0   1   2   3   4   5   6   0   1   2   3   4   5   6   0   1
vect[3]      = 6   0   1   2   3   4   5   6   0   1   2   3   4   5   6   0
vect[4]      = 0   0   0   0   0   0   6   0   1   2   3   4   5   6   0   1
vect[5]      = x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
vect[6]      = x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
vect[7]      = x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
vect[8]      = 1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
vect[9]      = 2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17
vect[10]     = 3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18
vect[11]     = 4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19
vect[12]     = 5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
vect[13]     = 6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21
vect[14]     = 7   8   9   10  11 12   3   14  15  16  17  18  19  20  21  22
vect[15]     = x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
vect[16]     = 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
vect[17]     = 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
vect[18]     = 2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
vect[19]     = 3   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3
```

```
vect[20]     = 4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4
vect[21]     = 5   5   5   5   5   5   5   5   5   5   5   5   5   5   5   5
vect[22]     = 6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6
vect[23]     = x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
vect[24]     = 0   1   2   3   4   5   6   0   1   2   3   4   5   6   0   1
vect[25]     = 0   1   2   3   4   5   6   0   1   2   3   4   5   6   0   1
vect[26]     = 0   1   2   3   4   5   6   0   1   2   3   4   5   6   0   0
vect[27]     = 0   1   2   3   4   5   6   0   1   2   3   4   5   6   0   0
vect[28]     = 0   1   2   3   4   5   6   0   1   2   3   4   5   6   0   0
vect[29]     = 0   1   2   3   4   5   6   0   1   2   3   4   5   6   0   0
vect[30]     = 0   1   2   3   4   5   6   0   1   2   3   4   5   6   0   0
vect[31]     = x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
vect[32]     = 112 112 112 112 112 112 112 x   x   x   x   x   x   x   x   x
vect[33]     = 133 133 133 133 133 133 133 x   x   x   x   x   x   x   x   x
vect[34]     = 154 154 154 154 154 154 154 x   x   x   x   x   x   x   x   x
vect[35]     = 175 175 175 175 175 175 175 x   x   x   x   x   x   x   x   x
vect[36]     = 196 196 196 196 196 196 196 x   x   x   x   x   x   x   x   x
vect[37]     = 217 217 217 217 217 217 217 x   x   x   x   x   x   x   x   x
vect[38]     = 238 238 238 238 238 238 238 x   x   x   x   x   x   x   x   x
```

The cycles counter provides: `cc = 462`. *Indeed, for* $N = 7$ *the theoretical evaluation provides the same results:* $T_{vectMatrixMult} = 462$.

◇

7.4 Matrix Move

The program moves a matrix form a source, S, specified by the location of the first line, to the destination, D, specified by the location of the first line. The code is:

```
/*****************************************************************************
TESTING MATRIX MOVE
*****************************************************************************/
            cLOAD(0);       NOP;
    LB(10); cSTORE(5);      NOP;
            cLOAD(6);       NOP;
            cVADD(1);       CALOAD;
            cSTORE(6);      NOP;
            cLOAD(7);       NOP;
            cVADD(1);       CSTORE;
            cSTORE(7);      NOP;

            cLOAD(5);       NOP;
            cVSUB(1);       NOP;
            cBRNZ(10);      NOP;
//===========================================================================
```

The following code is used to define the source and the destination of the move operation

```
//===========================================================================
            cVLOAD(38);     NOP;        //
            cSTORE(6);      NOP;        // mem[6] <= S (source)
            cVLOAD(8);      NOP;        //
            cSTORE(7);      NOP;        // mem[7] <= D (destination)
//===========================================================================
```

Chapter 8

The Second Dwarf: Sparse Linear Algebra

Part IV

Machine Learning

Chapter 9

What is Machine Learning

Chapter 10

Clustering

Clustering is a unsupervised learning technique. It deals with finding a structure in a collection of data by organizing objects into groups whose members are similar in some way. A cluster is a collection of objects which are *similar* between them and are *dissimilar* to the objects belonging to other clusters. The main problem is to define the meaning of "similar" and "dissimilar".

Types of clustering:

- distance-based clustering: the objects belong to the same cluster if they are *close* according to a given (usually geometrical distance
- conceptual clustering: objects are grouped according to their fit to descriptive concepts

The main applications are:

- WWW: document classification by clustering web data to emphasize groups of similar patterns.
- Insurance: identifying groups of insurance policy holders according to the claim cost
- Libraries: book ordering according to their content
- Marketing: identifying groups of customers with similar characteristics using the database of customer containing their properties and past records

The main clustering algorithms are:

- K-means
- Hierarchical clustering
- Fuzzy C-means
- Mixture of Gaussians

10.1 K-means

K-means [MacQueen '67] is an unsupervised learning algorithms that solve the clustering problem.

10.1.1 Distance-based k-means clustering

Given n d-dimension vectorial entities (points)

$$\langle \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n} \rangle$$

k-means clustering provides the partition into k sets. The generic algorithm consists of the following main steps:

```
/*************************************************************************
K–MEANS DISTANCE–BASED ALGORITHM
*************************************************************************/
  (1) set (randomly) k d–dimension centers and
      allocate each point (randomly) to a center
  (2) compute for each d–dimension point the Euclidean distance to the
      k centers and assign each point to the ‘‘nearest” center
  (3) compare the new assignments to the old ones
        if (no difference), then stop the process
          else continue
  (4) move the k centers to the means of created groups, and go to (2).
```

The algorithm is sensitive to the initial positions of the k center. It is recommended to place them as much as possible far away from each other.

Implementation

Let us consider the points stored in HOST's data memory as N sequences of d-dimension vectors. The result will be a N-dimension vector of scalars indicating the index of the center associated to each of the N points considered. To explain how the architectural features of our accelerator are used to solve this problem, we take a constant sized example.

Example 10.1 *If the size of array is $p = 1024$, and $d = 3$, we consider $N = 1023$. The algorithm is:*

```
/*************************************************************************
K–MEANS IMPLEMENTATION ON ACCELERATOR
*************************************************************************/
  (1) load from the HOST's data memory 3 1023–component vectors
  (2) instantiate the k centers in CONTROLLER's data memory
  (3) transpose the 341 3x3 matrix stored in the 3 vectors just loaded
      (each of the first 1023 cells will contain the coordinate of a point
  (4) instantiate a 4th vector with (index)mod k (the initial solution)
  (5) associate each point to the nearest center
  (6) if (solution is identical with the previous solution)
        then store the solution at HOST'S memory and stop the computation
  (7) compute the position of the k centers and go to (5)
```

The initial data in HOST's memory is a stream of triplets, as follows:

$$\begin{bmatrix} x_0 & y_0 & z_0 & x_1 & y_1 & z_1 & \ldots & x_{1023} & y_{1023} & z_{1023} \end{bmatrix}$$

The external data is loaded in three vectors in ARRAY, as follows:

$$\begin{bmatrix} x_0 & y_0 & z_0 & x_1 & y_1 & z_1 & \ldots & x_{340} & y_{340} & z_{340} \\ x_{341} & y_{341} & z_{341} & x_{342} & y_{342} & z_{342} & \ldots & x_{681} & y_{681} & z_{681} \\ x_{682} & y_{682} & z_{682} & x_{683} & y_{683} & z_{683} & \ldots & x_{1023} & y_{1023} & z_{1023} \end{bmatrix}$$

Once loaded, the three 1023-component vectors are considered as 341 3× matrices. The 3× matrices are transposed. Results in ARRAY the following 1023 3-component vertical vectors, on for each point. One vector, X, with the x coordinates, another, Y, for the y coordinates and a third,Z, for the z coordinates.

$$\begin{bmatrix} x_0 & x_{341} & x_{682} & x_1 & x_{342} & x_{683} & \dots & x_{340} & x_{681} & x_{1023} \\ y_0 & y_{341} & y_{682} & y_1 & y_{342} & y_{683} & \dots & y_{340} & y_{681} & y_{1023} \\ z_0 & y_{341} & z_{682} & z_1 & z_{342} & z_{683} & \dots & z_{340} & z_{681} & z_{1023} \end{bmatrix}$$

The final result is the vector K, containing the index of the cluster to which each point belongs.

$$\begin{bmatrix} k_0 & k_{341} & k_{682} & k_1 & k_{342} & k_{683} & \dots & k_{340} & k_{681} & k_{1023} \end{bmatrix}$$

The main open problem, for the time being, is how to send back to the external memory the result, because the vector K must be somehow reordered in order to be friendly used.

◇

Evaluation

The degree of parallelism for the steps 5 and 6 on the loop of the algorithm is maximal. Only the step 7 is executed with a degree of parallelism p/k.

Let us consider initially a number of points equal with p. Then, each point is associated to a cell, which stores the d coordinates. The computation is not I/O bounded even for the smallest data bandwidth of 4GB/sec, if $30k > p$. In these easy to fulfil conditions, by simulation, the architectural acceleration of a pure sequential computation results, for $k > 10$:

$$A \simeq p \times \frac{\alpha}{1+\alpha}$$

where:

$$\alpha = \frac{execution\_time\_for\_steps\ 5+6}{execution\_time\_for\_step\ 7} \simeq \frac{16}{4+log_2 p}$$

For $p = 1024$, $A \simeq 546$.

The number of points can be easy expanded, maintaining the acceleration, to hundreds of thousands if the computation remains not I/O bounded. The data for each set of p points is stored in $d+1$ horizontal vectors.

10.1.2 Conceptual k-means clustering

While for the distance-based k-means clustering the vector

$$o_j = \langle v_1, \dots, v_m \rangle$$

consists of numerical values, for conceptual k-means clustering it consists of bits representing the presence, for $b_i = 1$, or the absence, for $b_i = 0$, of the feature i associated to the object j:

$$x_i = \langle b_1, \dots, b_m \rangle$$

Instead of numerical evaluation of the "distance", now associative mechanisms must be used to evaluate the "distance" of each point from each center. A numerical evaluation is substituted with a fitting mechanism. The problem is:

Given :

- a set of abstract objects: $X = \{x_1, \dots, x_n\}$
- a set of attribute associated to the objects: $x_i = \langle b_{i1}, \dots, b_{im} \rangle$ for $i = 1, \dots, n$, i.e., each x_i is a m-bit binary number
- a body of knowledge for evaluating the belonging to a class

Find :
a hierarchy of objects in classes

One solution is a hierarchical approach which uses a recursive bi-partitioning clustering algorithm. The bi-partitioning clustering algorithm has the following stages:

1. define the similarity measure, s_{ij}, for the n m-bit binary variables, x_i and x_j, as the *Hamming distance* between them
2. compute the normalized information distance of each object from each other objects as a $n \times n$ similarity matrix $\mathbf{S}$, of form:
$$\mathbf{S} = \begin{bmatrix} s_{11} & \dots & s_{1n} \\ \vdots & \ddots & \vdots \\ s_{n1} & \dots & s_{nn} \end{bmatrix}$$
by starting from the vector $[x_1, \dots, x_n]$ with the computation of the symmetric matrix
$$\mathbf{XOR} = \begin{bmatrix} x_1 \oplus x_1 & x_1 \oplus x_2 & \dots & x_1 \oplus x_n \\ x_2 \oplus x_1 & x_2 \oplus x_2 & \dots & x_2 \oplus x_n \\ \vdots & \vdots & \ddots & \vdots \\ x_n \oplus x_1 & x_n \oplus x_2 & \dots & x_n \oplus x_n \end{bmatrix}$$
where $x_i \oplus x_j$ is the bitwise exclusive or logic function (the number of 1s in $x_i \oplus x_j$ represents the degree of similarity between x_i and x_j), and ending with the computing $s_{ij} = \sum_{k=1}^{m} b_{ik} \oplus b_{jk}$, the components of the similarity matrix $\mathbf{S}$.
3. apply a spectral clustering algorithm on $\mathbf{S}$, considered as the representation of a graph having in its vertexes objects and the edges marked with the distance between the objects it connects, as follows:
   (a) build the Laplacian matrix of $\mathbf{S}$,
   $$\mathbf{L} = \mathbf{D} - \mathbf{S}$$
   where $\mathbf{D}$ is the *degree matrix* of $\mathbf{S}$ defined as the diagonal matrix with $d_{ij} = \sum_{i=1}^{n} s_{ij}$
   (b) compute the dominant eigenvalue and the associated eigenvector[1]. The eigenvalue and eigenvector are defined by the following equation:
   $$\mathbf{Av} = \lambda \mathbf{v}$$
   where $\mathbf{A}$ is an $n \times n$ matrix, $\mathbf{v}$ is a non-zero n-component vector and λ is a scalar (real or complex). Any value of λ for which this equation has a solution is known as an ***eigenvalue*** of the matrix $\mathbf{A}$. The vector $\mathbf{v}$ which corresponds to this value is the associated ***eigenvector***. The above equation is transformed as follows:
   $$\mathbf{Av} - \lambda \mathbf{v} = 0$$
   $$\mathbf{Av} - \lambda \mathbf{Iv} = 0$$
   $$(\mathbf{Av} - \lambda \mathbf{I})\mathbf{v} = 0$$
   and, if $\mathbf{v}$ is a non-zero vector, then the equation has a solution if
   $$|\mathbf{Av} - \lambda \mathbf{I}| = 0$$
   Computing the determinant results an n-th order polynomial in λ. The n roots are computed only by numerical methods for big n, the usual case.
   Because, in the most of cases n is a big value, we have to use a numerical method as follows [http:eigenValuesVectors]:

[1] "Moreover, in our examples with K clusters so far, always the D = K dominant eigenvectors have been sufficient." [Fischer '04]

i. take an arbitrary vector $\mathbf{X}^{(0)}$

ii. compute its normalized form:

$$\mathbf{Y}^{(i)} = \frac{1}{max(\mathbf{X}^{(i)})}\mathbf{X}^{(i)}$$

iii. iterate the value of $\mathbf{X}^{(i)}$:

$$\mathbf{X}^{(i+1)} = \mathbf{Y}^{(i)}\mathbf{L}$$

iv. test for ending by computing:

$$\Delta(\mathbf{X}^{(i)}) = \mathbf{X}^{(i+1)} - \mathbf{X}^{(i)}$$

and:

A. **if** ($\Delta(\mathbf{X}^{(i)})$ is small enough) **then** stop the iterative process and return $\mathbf{X}^{(i+1)}$ as the dominant eigen vector

B. **else** continue going back to the step `ii`.

(c) use the dominant eigenvector to make the bi-partition.

Example 10.2 *Let's consider a simple example (following* [Hamad '08]*) of 6 objects:* $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ *defined by the the graph represented in Figure 10.1, where the edges are marked by the normalized similarity measure between* x_i *and* x_j*. Where the edge is missing the similarity measure is 0. The similarity matrix* **S** *is:*

$$\mathbf{S} = \begin{bmatrix} 0 & 0.8 & 0.6 & 0 & 0.1 & 0 \\ 0.8 & 0 & 0.8 & 0 & 0 & 0 \\ 0.6 & 0.8 & 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0.2 & 0 & 0.8 & 0.7 \\ 0.1 & 0 & 0 & 0.8 & 0 & 0.8 \\ 0 & 0 & 0 & 0.7 & 0.8 & 0 \end{bmatrix}$$

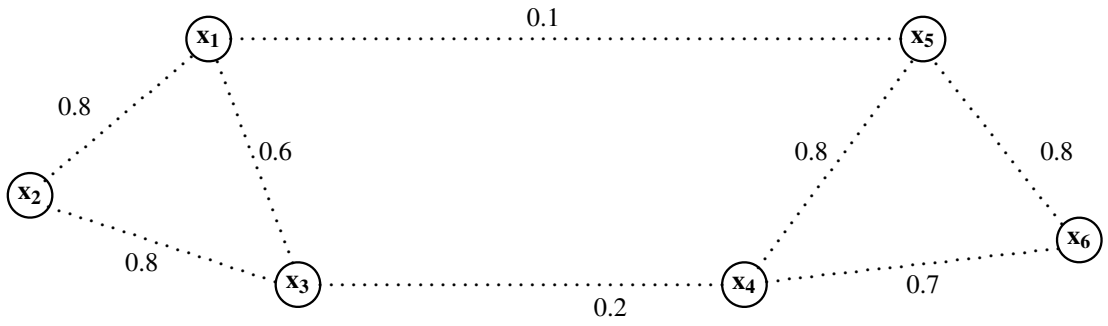

Figure 10.1:

The Laplacian matrix is build as using the degree matrix, **D***, as follows:*

$$\mathbf{L} = \mathbf{D} - \mathbf{S} = \begin{bmatrix} 1.5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.5 \end{bmatrix} - \begin{bmatrix} 0 & 0.8 & 0.6 & 0 & 0.1 & 0 \\ 0.8 & 0 & 0.8 & 0 & 0 & 0 \\ 0.6 & 0.8 & 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0.2 & 0 & 0.8 & 0.7 \\ 0.1 & 0 & 0 & 0.8 & 0 & 0.8 \\ 0 & 0 & 0 & 0.7 & 0.8 & 0 \end{bmatrix}$$

$$\mathbf{L} = \begin{bmatrix} 1.5 & -0.8 & -0.6 & 0 & -0.1 & 0 \\ -0.8 & 1.6 & -0.8 & 0 & 0 & 0 \\ -0.6 & -0.8 & 1.6 & -0.2 & 0 & 0 \\ 0 & 0 & -0.2 & 1.7 & -0.8 & -0.7 \\ -0.1 & 0 & 0 & -0.8 & 1.7 & -0.8 \\ 0 & 0 & 0 & -0.7 & -0.8 & 1.5 \end{bmatrix}$$

The dominant eigenvalue, λ, and the associated eigenvector, $\mathbf{X}$, are computed iteratively starting with an arbitrary vector $\mathbf{X}^{(0)} = [1\ 2\ 1\ 2\ 1\ 2\ 1\ 2]^T$ by computing

$$\mathbf{X}^{(1)} = \mathbf{L}\mathbf{X}^{(0)} = \begin{bmatrix} 1.5 & -0.8 & -0.6 & 0 & -0.1 & 0 \\ -0.8 & 1.6 & -0.8 & 0 & 0 & 0 \\ -0.6 & -0.8 & 1.6 & -0.2 & 0 & 0 \\ 0 & 0 & -0.2 & 1.7 & -0.8 & -0.7 \\ -0.1 & 0 & 0 & -0.8 & 1.7 & -0.8 \\ 0 & 0 & 0 & -0.7 & -0.8 & 1.5 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} -0.8 \\ 1.6 \\ -1 \\ 1 \\ -1.6 \\ 0.8 \end{bmatrix}$$

and then normalizing the result (to obtain a column vector with the biggest element of value 1)

$$\mathbf{Y}^{(1)} = \frac{1}{max(\mathbf{X}^{(1)})} = \frac{1}{1.6} \begin{bmatrix} -0.8 \\ 1.6 \\ -1 \\ 1 \\ -1.6 \\ 0.8 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 1 \\ -0.625 \\ 0.625 \\ -1 \\ 0.5 \end{bmatrix}$$

where $max(\mathbf{X}^{(1)})$ is the maximum component of $\mathbf{X}^{(1)}$. The process continue until the difference between $\mathbf{Y}^{(i-1)}$ and $\mathbf{Y}^{(i)}$ is small enough. Then, $\mathbf{X}^{(i)}$ is the eigen vector and the value used to normalize it is the eigen value.

For this example results the dominant eigen vector $[0.2\ \ 0.2\ \ 0.2\ \ -0.4\ \ -0.7\ \ -0.7]^T$. The position where the sign chances delimits the partition. Therefore, the partition is $\{\{x_1,x_2,x_3\},\{x_4,x_5,x_6\}\}$

$\diamond$

10.2 Hierarchical clustering

10.3 Fuzzy C-means

10.4 Mixture of Gaussians

Chapter 11

Regression

Regression is an important algorithm used in machine learning [Theobald '91]. It provides the base for other learning algorithms, such as the neural networks algorithm.

Regression is used in data mining, finance, business and investing. It is applied to determine the strength of a relationship between one *dependent variable* (typically represented at y) and other *independent variables*, (typically represented at $x_1, ...$).

11.1 Linear Regression

Linear regression uses one independent variable, x, to predict the outcome of the dependent variable, y, and is expressed through a straight regression line.

$$y = a + bx$$

The input for computation is a vector of pairs of coordinates, as follows:

$$\langle (x_1, y_1), (x_2, y_2), \ldots (x_n, y_n) \rangle$$

and the output is computed using the following expressions:

$$a = \frac{\Big(\sum_{i=1}^{n} y_i\Big)\Big(\sum_{i=1}^{n} x_i^2\Big) - \Big(\sum_{i=1}^{n} x_i\Big)\Big(\sum_{i=1}^{n} x_i y_i\Big)}{n\Big(\sum_{i=1}^{n} x_i^2\Big) - \Big(\sum_{i=1}^{n} x_i\Big)^2}$$

$$b = \frac{n\Big(\sum_{i=1}^{n} x_i y_i\Big) - \Big(\sum_{i=1}^{n} x_i\Big)\Big(\sum_{i=1}^{n} y_i\Big)}{n\Big(\sum_{i=1}^{n} x_i^2\Big) - \Big(\sum_{i=1}^{n} x_i\Big)^2}$$

Our hybrid system will be used to compute the sums: $\sum_{i=1}^{n} x_i$, $\sum_{i=1}^{n} y_i$, $\sum_{i=1}^{n} x_i^2$, and $\sum_{i=1}^{n} x_i y_i$, organized in a 4-component vector which is send back to the HOST's data memory. Then, the four values are used by HOST to compute a and b. Squaring the values of x_i and multiplying x_i with y_i, for $i = 1, \ldots, n$ is performed in parallel, while the sums are performed by the REDUCTION unit of ACCELERATOR. All the other operations, loads and stores and the final computation are performed sequentially.

For very big n the transfer operations can be at least partially overlapped with the computation, but the computing time remains IO bounded.

Example 11.1 *In Figure 11.1 there are 16 points in a two-dimension space.*

$$\langle (4,3),(1,5),(5,5),(7,7),(10,7),(2,8),(5,10),(12,10),$$
$$(3,12),(8,13),(12,13),(10,14),(15,14),(13,16),(10,17),(16,18)\rangle$$

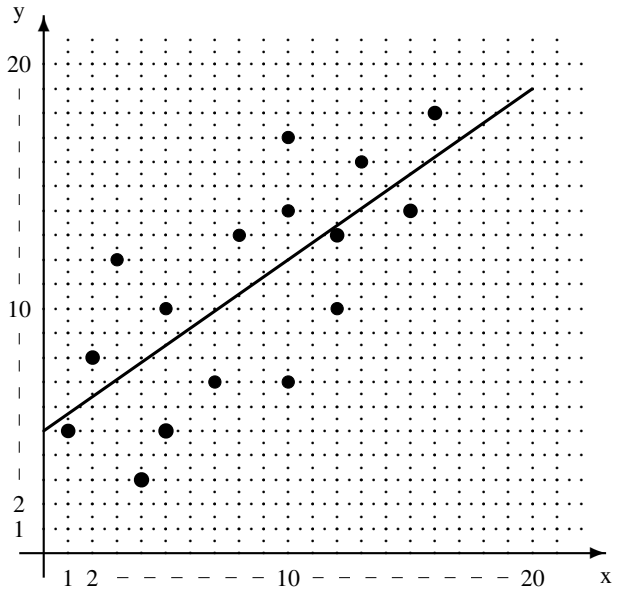

Figure 11.1:

```
/*****************************************************************************
REGRESSION ALGORITHM
    initial: 16 pairs of coordinated are stored in HOST's data memory
             starting form the address 96.
    final: rf[5] = 10a, rf[6] = 10b (to avoid floating point operations
*****************************************************************************/
  (1) Load the pairs of coordinates as two vectors in ARRAY at
        vectMem[33]
        vectMem[34]
  (2) transpose the 2x2 matrices stored in the two vectors: thus each
        cell contains a pais of coordinates
  (3) push left in serialReg redSum(vectMem[33])
  (4) push left in serialReg redSum(vectMem[33] x vectMem[33])
  (5) push left in serialReg redSum(vectMem[34])
  (6) push left in serialReg redSum(vectMem[33] x vectMem[34])
  (7) send serialReg to HOST in:
        mem[0] <= SUM(xy)
        mem[1] <= SUM(y)
        mem[2] <= SUM(x^2)
        mem{3] <= SUM(x)
  (8) compute in HOST the parameters of the line:
```

```
        rf[5] <= 10a
        rf[6] <= 10b
```

```
/*****************************************************************************
REGRESSION ALGORITHM: the program running on HOST
File name: 05_hostRegression.v
*****************************************************************************/
    hSTART;
    RUN(`MLOAD, 33, 96, 16, 1);      // load first vector with n/2 pairs of coordinates
    RUN(`MLOAD, 34, 112, 16, 1);     // load second vector with n/2 pairs of coordinates
    hRUN(`REG);                      // call the function REG running on ACCELERATOR
    RUN(`MSTORE, 37, 90, 4, 1);      // store the 4-scalar vector in HOST's data memory
    hRUN(`EOP);                      // call END OF PROGRAM to know when MSTORE ends
    hWAITACC;                        // wait for end of program on ACCELERATOR
    hIGET(90);
    hLOAD(0);                        // rf[0] <= SUM(xy)
    hIGET(91);
    hLOAD(1);                        // rf[1] <= SUM(y)
    hIGET(92);
    hLOAD(2);                        // rf[2] <= SUM(x^2)
    hIGET(93);
    hLOAD(3);                        // rf[3] <= SUM(x)
    hVMULT(4, 2, 16);
    hMULT(5, 3, 3);
    hSUB(4, 4, 5);
    hVDIV(4, 4, 10);                 // to increase the precision
    hMULT(5, 3, 0);
    hMULT(6, 1, 2);
    hSUB(5, 6, 5);
    hDIV(5, 5, 4);                   // rf[5] <= 10a
    hVMULT(6, 0, 16);
    hMULT(7, 1, 3);
    hSUB(6, 6, 7);
    hDIV(6, 6, 4);                   // rf[6] <= 10b
    hSTOP;
    hHALT;
```

The program written for ACCELERATOR is:

```
/*****************************************************************************
REGRESSION ALGORITHM: the program running on ACCELERATOR
File name: 05_regression.v
*****************************************************************************/
                    `define MLOAD   2
                    `define ML      6
                    `define REG     8
                    `define MSTORE  1
                    `define MS1     3
                    `define MS2     5
                    `define EOP     4
```

```
LB('MLOAD);     cPOPFIFO;       NOP;
                cNOP;           CLOAD;
                cPOPFIFO;       VSUB(1);
                cSTORE(2);      ADDRLD;
                cPOPFIFO;       NOP;
                cSTORE(1);      NOP;
                cPOPFIFO;       NOP;
                cSTORE(3);      NOP;

LB('ML);        cLADDR(2);      NOP;
                cLSIZE(1);      NOP;
                cTRUN(1);       NOP;
                cLOAD(2);       NOP;
                cADD(1);        NOP;
                cSTORE(2);      NOP;
                cIOWAIT;        NOP;
                cLOAD(3);       NOP;
                cVSUB(1);       IOLOAD;
                cSTORE(3);      RISTORE(1);
                cBRNZ('ML);     NOP;
                cHALT;          NOP;

LB('MSTORE);    cPOPFIFO;       NOP;
                cNOP;           CADDRLD;
                cPOPFIFO;       RILOAD(0);
                cSTORE(2);      NOP;
                cPOPFIFO;       NOP;
                cSTORE(1);      NOP;
                cPOPFIFO;       IOSTORE;
                cSTORE(3);      NOP;

LB('MS1);       cLADDR(2);      NOP;
                cLSIZE(1);      NOP;        load io register with acc
                cTRUN(2);       NOP;        ration in DMA
                cLOAD(3);       NOP;
                cVSUB(1);       NOP;
                cBRZ('MS2);     NOP;
                cSTORE(3);      NOP;
                cLOAD(2);       RILOAD(1);
                cADD(1);        NOP;
                cSTORE(2);      NOP;
                cIOWAIT;        NOP;
                cJMP('MS1);     IOSTORE;
LB('MS2);       cIOWAIT;        NOP;
                cHALT;          NOP;


LB('EOP);       cTRUN(7);       NOP;
                cHALT;          NOP;
                                            // REGRESSION PROGRAM
LB('REG);       cSTART;         LOAD(33);   // TRANSPOSE
                cNOP;           GLSHIFT;
                cNOP;           STORE(35);
```

```
cNOP;              LOAD(34);
cNOP;              GRSHIFT;
cNOP;              STORE(36);
cNOP;              IXLOAD;
cNOP;              VAND(1);
cNOP;              WHEREZERO;
cNOP;              LOAD(35);
cNOP;              STORE(34);
cNOP;              ELSEWHERE;
cNOP;              LOAD(36);
cNOP;              STORE(33);
cNOP;              ENDWHERE;
cNOP;              LOAD(33);     // COMPUTE SUMS
cCPUSHL(0);        MULT(33);
cCPUSHL(0);        LOAD(34);
cCPUSHL(0);        MULT(33);
cCPUSHL(0);        NOP;
cNOP;              NOP;
cNOP;              NOP;
cNOP;              NOP;
cNOP;              SRLOAD;
cSTOP;             STORE(37);
cHALT;             NOP;
```

The result of the program is stored in the register file of HOST:

$$a = 4.9$$

$$b = 0.7$$

Using them, the line from Figure 11.1 is drown.

◇

11.2 Non-Linear Regression

Non-linear regression is similar to linear regression in that it seeks to track a particular response from a set of variables on the graph. However, non-linear models are somewhat more complicated to develop. Non-linear models are created through a series of iterations. The Gauss-Newton method is the most used non-linear regression modelling techniques. It receive as input a set of n points in a two-dimension space:

$$\mathbf{p} = \{(x_i, y_i) \mid i = 1, \ldots, n\}$$

and a function f which is supposed to approximate the evolution described by $\mathbf{x}$ in the two-dimension space:

$$y = f(x, a_1, \ldots, a_m)$$

where the vector $\mathbf{a} = [a_1, \ldots, a_m]$ contains the parameters whose values will be approximated using the Gauss-Newton method. The problem is to find the actual values in $\mathbf{a}$ for which

$$\varepsilon(a_1, \ldots, a_m) = \sum_{i=1}^{n} (y_i - f(x_i, a_1, \ldots, a_m))^2 = \sum_{i=1}^{n} (y_i - f_i(a_1, \ldots, a_m))^2 = \sum_{i=1}^{n} r_i^2$$

with $r_i = y_i - f_i(a_1, \ldots, a_m)$, is minimal. Therefore, the following equations must be solved:

$$\frac{\delta}{\delta a_j} = \varepsilon(a_1, \ldots, a_m) = -2 \sum_{i=1}^{n} (y_i - f(x_i, a_1, \ldots, a_m)) \frac{\delta f_i(a_1, \ldots, a_m)}{\delta a_j} = 0$$

for $j = 1, \ldots, m$, where

$$J_{ij} = \frac{\delta f_i(a_1, \ldots, a_m)}{\delta a_j} \; for \; (i = 1, \ldots, n); j = 1, \ldots, m)$$

is an element of the Jacobian matrix $\mathbf{J_f}$.

$$\mathbf{J_f} = \begin{bmatrix} J_{11} & \ldots & J_{1m} \\ \vdots & \ddots & \vdots \\ J_{n1} & \ldots & J_{nm} \end{bmatrix} = \begin{bmatrix} \frac{\delta f_1(a_1,\ldots,a_m)}{\delta a_1} & \ldots & \frac{\delta f_1(a_1,\ldots,a_m)}{\delta a_m} \\ \vdots & \ddots & \vdots \\ \frac{\delta f_n(a_1,\ldots,a_m)}{\delta a_1} & \ldots & \frac{\delta f_n(a_1,\ldots,a_m)}{\delta a_m} \end{bmatrix}$$

The solution of the problem (see [http:GaussNewtonMethod]) is iterative, in each step, starting from an initial "inspired guess" $\mathbf{a}^{(0)}$, is computed

$$\mathbf{a}^{(t+1)} = \mathbf{a}^{(t)} + \Delta$$

where:

$$\Delta = (\mathbf{J_f}^T \mathbf{J_f})^{-1} \mathbf{J_f}^T \mathbf{r}$$

with $\mathbf{r} = [r_1, \ldots, r_n]^T$. The iterative process stops when the components of Δ become small enough. Some times, problems of ill-conditioning and divergence can occur. They can be corrected by finding initial parameter estimates that are near to the optimal values.

The algorithm for the hybrid system has the following steps:

1. initialization:
   - load the vectors $\mathbf{x} = [x_1, \ldots, x_n]$ and $\mathbf{y} = [y_1, \ldots, y_n]$ in ARRAY
   - load the vector $\mathbf{a}^0 = [a_1^0, \ldots, a_m^0]$ in CONTROLLER's data memory
2. compute in ARRAY: the matrix $\mathbf{J}_f^T$ as m n-component vectors
3. compute in ARRAY $\mathbf{J_f}^T \mathbf{J_f}$ as a $m \times m$ matrix
4. compute the inverse of $\mathbf{J_f}^T \mathbf{J_f}$ by:
   - sending the $m \times m$ matrix to HOST
   - computing in HOST the inverse
   - sending back to CONTROLLER the $m \times m$ matrix $(\mathbf{J_f}^T \mathbf{J_f})^{-1}$
5. compute in ARRAY $(\mathbf{J_f}^T \mathbf{J_f})^{-1} \mathbf{J_f}^T$ as matrix $m \times n$
6. compute in ARRAY the n-component vector $\mathbf{r} = [(y_1 - f_1(a_1, \ldots, a_m)), \ldots, (y_n - f_n(a_1, \ldots, a_m))]$
7. compute Δ as a m-component vector and update $\mathbf{a}$
8. if the components of Δ are small enough, then end the process sending $\mathbf{a}^{(final)}$ to HOST, else go to 6.

In the current applications $n >> m$. This is the reason for which the matrix of $m \times m$ is send back to HOST for computing its inverse.

Example 11.2 *Let us consider an application with* $n = 1024$ *and the function* f *is*

$$f(x) = ax^2 + bx + c$$

Then, $m = 3$. *The next steps are used to compute the non-linear regression:*

1. *initialization:*

(a) *load the vectors* $\mathbf{x} = [x_1, \ldots, x_{1024}]$ *and* $\mathbf{y} = [y_1, \ldots, y_{1024}]$ *in ARRAY*

(b) *load the vector* $\mathbf{a}^{(0)} = [a^{(0)}, b^{(0)}, c^{(0)}]$ *in CONTROLLER's data memory*

2. *compute in ARRAY the matrix* $\mathbf{J}_f^T$ *as 3 1024-component vectors as follows: because* $\frac{\delta f}{\delta a} = x^2$, $\frac{\delta f}{\delta b} = x$, *and* $\frac{\delta f}{\delta c} = 0$, *the Jacobian is*

$$\mathbf{J} = \begin{bmatrix} \frac{\delta f_1}{\delta a} & \frac{\delta f_1}{\delta b} & \frac{\delta f_1}{\delta c} \\ \vdots & \vdots & \vdots \\ \frac{\delta f_{1024}}{\delta a} & \frac{\delta f_{1024}}{\delta b} & \frac{\delta f_{1024}}{\delta c} \end{bmatrix} = \begin{bmatrix} x_1^2 & x_1 & 0 \\ \vdots & \vdots & \vdots \\ x_{1024}^2 & x_{1024} & 0 \end{bmatrix}$$

where each column is represented as a n-component vector in ARRAY

3. *compute in ARRAY* $(\mathbf{J_f}^T \mathbf{J_f})^{-1} \mathbf{J_f}^T$ *as matrix* 3×3

4. *compute the inverse of* $(\mathbf{J_f}^T \mathbf{J_f})^{-1}$ *by:*

   - *sending the* 3×3 *matrix to HOST*
   - *computing in HOST the inverse*
   - *sending back to CONTROLLER the* 3×3 *resulting matrix*

5. *compute in ARRAY* $(\mathbf{J_f}^T \mathbf{J_f})^{-1} \mathbf{J_f}^T$ *as matrix* 3×1024

6. *compute in ARRAY the* 1024*-component vector* $\mathbf{r} = [(y_1 - f_1(a,b,c)), \ldots, (y_{1024} - f_{1024}(a,b,c))]$ *using* $\mathbf{a}^{(k)}$

7. *compute* Δ *as a* 3*-component vector and* $\mathbf{a}^{(k+1)} = \mathbf{a}^{(k)} + \Delta$

8. **if** *the components of* Δ *are small enough,* **then** *end the process sending* $\mathbf{a}^{(final)} = \mathbf{a}^{(k+1)}$ *to HOST,* **else** *go to 6.*

◇

Part V

ANNEXES

Appendix A

Kleene's Mathematical Model of Computation

A.1 The Recursive function Definition

From [Kleene '36] the following definition for partial recursive functions is extracted:

Definition A.1 *Any partial recursive function* $f : \mathbb{N}^n \to \mathbb{N}$ *can be computed using the* ***initial functions****:*

- $ZERO(x) = 0$*: the variable x takes the value zero*
- $INC(x) = x+1$*: increments the variable* $x \in \mathbb{N}$
- $SEL(i,x_1,\ldots,x_n) = x_i$*: i selects the value of* x_i *from the sequence* $X =< x_1,\ldots,x_n >$ *(called* identity function *in Kleene's paper)*

and the application of the following ***rules****:*

- Composition:
  $f(x_1,\ldots,x_n) = g(h_1(x_1,\ldots,x_n),\ldots,h_p(x_1,\ldots,x_n))$ *where: f is a total function if* $g : \mathbb{N}^p \to \mathbb{N}$ *and* $h_i : \mathbb{N}^n \to \mathbb{N}$*, for* $i = 1,\ldots p$*, are total functions*
- Primitive recursion:
  $f(x_1,\ldots,x_n,y) = g(x_1,\ldots,x_n,f(x_1,\ldots,x_n,y-1))$ *while*
  $f(x_1,\ldots,x_n,0) = h(x_1,\ldots,x_n)$*, where: f is a total function if g and h are total functions (called* ordinary recursion *in Kleene's paper)*
- Minimization (least-search):
  $f(x,y) = \mu y[g(x,y) = 0]$*, i.e., the rule computes the value of function* $f : \mathbb{N}^2 \to \mathbb{N}$ *as the smallest y for which the function* $g(x,y) = 0$*, if any.*

$\diamond$

A.2 Theorems

A.2.1 Preliminary definitions

used to prove the two theorems.

Definition A.2 *The reduction-less composition or* map composition*, MC, is, according to Definition A.1 (see Appendix A), the particular composition* $f : \mathbb{N}^n \to \mathbb{N}^p$ *where:*

$$f(X) = f(x_0, \ldots x_{n-1}) = \langle h_1(X), \ldots h_p(X) \rangle = \langle y_1, \ldots y_p \rangle$$

$h_i : \mathbb{N}^n \to \mathbb{N}$*, and* $g(y_1, \ldots, y_p) = \langle y_1, \ldots, y_p \rangle$ *is the identity function, for* $i = 1, \ldots, p$*.*

Definition A.3 *The map-less composition or* reduction composition*, RC, is according to Definition A.1, the particular composition* $f : \mathbb{N}^n \to \mathbb{N}$*, for* $n = p$*, where:*

$$f(X) = f(x_0, \ldots x_{n-1}) = g(x_1, \ldots x_p)$$

with $y_i = h_i(X) = SEL(i-1, X) = x_{i-1}$*, for* $i = 1, \ldots, p$*.*

According to the previous two definitions, composition is a ***map-reduce*** structure (Figure A.1), where a MC is serially connected with a RC. The two functional levels have physical implementations with the h_i functions and the g function embodied in various forms, starting from combinational circuits and reaching the complexity and competence of a processor, even of a computer.

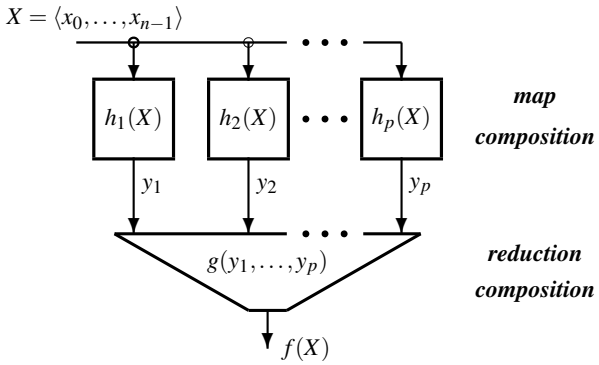

Figure A.1: **The circuit version of composition**. It is a two-layer construct: the parallel expanded *map* layer serially connected with the *reduction* layer.

Definition A.4 *For* $p = n-1$ *and* $X = \langle x_0, \ldots x_{n-1} \rangle$ *there is a MC with function CDR* $: \mathbb{N}^n \to \mathbb{N}^{n-1}$ *defined as:*

$$CDR(X) = \langle SEL(1, X), \ldots, SEL(n-1, X) \rangle = \langle x_1, \ldots, x_{n-1} \rangle$$

The CDR function deletes the first element of X.

Definition A.5 *The function* $C_i : \mathbb{N}^{i \times n} \to \mathbb{N}^{(i+1) \times n}$ *is the MC:*

$$C_i(Y) = \langle Y, P_i(X_i) \rangle = \langle X_1, \ldots, X_i, P_i(X_i) \rangle$$

where: $Y = \langle X_1, \ldots, X_i \rangle$*, the argument, is a sequence of sequences with* $X_j \in \mathbb{N}^n$*, for* $j = 1, \ldots, i$*, while* $h_j(Y) = SEL(j, Y) = X_j$ *for* $j = 1, 2, \ldots, i$ *and* $h_{i+1}(Y) = P_i(SEL(i, Y)) = P_i(X_i)$*.*

While the CDR function deletes the first element from X, the C function adds, at the end of the sequence X, a new element computed from the last element of X.

Definition A.6 *The k-time application of* C_i *(see Figure A.2a), starting from* $C_1(\langle X \rangle) = \langle X, P_1(X) \rangle$*, where the argument* $\langle X \rangle$ *is a n-scalar sequence, defines the* multi-output pipeline*, MOP:*

$$MOP(X) = \langle X, P_1(X), \ldots, P_k(P_{k-1}(\ldots, (P_1(X) \ldots))), \ldots \rangle$$

The resulting structure, with one sequence, $Y = \langle X \rangle$*, as argument and as many as necessary computed values,* $P_k(P_{k-1}(\ldots, (P_1(X) \ldots)))$*, is represented in Figure A.2b.*

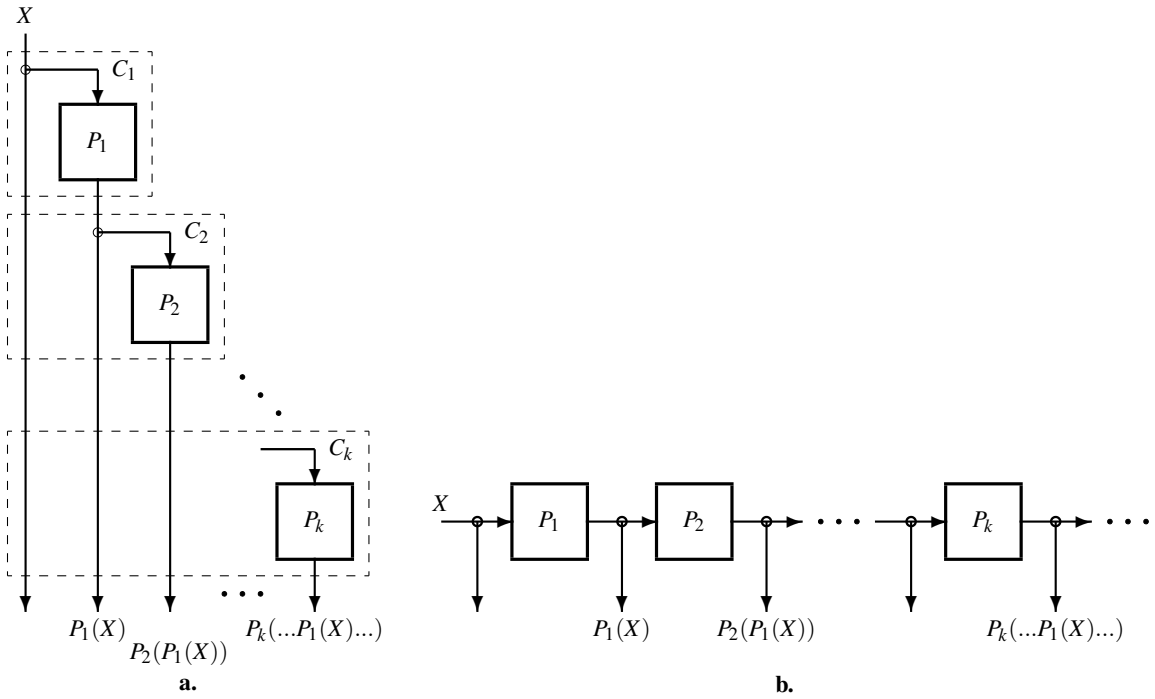

Figure A.2: **The multi-output pipeline structure (MOP)**. **a**. The explicit application of C_i. **b**. The resulting MOP circuit structure.

The function $MOP(X)$ is a total function if the functions P_i are total functions, since it is computed using only the repeated application of the composition C_i. For the theoretical model, k is not limited to a specific value. By defining this sequence of MCs, a left to right serial connection between cells is added to the general form of the MC structure.

Definition A.7 *The function* $PS : \{\{0,1\} \times \mathbb{N}\}^n \to \mathbb{N}$*, called* predicated selection*, is*

$$PS(S) = ADD(V(SEL(1,S)),\ldots,V(SEL(i,S)),\ldots)$$

where: $S = \langle\langle p_1,s_1\rangle,\langle p_2,s_2\rangle,\ldots\rangle$*, takes a sequence of pairs* $\langle predicate, scalar\rangle$*, and returns a scalar. The function ADD adds the scalars validated by the function* $V(\langle p,s\rangle) = p\ ?\ s\ :\ 0$*.*

V is the function *mapped* on the first level of *PS*, while ADD is the *reduction* function associated to the same composition. When no more than one p_i takes the value 1, the function is used to select a scalar from the vector $S = \langle s_1,s_2,\ldots\rangle$.

A.2.2 Primitive Recursion as a Sequence of Compositions

Theorem A.1 *The primitive recursive rule is reducible to repeated applications of specific compositions.*

Proof A.1 *The primitive recursion rule (see Definition A.1) could be applied using its iteratively expanded form:*

$$f(x,y) = g(x,f(x,y-1)) = \ldots =$$

$$= \underbrace{g(x,g(x,g(x,\ldots g}_{y-1\ times}(x,f(x,1))\ldots))) =$$

$$= \underbrace{g(x, g(x, g(x, \ldots g}_{y\ times}(x, f(x,0)) \ldots))) =$$

$$= \underbrace{g(x, g(x, g(x, \ldots g}_{y\ times}(x, h(x)) \ldots)))$$

Let be, in Figure A.3, the specific instantiation of the MOP function (see Definition A.6). It computes iteratively, starting in the first stage with the function $f(x,0) = h(x)$, the values $f(x,i)$ for $i = 0,1,....$ In each stage the predicate $i == y$ is provided. The PS function takes from the MOP its arguments as pairs of $D_i = \langle predicate_i, value_i \rangle = \langle (i == y), f(x,i) \rangle$. Because only one D_i has $predicate_i = 1$, PS returns the result.

Formally, the functions P_i return a quintuples, as follows:

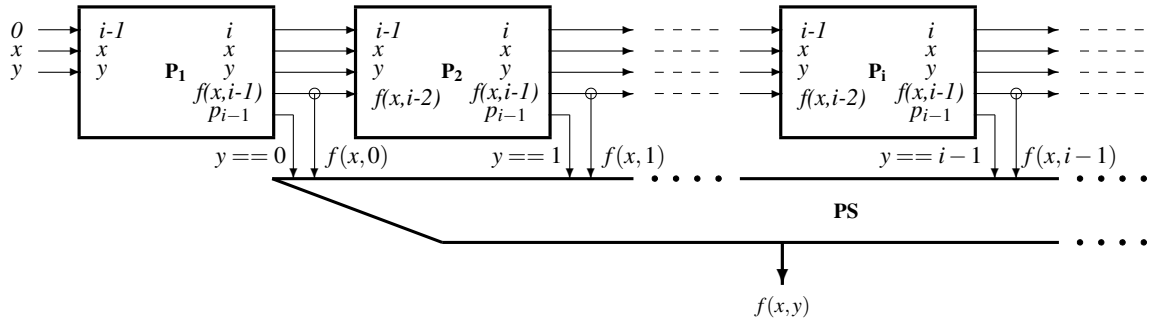

Figure A.3: **The MOP structure for partial recursive computation**.

$$P_1(\langle 0,x,y \rangle) = \langle p_0, f(x,0), 1, x, y \rangle = \langle p_0, h(x), 1, x, y \rangle$$

and

$$P_i(CDR(P_{i-1})) = P_i(\langle f(x,i-2), i-1, x, y \rangle) =$$
$$= \langle p_{i-1}, f(x,i-1), i, x, y \rangle$$

for $i = 2,3,\ldots$, where p_i is the predicate $p_i = (y == i)$, for $i = 1,2,\ldots$. From the values computed by MOP we select: $S = \langle D_1, \ldots, D_i, \ldots \rangle$ where each doublet D_i is

$$D_i = \langle SEL(1,P_i), SEL(2,P_i) \rangle$$

The sequence S is used as input for the function PS which provides the result $PS(S) = f(x,y)$. Thus, for primitive recursion we need to compose two compositions, MOP and PS.

Figure A.3 presents the circuit version of the function obtained by composing a specific MOP with the PS function. The two stage computation just described, as a structure indefinitely extensible to the right, is a theoretical model because the index i takes values no matter how large, similar with the "infinite" tape of Turing's machine. It is important that the algorithmic complexity of the description is in $O(1)$, because the functions P_i, *MOP* and *PS* have constant size descriptions.

A.2.3 Minimalization as a Sequence of Compositions

Theorem A.2 *The minimization (least-search) rule is reducible to repeated applications of specific compositions.*

Proof A.2 *The minimization (least-search) rule computes the value of $f(x)$ as the smallest, if any, y for which $g(x,y) = 0$. Let be, a specific instantiation of the MOP function (see Definition A.6) in Figure A.4. In each stage, P_i, is computed the value of the function $g(x,i-1)$ and is evaluated if the result is zero. The predicate's evaluation is stopped when it provides the first hit, if any. The pairs $\langle predicate, index \rangle$ are the arguments of a PS function which returns the index, i, paired by 1, if any, else the function returns 0. If, for $x = a$, the value is 0, then the function $f(a)$ is not defined for a, else $f(a) = i-1$.*

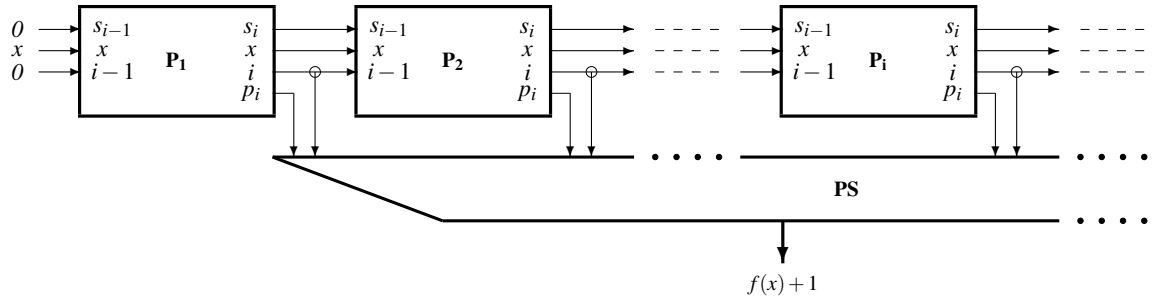

Figure A.4: **The MOP structure for minimalization**.

Each stage in MOP is defined as follows:

$$P_i(\langle i-1, x, s_{i-1}\rangle) = \langle p_i, i, x, s\rangle$$

where there are involved two predicates:

$$p_i = SEL(s_{i-1}, \langle (g(x, i-1) == 0), 0\rangle)$$

and

$$s_i = SEL(s_{i-1}, \langle p_i, 1\rangle)$$

The modules P_i in MOP are interconnected as follows:

$$P_i(CDR(P_{i-1})) = \langle p_i, i, x, s\rangle$$

with: $X = \langle 0, x, 0\rangle$.

From the values computed by the MOP function we select: $S = \langle D_1, \ldots, D_i, \ldots\rangle$ where each doublet D_i is of the form

$$D_i = \langle SEL(1, P_i), SEL(2, P_i)\rangle$$

*to be used as input for the function PS which allows us to compute $f(x,y) = PS(S) - 1$ **only if** $PS(S) \neq 0$.*

*Each stage of the MOP function propagates the value of x, provides the index, i, computes the function $g(x,i)$ and compares the results with zero, providing the predicate p_i, and computes the predicate s_i used as stop signal. The function PS selects, **if any**, from the outputs of the MOP the incremented index, corresponding to the occurrence of $p_i = 1$.*

The computation just described is only a theoretical model, because the index *i* has an indefinitely large value. But, the size of the algorithmic description remains $O(1)$.

Corollary A.1 *Any computation defined in Definition A.1 (see Appendix A) can be done, according to Theorem A.1 and Theorem A.2, using the initial functions and the repeated application of the composition rule.*

Bibliography

[Alfke '05] Peter Alfke: "Metastable Recovery in Virtex-II Pro FPGAs", *Application Note: Virtex-II Pro Family*, [Online]. Available: `http://www.xilinx.com/support/documentation/application\_notes/xapp094.pdf`, XILINX, 2005.

[Andonie '95] Răzvan Andonie, Ilie Gârbacea: *Algoritmi fundamentali. O perspectivă C^{++}*, Ed. Libris, Cluj-Napoca, 1995. (in Roumanian)

[Ajtai '83] M. Ajtai, et al.: "An O(n log n) sorting network", Proc. 15th Ann. ACM Symp. on Theory of Computing, Boston, Mass., 1983.

[Backus '78] J. Backus: "Can programming be liberated from the von neumann style? a functional style and its algebra of programs" *Communications of the ACM*, 21:613–641, August 1978. ,

[Batcher '68] K. E. Batcher: "Sorting networks and their applications", in *Proc. AFIPS Spring Joint Computer Conference*, vol. 32, 1968.

[Benes '68] Václav E. Beneš: *Mathematical Theory of Connecting Networks and Telephone Traffic*. New York: Academic, 1968.

[Blakeslee '79] T. R. Blakeslee: *Digital Design with Standard MSI and LSI*, John Wiley & Sons, 1979.

[Booth '67] T. L. Booth: *Sequential Machines and Automata Theory*, John Wiley & Sons, Inc., 1967.

[Bremermann '62] H. G. Bremermann: "Optimization through Evolution and Recombination", in *Self-Organizing Systems*, ed.: M. C. Yovits, S. Cameron, Washington DC, Spartan, 1962.

[Calude '82] Cristian Calude: *Complexitatea calculului. Aspecte calitative* (The Complexity of Computation. Qualitative Aspects), Ed. Stiintifica si Enciclopedica, Bucuresti, 1982.

[Calude '94] Cristian Calude: *Information and Randomness*, Springer-Verlag, 1994.

[Casti '92] John L. Casti: *Reality Rules: II. Picturing the World in Mathematics - The Frontier*, John Wiley & Sons, Inc., 1992.

[Cavanagh '07] Joseph Cavanagh: *Sequential Logic. Analysis and Synthesis*, CRC Taylor & Francis, 2007.

[Chaitin '66] Gregory Chaitin: "On the Length of Programs for Computing Binary Sequences", *J. of the ACM*, Oct., 1966.

[Chaitin '70] Gregory Chaitin: "On the Difficulty of Computation", in *IEEE Transactions of Information Theory*, ian. 1970.

[Chaitin '77] Gregory Chaitin: "Algorithmic Information Theory", in *IBM J. Res. Develop.*, Iulie, 1977.

[Chaitin '87] Gregory Chaitin: *Algorithmic Information Theory*, Cambridge University Press, 1987.

[Chaitin '90] Gregory Chaitin: *Information, Randomness and Incompletness*, World Scientific,1990.

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

[Chaitin '06] Gregory Chaitin: "The Limit of Rason", in *Scientific American*, Martie, 2006.

[1] Noam Chomsky, "Three Models for the Description of Languages", *IEEE Trans. on Information Theory*, 2:3 , 1956.

[2] Noam Chomsky: *Syntactic Structures*. Mouton, The Hague, 1957.

[3] Noam Chomsky, "On Certain Formal Properties of Grammars", *Information and Control*, 2:2, 1959.

[4] Noam Chomsky, "Formal Properties of Grammars", *Handbook of Mathematical Psychology*, Wiley, New-York, 1963.

[Church '36] Alonzo Church: "An Unsolvable Problem of Elementary Number Theory", in *American Journal of Mathematics*, vol. 58, pag. 345-363, 1936.

[Clare '72] C. Clare: *Designing Logic Systems Using State Machines*, Mc Graw-Hill, Inc., 1972.

[Cormen '90] Thomas H. Cormen, Charles E. Leiserson, Donsld R. Rivest: *Introduction to Algorithms*, MIT Press, 1990.

[Dascălu '98] Monica Dascălu, Eduard Franţi, Gheorghe Ştefan: "Modeling Production with Artificial Societies: the Emergence of Social Structure", in S. Bandini, R. Serra, F. Suggi Liverani (Eds.): *Cellular Automata: Research Towars Industry. ACRI '98 - Proceedings of the Third Conference on Cellular Automata for Research and Industry*, Trieste, 7 - 9 October 1998, Springer Verlag, 1998. p 218 - 229.

[Dascălu '98a] Monica Dascălu, Eduard Franţi, Gheorghe Ştefan: "Artificial Societies: a New Paradigm for Complex Systems' Modeling", in *IFAC Conference on Supplemental Ways for Improving International Stability - SWIIIS '98*, May 14-16, Sinaia, 1998. p.62-67.

[Drăgănescu '84] Mihai Drăgănescu: "Information, Heuristics, Creation", in Plauder, I. (ed): *Artificial Inteligence and Information Control System of Robots*, Elsevier Publishers B. V. (North-Holland), 1984.

[Drăgănescu '91] Mihai Drăgănescu, Gheorghe Ştefan, Cornel Burileanu: *Electronica functională*, Ed. Tehnică, Bucureşti, 1991 (in Roumanian).

[Einspruch '86] N. G. Einspruch ed.: *VLSI Electronics. Microstructure Science. vol. 14 : VLSI Design*, Academic Press, Inc., 1986.

[Einspruch '91] N. G. Einspruch, J. L. Hilbert: *Application Specific Integrated Circuits (ASIC) Technology*, Academic Press, Inc., 1991.

[Ercegovac '04] Miloš D. Ercegovac, Tomás Lang: *Digital Arithmetic*, Morgan Kaufman, 2004.

[Fischer '04] Igor Fischer, Jan Poland: *New Methods for Spectral Clustering*, Technical Report IDSIA-12-04, 2004.

[Flynn '72] Flynn, M.J.: "Some computer organization and their affectiveness", *IEEE Trans. Comp.* C21:9 (Sept. 1972), pp. 948-960.

[Gheolbanoiu '14] Alexandru Gheolbanoiu, Dan Mocanu, Radu Hobincu, Lucian Petrica: "Cellular Automaton pRNG with a Global Loop for Non-Uniform Rule Control", 18th International Conference on Ciruits, Systems, Communications and Computers (CSCC 2014), Santorini Island, Greece, July 17-21, 2014, vol. II, 415-420.[Online]. Available: `http://www.europment.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-15.pdf`

[Glushkov '66] V. M. Glushkov: *Introduction to Cybernetics*, Academic Press, 1966.

[Gödels '31] Kurt Gödel: "On Formally Decidable Propositions of Principia Mathematica and Related Systems I", reprinted in S. Fefermann et all.: *Collected Works I: Publications 1929 - 1936,* Oxford Univ. Press, New York, 1986.

[Hamad '08] Denis Hamad, Philippe Biela: *Introduction to spectral clustering*, 2008. [Online]. Available: `http://lagis-vi.univ-lille1.fr/~lm/classpec/reunion_28_02_08/Introduction_to_spectral_clustering.pdf`

[Hartley '95] Richard I. Hartley: *Digit-Serial Computation*, Kulwer Academic Pub., 1995.

[Hascsi '95] Zoltan Hascsi, Gheorghe Ştefan: "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.

[Hascsi '96] Zoltan Hascsi, Bogdan Mîţu, Mariana Petre, Gheorghe Ştefan, "High-Level Synthesis of an Enchanced Connex memory", in *Proceedings of the International Semiconductor Conference*, Sinaia, October 1996, p. 163-166.

[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.

[Helbing 89'] Walter A. Helbing, Veljko M. Milutinovic: "Architecture and Design of a 32-bit GaAs Microprocessor", in [Milutinovic 89'].

[Hennessy '07] John L. Hennessy, David A. Patterson: *Computer Architecture: A Quantitative Approach*, Fourth Edition, Morgan Kaufmann, 2007.

[Hennie '68] F. C. Hennie: *Finite-State Models for Logical Machine*, John Wiley & Sons, Inc., 1968.

[Hillis '85] W. D. Hillis: *The Connection Machine*, The MIT Press, Cambridge, Mass., 1985.

[Kaeslin '01] Hubert Kaeslin: *Digital Integrated Circuit Design*, Cambridge Univ. Press, 2008.

[Keeth '01] Brent Keeth, R. jacob Baker: *DRAM Circuit Design. A Tutorial*, IEEE Press, 2001.

[Kleene '36] Stephen C. Kleene: "General Recursive Functions of Natural Numbers", in *Math. Ann.*, 112, 1936.

[Karim '08] Mohammad A. Karim, Xinghao Chen: *Digital Design*, CRC Press, 2008.

[Knuth '73] D. E. Knuth: *The Art of Programming. Sorting and Searching*, Addison-Wesley, 1973.

[Kolmogorov '65] A.A. Kolmogorov: "Three Approaches to the Definition of the Concept "Quantity of Information" ", in *Probl. Peredachi Inform.*, vol. 1, pag. 3-11, 1965.

[Kung '79] H. T. Kung, C. E. Leiserson: "Algorithms for VLSI processor arrays", in [Mead '79].

[Ladner '80] R. E. ladner, M. J. Fischer: "Parallel prefix computation", *J. ACM*, Oct. 1980.

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

[MacQueen '67] J. B. MacQueen (1967): "Some Methods for classification and Analysis of Multivariate Observations", Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability", Berkeley, University of California Press, 1:281-297

[Maliţa '06] Mihaela Maliţa, Gheorghe Ştefan, Marius Stoian: "Complex vs. Intensive in Parallel Computation", in *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C - ICCGI*, 2006 Bucharest, Romania, August 1-3, 2006

[Maliţa '07] Mihaela Maliţa, Gheorghe Ştefan, Dominique Thiébaut: "Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" in *International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing* June 17, 2007 Seattle, WA, USA.

[Maliţa '13] Mihaela Maliţa, Gheorghe M. Ştefan: "Control Global Loops in Self-Organizing Systems", *ROMJIST*, Volume 16, Numbers 23, 2013, 177-191. [Online]. Available: `http://www.imt.ro/romjist/Volum16/Number16\_2/pdf/05-Malita-Stefan2.pdf`

[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)

[Mead '79] Carver Mead, Lynn Convay: *Introduction to VLSI Systems*, Addison-Wesley Pub, 1979.

[MicroBlaze] *** *MicroBlaze Processor. Reference Guide.* [Online]. Available: `http://www.xilinx.com/support/documentation/sw\_manuals/xilinx14\_1/mb\_ref\_guide.pdf`

[Milutinovic 89'] Veljko M. Milutinovic (ed.): *High-Level Language Computer Architecture*, Computer Science Press, 1989.

[Mindell '00] Arnold Mindell: *Quantum Mind. The Edge Between Physics and Psychology*, Lao Tse Press, 2000.

[Minsky '67] M. L. Minsky: *Computation: Finite and Infinite Machine*, Prentice - Hall, Inc., 1967.

[Mîţu '00] Bogdan Mîţu, Gheorghe Ştefan, "Low-Power Oriented Microcontroller Architecture", in *CAS 2000 Proceedings*, Oct. 2000, Sinaia, Romania

[Moto-Oka '82] T. Moto-Oka (ed.): *Fifth Generation Computer Systems*, North-HollandPub. Comp., 1982.

[Omondi '94] Amos R. Omondi: *Computer Arithmetic. Algorithm, Architecture and Implementation*, Prentice Hall, 1994.

[Palnitkar '96] Samir Palnitkar: *Verilog HDL. AGuide to Digital Design and Synthesis*, SunSoft Press, 1996.

[Parberry 87] Ian Parberry: *Parallel Complexity Theory*. Research Notes in Theoretical Computer science. Pitman Publishing, London, 1987.

[Parberry 94] Ian Parberry: *Circuit Complexity and Neural Networks*, The MIT Presss, 1994.

[Patterson '05] David A. Patterson, John L.Hennessy: *Computer Organization & Design. The Hardware / Software Interface*, Third Edition, Morgan Kaufmann, 2005.

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

[Păun '85] A. Păun, Gh. Ştefan, A. Birnbaum, V. Bistriceanu, "DIALISP - experiment de structurare neconventionala a unei masini LISP", in *Calculatoarele electronice ale generatiei a cincea*, Ed. Academiei RSR, Bucuresti 1985. p. 160 - 165.

[Post '36] Emil Post: "Finite Combinatory Processes. Formulation I", in*The Journal of Symbolic Logic*, vol. 1, p. 103 -105, 1936.

[Prince '99] Betty Prince: *High Performance Memories. New architecture DRAMs and SRAMs evolution anad function*, John Wiley & Sons, 1999.

[Rafiquzzaman '05] Mohamed Rafiquzzaman: *Fundamentals of Digital Logic and Microcomputer Design*, Fifth Edition, Wiley – Interscience, 2005.

[Salomaa '69] Arto Salomaa: *Theory of Automata*, Pergamon Press, 1969.

[Salomaa '73] Arto Salomaa: *Formal Languages*, Academic Press, Inc., 1973.

[Salomaa '81] Arto Salomaa: *Jewels of Formal Language Theory*, Computer Science Press, Inc., 1981.

[Savage '87] John Savage: *The Complexity of Computing*, Robert E. Krieger Pub. Comp., 1987.

[Shankar '89] R. Shankar, E. B. Fernandez: *VLSI Computer Architecture*, Academic Press, Inc., 1989.

[Shannon '38] C. E. Shannon: "A Symbolic Annalysis of Relay and Switching Circuits", *Trans. AIEE*, vol. 57, p.713-723, 1938.

[Shannon '48] C. E. Shannon: "A Mathematical Theory of Communication", *Bell System Tech. J.*, Vol. 27, 1948.

[Shannon '56] C. E. Shannon: "A Universal Turing Machine with Two Internal States", in *Annals of Mathematics Studies, No. 34: Automata Studies*, Princeton Univ. Press, pp 157-165, 1956.

[Sharma '97] Ashok K. Sharma: *Semiconductor Memories. Techology, Testing, and Reliability*, Wiley – Interscience, 1997.

[Sharma '03] Ashok K. Sharma: *Advanced Smiconductor Memories. Architectures, Designs, and Applications*, Whiley-Interscience, 2003.

[Solomonoff '64] R. J. Solomonoff: "A Formal Theory of Inductive Inference", in *Information and Control*, vol. 7, pag. 1- 22 , pag. 224-254, 1964.

[Spira '71] P. M. Spira: "On time-Hardware Complexity Tradeoff for Boolean Functions", in *Preceedings of Fourth Hawaii International Symposium on System Sciences*, pp. 525-527, 1971.

[Stoian '07] Marius Stoian, Gheorghe Ştefan: "Stacks or File-Registers in Cellular Computing?", in *CAS, Sinaia 2007*.

[Streinu '85] Ileana Streinu: "Explicit Computation of an Independent Gödel Sentence", in *Recursive Functions Theory Newsletter*, June 1985.

[Ştefan '97] Denisa Ştefan, Gheorghe Ştefan, "Bi-thread Microcontroller as Digital Signal Processor", in *CAS '97 Proceedings, 1997 International Semiconductor Conference*, October 7 -11, 1997, Sinaia, Romania.

[Ştefan '99] Denisa Ştefan, Gheorghe Ştefan: "A Procesor Network without Interconnectio Path", in *CAS 99 Proceedings, Oct., 1999*, Sinaia, Romania. p. 305-308.

[Ştefan '80] Gheorghe Ştefan: *LSI Circuits for Processors*, Ph.D. Thesis (in Roumanian), Politechnical Institute of Bucharest, 1980.

[Ştefan '83] Gheorghe Ştefan: "Structurari neechilibrate in sisteme de prelucrare a informatiei", in *Inteligenta artificiala si robotica*, Ed. Academiei RSR, Bucuresti, 1983. p. 129 - 140.

[Ştefan '83] Gheorghe Ştefan, et al.: *Circuite integrate digitale*, Ed. Did. si Ped., Bucuresti, 1983.

[Ştefan '84] Gheorghe Ştefan, et al.: "DIALISP - a LISP Machine", in *Proceedings of the ACM Symposium on LISP and Functional Programming*, Austin, Texas, Aug. 1984. p. 123 - 128.

[Ştefan '85] Gheorghe Ştefan, A. Păun, "Compatibilitatea functie - structura ca mecanism al evolutiei arhitecturale", in *Calculatoarele electronice ale generatiei a cincea*, Ed. Academiei RSR, Bucuresti, 1985. p. 113 - 135.

[Ştefan '85a] Gheorghe Ştefan, V. Bistriceanu, A. Păun, "Catre un mod natural de implementare a LISP-ului", in *Sisteme cu inteligenta artificiala*, Ed. Academiei Romane, Bucuresti, 1991 (paper at *Al doilea simpozion national de inteligenta artificiala*, Sept. 1985). p. 218 - 224.

[Ştefan '86] Gheorghe Stefan, M. Bodea, "Note de lectura la volumul lui T. Blakeslee: Proiectarea cu circuite MSI si LSI", in *T. Blakeslee: Prioectarea cu circuite integrate MSI si LSI*, Ed. Tehnica, Bucuresti, 1986 (translated from English by M. Bodea, M. Hancu, Gh. Stefan). p. 338 - 364.

[Ştefan '86a] Gheorghe Stefan, "Memorie conexa" in *CNETAC 1986* Vol. 2, IPB, Bucuresti, 1986, p. 79 - 81.

[Ştefan '91] Gheorghe Ştefan: *Functie si structura in sistemele digitale*, Ed. Academiei Romane, 1991.

[Ştefan '91] Gheorghe Ştefan, 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 '93] Gheorghe Ştefan: *Circuite integrate digitale*. Ed. Denix, 1993.

[Ştefan '95] Gheorghe Ştefan, 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] Gheorghe Ştefan, Mihaela Maliţa: "Chaitin's Toy-Lisp on Connex Memory Machine", *Journal of Universal Computer Science*, vol. 2, no. 5, 1996, pp. 410-426.

[Ştefan '97] Gheorghe Ştefan, Mihaela Maliţa: "DNA Computing with the Connex Memory", in *RECOMB 97 First International Conference on Computational Molecular Biology*. January 20 - 23, Santa Fe, New Mexico, 1997. p. 97-98.

[Ştefan '97a] Gheorghe Ştefan, Mihaela Maliţa: " 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] Gheorghe Ştefan, "Silicon or Molecules? What's the Best for Splicing", in Gheorghe Păun (ed.): *Computing with Bio-Molecules. Theory and Experiments.* Springer, 1998. p. 158-181

[Ştefan '98a] Gheorghe Ştefan, " "Looking for the Lost Noise" ", in *CAS '98 Proceedings, Oct. 6 - 10, 1998*, Sinaia, Romania. p.579 - 582.
`http://arh.pub.ro/gstefan/CAS98.pdf`

[Ştefan '98b] Gheorghe Ştefan, "The Connex Memory: A Physical Support for Tree / List Processing" in *The Roumanian Journal of Information Science and Technology*, Vol.1, Number 1, 1998, p. 85 - 104.

[Ştefan '98] Gheorghe Ştefan, Robrt Benea: "Connex Memories & Rewrieting Systems", in *MELECON '98*, Tel-Aviv, May 18 -20, 1998.

[Ştefan '99] Gheorghe Ştefan, Robert Benea: "Experimente in info cu acizi nucleici", in M. Drăgănescu, Ştefan Trăusan-Matu (eds): *Natura realitatii fizice si a informatiei*, Editura Tempus, 1999.

[Ştefan '99a] Gheorghe Ştefan: "A Multi-Thread Approach in Order to Avoid Pipeline Penalties", in *Proceedings of 12th International Conference on Control Systems and Computer Science*, Vol. II, May 26-29, 1999, Bucharest, Romania. p. 157-162.

[Ştefan '00] Gheorghe Ştefan: "Parallel Architecturing starting from Natural Computational Models", in *Proceedings of the Romanian Academy*, Series A: Mathematics, Physics, Technical Sciences, Information Science, vol. 1, no. 3 Sept-Dec 2000.

[Ştefan '01] Gheorghe Ştefan, Dominique Thiébaut, "Hardware-Assisted String-Matching Algorithms", in *WABI 2001, 1st Workshop on Algorithms in BioInformatics, BRICS*, University of Aarhaus, Danemark, August 28-31, 2001.

[Ştefan '04] Gheorghe Ştefan, Mihaela Maliţa: "Granularity and Complexity in Parallel Systems", in *Proceedings of the 15 IASTED International Conf, 2004*, Marina Del Rey, CA, ISBN 0-88986-391-1, pp.442-447.

[Ştefan '06] Gheorghe Ştefan: "Integral Parallel Computation", in *Proceedings of the Romanian Academy, Series A: Mathematics, Physics, Technical Sciences, Information Science*, vol. 7, no. 3 Sept-Dec 2006, p.233-240.

[Ştefan '06a] Gheorghe Ştefan: "A Universal Turing Machine with Zero Internal States", in *Romanian Journal of Information Science and Technology*, Vol. 9, no. 3, 2006, p. 227-243

[Ştefan '06b] Gheorghe Ştefan: "The CA1024: SoC with Integral Parallel Architecture for HDTV Processing", invited paper at *4th International System-on-Chip (SoC) Conference & Exhibit*, November 1 & 2, 2006, Radisson Hotel Newport Beach, CA

[Ştefan '06c] Gheorghe Ştefan, Anand Sheel, Bogdan Mîţu, Tom Thomson, Dan Tomescu: "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing", in *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University, August 20 to 22, 2006.

[Ştefan '06d] Gheorghe Ştefan: "The CA1024: A Massively Parallel Processor for Cost-Effective HDTV", in *SPRING PROCESSOR FORUM: Power-Efficient Design*, May 15-17, 2006, Doubletree Hotel, San Jose, CA.

[Ştefan '06e] Gheorghe Ştefan: "The CA1024: A Massively Parallel Processor for Cost-Effective HDTV", in *SPRING PROCESSOR FORUM JAPAN*, June 8-9, 2006, Tokyo.

[Ştefan '07] Gheorghe Ştefan: "Membrane Computing in Connex Environment", invited paper at *8th Workshop on Membrane Computing (WMC8)* June 25-28, 2007 Thessaloniki, Greece

[Ştefan '07a] Gheorghe Ştefan, Marius Stoian: "The efficiency of the register file based architectures in OOP languages era", in *SINTES13* Craiova, 2007.

[Ştefan '07b] Gheorghe Ştefan: "Chomsky's Hierarchy & a Loop-Based Taxonomy for Digital Systems", in *Romanian Journal of Information Science and Technology* vol. 10, no. 2, 2007.

[Ştefan '14] Gheorghe M. Stefan, Mihaela Malita: "Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation", *18th International Conference on Ciruits, Systems, Communications and Computers (CSCC 2014)*, Santorini Island, Greece, July 17-21, 2014, 582-597. [Online]. Available: `http://www.inase.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-42.pdf`

[Sutherland '02] Stuart Sutherland: *Verilog 2001. A Guide to the New Features of the Verilog Hardware Description Language*, Kluwer Academic Publishers, 2002.

[Tabak '91] D. Tabak: *Advanced Microprocessors*, McGrow- Hill, Inc., 1991.

[Tanenbaum '90] A. S. Tanenbaum: *Structured Computer Organisation* third edition, Prentice-Hall, 1990.

[Theobald '91] Oliver Theobald: Machine Learning for Absolute Beginners. Kindle Edition.

[Thiébaut '06] Dominique Thiébaut, Gheorghe Ştefan, Mihaela Maliţa: "DNA search and the Connex technology" in *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C - ICCGI*, 2006 Bucharest, Romania, August 1-3, 2006

[ThoughtSpot] ThoughtSpot: *SpotIQ AI-Driven Analytics. Architecting Automated Insights for the Masses*, White paper, [Online]. Available: `https://go.thoughtspot.com/rs/816-MBH-536/images/ThoughtSpot-SpotIQ-AI-Driven-Analytics-White-Paper.pdf`

[Tokheim '94] Roger L. Tokheim: *Digital Principles*, Third Edition, McGraw-Hill, 1994.

[Turing '36] Alan M. Turing: "On computable Numbers with an Application to the Eintscheidungsproblem", in *Proc. London Mathematical Society,* 42 (1936), 43 (1937).

[Vahid '06] Frank Vahid: *Digital Design*, Wiley, 2006.

[von 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.

[Uyemura '02] John P. Uyemura: *CMOS Logic Circuit Design*, Kluver Academic Publishers, 2002.

[Ward '90] S. A. Ward, R. H. Halstead: *Computation Structures*, The MIT Press, McGraw-Hill Book Company, 1990.

[Wedig '89] Robert G. Wedig: "Direct Correspondence Architectures: Principles, Architecture, and Design" in [Milutinovic '89].

[Waksman '68] Abraham Waksman, "A permutation network," in *J. Ass. Comput. Mach.*, vol. 15, pp. 159-163, Jan. 1968.

[webRef_1] [Online]. Available: `http://www.fpga-faq.com/FAQ_Pages/0017_Tell_me_about_metastables.htm`

[webRef_2] [Online]. Available: `http://www.fpga-faq.com/Images/meta_pic_1.jpg`

[webRef_3] [Online]. Available: `http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html#fsa_pfx`

[Weste '94] Neil H. E. Weste, Kamran Eshraghian: *Principle of CMOS VLSI Design. ASystem Perspective*, Second Edition, Addisson Wesley, 1994.

[Wolfram '02] Stephen Wolfram: *A New Kind of Science*, Wolfram Media, Inc., 2002.

[Zurada '95] Jacek M. Zurada: *Introductin to Artificial Neural network*, PWS Pub. Company, 1995.

[Yanushkevich '08] Svetlana N. Yanushkevich, Vlad P. Shmerko: *Introduction to Logic Design*, CRC Press, 2008.

[http:GaussNewtonMethod] http://fourier.eng.hmc.edu/e176/lectures/NM/node36.html

[http:eigenValuesVectors] Numerical Determination of Eigenvalues and Eigenvectors