FUNCTIONAL ELECTRONICS

*

Lecture Notes on Parallel Embedded Systems

(work in progress)

Gheorghe M. Ştefan

– 2025 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 Heterogenous Parallel Accelerators (HPA). In Figure 1, HPFE includes, by turn:

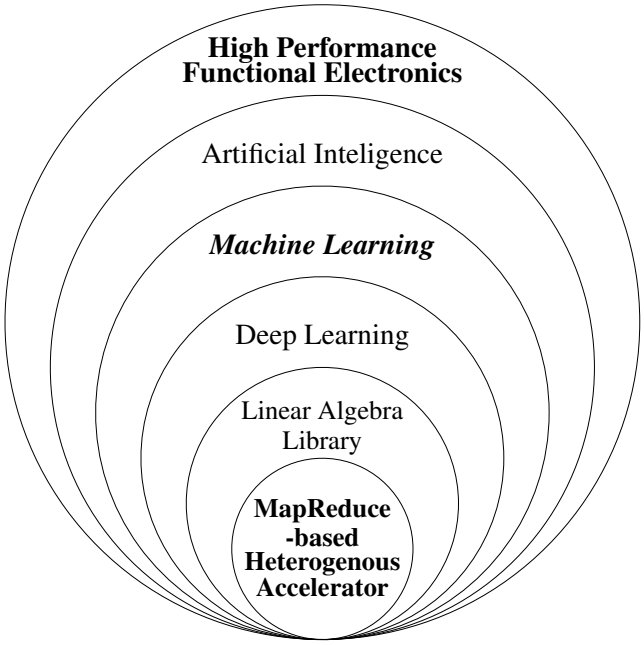

Figure 1:

- 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

Part I: Function & Structure & Information

Chapter 2 Formal Languages is intended as an introduction to Noam Chomsky's theory of formal languages to support the association between function and structure from the perspective of formal language of description.

Chapter 3 Structures & Languages associates to each type of language a structure with a certain level of autonomy given by the number of loops that define the functionality of the system, so that languages of type three are associated with systems with two loops, those of type two with systems with three loops, and those of type one to four-loop systems.

Chapter 4 Loops & Information qualitatively defines the concept of information as a structure with syntactic order that acts through the associated meaning in a given context.

Part II: The Parallel Engine

Chapter 5 What Means Parallel Computation introduces, starting form the computational model proposed by Stephen Kleene, the concept of ***parallel computation***.

Chapter 6 The Generic Parallel Engine describes the generic version of the parallel accelerator used in these lectures.

Part III: Berkeley view of parallel computing

Chapter 7 Berkely's View presents the 13 class of problems ("dwarfs") typical for parallel accelerators.

Chapter 8 The first Dwarf: Dense Linear Algebra presents the dense linear algebra problems.

Chapter 9 The Second Dwarf: Sparse Linear Algebra presents the sparse linear algebra problems.

Part IV: Machine Learning

Chapter 10 What is Machine Learning? is answered by *learning from data*.

Chapter 11 Clustering

Chapter 12 Regresion

Chapter 13 Markov Models

Appendix A introduces several ways of dealing with the histories leading to the emergence and development of computer science and technology.

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

Appendix C describes the design and use of a heterogeneous computing system based on a MapScanReduce accelerator introduced in Chapter 5.

Contents

Contents (detailed)

Part I

Function & Structure & Information

Chapter 1

Functional Approach

In this section the meaning of the term ***Functional Electronics***, FE, is explained. The origin of the term, how its perception and application evolved, but especially the perspective offered to digital electronics.

1.1 The Origin of the Term Functional Electronics

The term functional electronics was first used in 1959 by J. A. Morton, from Bell Telephone Laboratories, in an attempt to find an alternative to the excessive structuring mechanisms in electronics. With this he tried to launch an offensive against the ”tyranny of numbers”, that is, against the tendencies of an excessive formal approach. In this first understanding, functional electronics tried to achieve functions based only on the intrinsic properties of certain materials, avoiding circuit configurations and numerical modeling. Too many networks and too many numbers!

The evolution of research and its market did not confirm, as expected, this orientation. J. A. Morton's main error was that he did not make the necessary distinction between the numerical and the non-numerical, thinking of computer science as an exclusive domain of the numerical. By doing so, he missed the fusion between electronic circuit structures and informational ones, a fact that was excusable in the sixties, when microelectronics had just emerged as a term and the image that researchers had of it was far from what it is today. Nor was the meaning of information, as a symbolic structure of what acts, sufficiently clarified. Information must be seen as a distinct entity, apart from data (numerical or non-numerical) in a computing structure, it is a fundamental ingredient that allows a new approach to electronics in a truly functional perspective. This new approach is proposed by Mihai Drăgănescu in the eighties.

1.2 Current Acceptance of the Term Functional Electronics

In its new acceptance, functional electronics is constituted as a set of techniques that involve logical and informational structures subordinated to architectures designed in such a way as to satisfy the requirements of man and society in their effort to achieve a state of socio-human civilization. Such a vision goes beyond the purely technical approach. The world of technical objects, in alliance with the world of signs, serve man and society in a way in which structure and form are pushed into the background so that the set of functions grouped in architectures can appear in the foreground.

We could define traditional electronics as *structural electronics*, which provide the functions that technologically possible electronic structures can perform under conditions of efficient production. What

does functional electronics imply in addition or otherwise?

Functional electronics came at the end of a very long journey, traveled by man and society, during which a world of *signs* and a world of *tools* emerged and developed, which today tend to merge into a world of *intelligent tools*.

Through successive externalizations[1], man created two new worlds: the world of signs and the world of tools. We could speak of the world of signs as an ethnosphere in which, in addition to the species, essential information for the existence of man and society is stored and memorized. Similarly, we can use the term technosphere for the world of tools.

The chance to be and to preserve freedom, therefore the meaning of existence, man ensured through the fusion between the technosphere and the ethnosphere, between tool and sign, between action and sign, between work and culture.

The dominant concept in the technosphere was and still is that of structure, and in the ethnosphere that of form. Technical objects were obstinately structured and the world of signs was formalized, both thus trying to solve the problems posed by complexity.

Through externalization, man avoided, in addition to excessive specialization, complexity. Until now, both the technosphere and the ethnosphere have struggled with complexity largely independently of each other. They may have a better chance if they join forces in a synthesis that would allow them to overcome structuralism and formalism through an architectural approach. The unstructured and the informal can impose themselves, at least partially, within the scope of an architecture, breaking through rigid barriers in the effort to master complexity.

Form and structure support each other in the scientific endeavor. A structure can be formally described, and a formal description is usually structured. In many situations, speaking of a structure is equivalent to speaking of a form. A structure can associate a language and, and, reciprocally, in the cutting edge of the world of tools and signs. Microelectronics and computer science seem to be the most advanced echelons of the two worlds that, at this level, are ready to merge, opening the era of tools that process signs or of signs that behave like tools.

A first step on the path to this fusion is functional electronics.

1.2.1 Function vs. Structure

The system, a very ”open” entity. How can the architectural approach surpass the formal-structural one? By the fact that it can also encompass informal functions achievable with unstructured objects.

We understand by an architecture a set of functions defined at the interface between two domains. M. Drăgănescu does not limit the functions associated with an architecture to those that can be defined rigorously formally. As a consequence, the concrete realization of these functions will not always imply structures either. Unstructured, informal objects can coexist with formally defined structures.

Structures can only be formal, similar to forms that can only be structured. Structural electronics is limited to what is formally structurable, while functional electronics is much more highly limited, the domain of informal functions being much extended beyond that of functions achievable with formal structures.

The structure supports a purely systemic approach by offering products whose ”closure” through reality is not taken as a fundamental design criterion.

[1] Term coined by André Leroi-Gourhan, [31].

1.2.2 The System, a Too "Open" Entity

The technosphere dominated by structural electronics is open to man and society, in the sense that a loop is not closed that would generate in all situations a truly valuable integration of the products of this technical field. Electronics, through its future functional character, will have to deal with what it is not, in order to ensure its closure in a context of potentiated value. How many of the products offered by traditional electronics correspond to real human or social needs? (Let's ask this question in the context in which the situation in the electronics field seems to be, in any case, among the least serious). A bad closure of a technical domain through social, human and natural can only lead to a process of continuous degradation of the environment and life, reducing the chances of access to civilization.

The coupling beyond the strictly technical domain ensures functional electronics the possibility of introducing into reality objects that tend to be, and exceed, the status of a system through the capacity to integrate, leaving it with as few open loops as possible. In the limit, a system that closes itself tends to cease to be a system. Electronics thus transcends the purely technical, or the technical, for the beginning through functional electronics, being able to attach valences that truly integrate it into the nature-man-society triad. An electronic system can have a domain of use.

An electronic function presupposes a codomain in which its utility can manifest itself. If the function tends to be bijective, the premises for closing a loop that suspends the systemic character of the electronic object are created.

1.2.3 Formal vs. Non-formal

In an architecture we can include formal and informal functions in a coherent whole, compatible with a certain context and subordinated to a certain purpose.

The system had its own function, independent of the context. Architecture is always related to something, it can never be an architecture in itself. The system focuses on its function, and architecture offers functions at the interface between two realities, which can be described formally or informally. The system is not fulfilled as a concept except in a formal framework, while architecture allows for formal and informal approaches in equal measure. When the informal should not be taken into consideration, the systemic approach proves to be sufficient. We are forced to take informality into account when we want to close at least partially the too large openings that systemicity implies.

Objects of traditional structural electronics interact with man, nature or other objects mainly through signals, which can be audiovisual, obtained from transducers or generated to trigger electromechanical actions. Functional electronics mainly involves symbolic interactions through natural language or languages with minimal syntactic restrictions; images also play an increasingly important role. Another characteristic is that the syntactic that had priority in structural electronics is increasingly replaced by semantics. The signals at the terminals of functional electronic objects become symbols with the capacity to mean, thus flooding reality with information flows that did not previously exist.

Internal symbols manipulated by an electronic tool can also acquire semantic values. When a symbolic structure acquires meaning in a digital structure, we can say that that structure has degenerated into information [16]. For reasons strictly related to the optimization of the internal structure, digital systems have evolved in such a way that symbolic structures have appeared and developed within them. At the moment when they have acquired such significance for internal functions, they have begun to seize control of the functions seen from the outside, or, in other words, they have taken over control of the actualization from the architectural perspective. The development and definition of systems from the architectural perspective is subject to strong structural restrictions at the moment when information

seizes control of the functional actualization.

At the level of microelectronics and computer science, the channel through which the world of tools communicates stimulatingly, with rapid and continuous effects, with the world of signs.

1.3 Algorithmic Complexity in Defining Functional Electronics

1.3.1 Solomonov – Kolmogorov – Chaitin: Algorithmic Information

All big ideas have many starting points. It is the case of *algorithmic information theory* too. We can emphasize three origins of this theory [8]:

- Ray Solomonoff's researches on the inference processes [41]
- Andrey Nikolaevich Kolmogorov's works on the string complexity [27]
- Gregory Chaitin's work on the length of programs that generate binary strings [7].

Solomonoff's researches

Solomonoff's on prediction theory can be presented using a short story. A physicist makes the following experience: he observes at each second a binary manifested process and records the events as a string of 0's and of 1's, thus obtaining a 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, 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 is the pattern emphasized in the recorded binary string. Thus, 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
- compressible strings in which finite substrings, the patterns, are periodically repeated, allowing a shortest description.

Kolmogorov's work

Kolmogorov 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. The first string has a well-defined generation rule and the second seems to be randomly generated. We need, according to Kolmogorov, additional concepts 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 1.1 *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) 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.

Kolmogorov proved that always there exists a function that generates the *shortest* description for obtaining the string x starting from the string y.

Chaitin's approach

The teeneager Chaitin used a *Universal Turing Machine*, M, instead of the function f. He was preoccupied to study the minimum length of the programs that generate binary strings.

Definition 1.2 *Chaitin's complexity of the string x as follows:*

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

where p is the shortest program *of length $|p|$ that generate on the machine M the string x starting with an* empty string *on its tape.* $\diamond$

Chaitin defines the basic concepts of algorithmic information theory, as follows [**?**].

Definition 1.3 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 p results by a separate toss of an unbiased coin.* $\diamond$

Definition 1.4 *The* algorithmic entropy *of the binary string* s *is* $H(s) = -log_2 P(s)$. $\diamond$

Definition 1.5 *The* algorithmic information *of the string* s *is* $I(s) = min(H(s))$*, i.e. the shortest program written for the best machine M.* $\diamond$

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.

Theorem 1.1 *The minimal algorithmic entropy for a certain n-bit string is in* $O(log\, n)$. $\diamond$

Therefore, 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.

Theorem 1.2 *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.* $\diamond$

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. To widen the scope of the formal approach, we need to "filter out". in the direct representations of reality, insignificant nuances. It increases so that the domain of the real in which formalisms can be applied.

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, as well as the rules of inference. Therefore, the complexity of a theory is the complexity of the string that contains its formal description.

Theorem 1.3 *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 "*$H(specific\ string) \geq n$*".*" $\diamond$

Consequences

Many aspects of the reality can be encoded in finite binary strings with more or less accuracy. As most of these strings are random, our capacity to provide *strict rigorously* forms for all the processes in reality is practically null. Indeed, 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 sometimes a nonsense.

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

1.3.2 Size vs. Complexity of Digital Systems

The huge *size* of the actual circuits implemented on a single chip imposes a more precise distinction between *simple* circuits and *complex* circuits. When we can integrated on a single chip more than 10^9 components, the size of circuits becomes less important than their complexity. Unfortunately we don't make a clear distinction between *size* and *complexity*. We say usually: "the complexity of a computation is given by the size of memory and by the CPU time". But, if we have to design a circuit of 100 million transistors it is very important to distinguish between a circuit having an uniform structure and a patternless – sometimes we say randomly – structured ones. In the first case the circuit can be easy specified, easy described in a Hardware description Language (HDL), easy tested and so on. Otherwise, if the structure is completely random, without any repetitive substructure inside, it can be described using only a description having a similar dimension, expressed in number of line code, with the circuit's size. When the circuit is small, it is not a problem, but for million of components the problem has no solution. Therefore, if the circuit is very big, it is not enough to deal only with its size, the most important becomes also the degree of uniformity of the circuit. This degree of uniformity – the degree of order inside the circuit – can be specified by its **complexity**.

As a consequence we must distinguish more carefully the concept of *size* by the concept of *complexity*. Follow the definitions of these terms with the meanings we will use in this book.

Definition 1.6 *The* **size** *of a digital circuit, $S_{digital\ circuit}$, is given by the dimension of the physical resources used to implement it.*

◇

In order to provide a numerical expression for size we need a more detailed definition which takes into account technological aspects. In the '40s we counted electronic bulbs, in the '50s we counted transistors, in the '60s we counted SSI[2] and MSI[3] packages. In the '70s we started to use two measures: sometimes the number of transistors or the number of 2-input gates on the silicon die and other times the silicon die area. Thus, we propose two numerical measures for the size.

Definition 1.7 *The* **gate size** *of a digital circuit, $GS_{digital\ circuit}$, is expressed as the total number of CMOS pairs of transistors used for building it*[4].

◇

This definition of size offers an almost accurate image about the silicon area used to implement the circuit, but the effects of lay-out, of fan-out and of speed are not catched by this definition.

Definition 1.8 *The* **area size** *of a digital circuit, $AS_{digital\ circuit}$, is given by the dimension of the area, expressed in mm^2, on the silicon area used to implement it.*

◇

The area size is useful to compute the price of the implementation because when a circuit is produced we pay for the number of wafers. If the circuit has a big area, the number of the circuits per wafer is small and the yield is low[5].

[2] *Small Size Integrated* circuits

[3] *Medium Size Integrated* circuits

[4] Sometimes gate size is expressed in the total number of 2-input gates necessary to implement the circuit. We prefer to count CMOS pairs of transistors (almost identical with the number of inputs) instead of equivalent 2-input gates because is simplest. Anyway, both ways are partially inaccurate because, for various reasons, the transistors used in implementing a gate have different areas.

[5] The same number of errors make useless a bigger area of the wafer containing large circuits.

Definition 1.9 *The* **algorithmic complexity** *of a digital circuit, simply the* **complexity**, $C_{digital\ circuit}$, *has the magnitude order given by the minimal number of symbols needed to express its definition.*

◇

Definition 1.9, inspired by Gregory Chaitin's definition for the algorithmic complexity of a string of symbols is a concept able to clarify, in the context of Moor's Law exponential growing mechanism, the necessary distinction between size and complexity of an entity, more specific an integrated electronic system. The confusion between size and complexity of digital systems can be removed using the concept of algorithmic complexity based on *Solomonoff-Kolmogorov-Chaitin complexity* [41] [27] [7] [8]. The *algorithmic complexity* of a string is related to the dimension of the smallest program that generates it. Our $C_{digital\ circuit}$ can be associated to the shortest unambiguous circuit description in a certain HDL (in the most of cases it is about a behavioral description).

Definition 1.10 *A* **simple circuit** *is an n-input circuit having the complexity in* $O(1)$*, much smaller than its size in* $O(f(n))$*, so as*

$$C_{simple\ circuit} << GS_{simple\ circuit}$$

for $n > n_0$.

◇

Definition 1.11 *A* **complex circuit** *is a circuit having the complexity in the same magnitude order with its size:*

$$C_{complex\ circuit} \sim S_{complex\ circuit}$$

◇

Example 1.1 *The following Verilog program describes a* complex circuit*, because the size of its definition (the program) is*

$$S_{def.\ of\ random\_circ} = k_1 + k_2 \times S_{random\_circ} \in O(S_{random\_circ}).$$

```
/*****************************************************
File name: randomCirc.sv
    The description of a complex circuit. It describes a
    4-gate circuit in (4 + const) lines of code
*****************************************************/
module randomCirc(  output  logic f, g,
                    input   logic a, b, c, d, e);
   logic w1, w2;

   and  and1(w1, a, b),
        and2(w2, c, d);
   or   or1(f, w1, c),
        or2(g, e, w2);
endmodule
```

◇

Example 1.2 *The following Verilog program describes a* simple circuit*, because the program that define completely the circuit is the same for any value of* n*.*

```
/*************************************************************
File name: orPrefixes.sv
    The description of a (big and) simple circuit. Its size
    is in O(n), while de number of lines of code is constant
*************************************************************/
 module orPrefixes #(parameter n = 256)
            (output logic [0:n-1] out,
             input  logic [0:n-1] in );

    integer k;
    always_comb
        begin out[0] = in[0];
            for (k=1; k<n; k=k+1) out[k] = in[k] | out[k-1];
        end
 endmodule
```

The prefixes of OR *circuit consists in* n OR_2 *gates connected in a very regular form. The definition is the same for any value of* n[6].

◇

Composing circuits generate not only biggest structures, but also *deepest* ones. The depth of the circuit is related with the associated propagation time.

Definition 1.12 *The* depth *of a combinational circuit is equal with the total number of serially connected* constant *input gates (usually 2-input gates) on the longest path from inputs to the outputs of the circuit.*

◇

The previous definition offers also only an approximate image about the propagation time through a combinational circuit. Inspecting the parameters of the gates from a *Standard cell libraries* you will see more complex dependence contributing to the delay introduced by a certain circuit. Also, the contribution of the interconnecting wires must be considered when the actual propagation time in a combinational circuit is evaluated.

Some digital functions can be described starting from the associated *elementary circuit* by adding a *recursive rule* for building a circuit that executes the same function for any size of the input. For the rest of the circuits, which don't have such type of definitions, we must use a definition that describes in detail the entire circuit no matter how big it is. This description will be non-recursive and thus *complex*, because the resulting dimension is proportional with the size of circuit (each part of the circuit must be explicitly specified in this kind of definition). We will call a complex circuit *random circuit*, because there is no (simple) rule for describing it.

The first type of circuits, having recursive definitions, are *simple* circuits. Indeed, the elementary circuit has a constant (usually small) size and the recursive rule can be expressed using a constant number

[6]A short discussion occurs when the dimension of the input is specified. To be extremely rigorous, the parameter n is expressed using a string o symbols in $O(\log n)$. But usually this aspect can be ignored.

of signs (symbolic expressions or drawings). Therefore, the dimension of the definition remains constant, independent by *n*, for this kind of circuits. In this book, this distinction, between simple and complex, will be exemplified and will be used to promote useful distinctions between different solutions.

At the current technological level the size becomes less important than the complexity, because we can *produce* circuits having an increasing number of components, but we can *describe* only circuits having the range of complexity limited by our mental capacity to deal efficiently with complex representations. The first step to provide a circuit is to express what it must do in a behavioral description written in a certain HDL. If this "definition" is too large, having the magnitude order of a huge multi-billion-transistor circuit, we don't have the possibility to write the program expressing our desire.

In the domain of circuit design we passed long ago beyond the stage of *minimizing* the number of gates in a few gates circuit. Now, the most important thing, in the multi-billion-transistor circuit era, is the *ability to describe*, by recursive definitions, simple (because we can't write huge programs), big (because we can produce more circuits on the same area) sized circuits. We must take into consideration that the Moore's Law applies to size not to complexity.

Conjecture 1.1 *As the size of a circuit increases exponentially, its complexity is limited to a logarithmic increase.*

◇

Corollary 1.1 *If the size of a digital system increases exponentially and its complexity is limited to increase logarithmic, then the system must be programmable to justify increasing its size.*

◇

Because the dynamics of complexity cannot keep pace with the dynamics of size, we will have to consider the functional dynamics offered by the programmability of the structure deployed on an integrated circuit. Functional complexity will not be able to increase significantly unless the circuit function can be specified by a program running on a much simpler circuit structure (memories tightly interleaved with execution units). Thus, the incompressible bit structure of programs will give the functional complexity [42].

1.3.3 Intensity vs. Complexity of Computation

Just as we made a clear distinction between complex circuits and simple circuits (see section 1.3.2), we will make, also based on the definition of algorithmic complexity [7] [27] [41], an equally clear distinction between complex computation and intensive computation (the simply expressed one).

Definition 1.13 *Complex computation is characterized by an execution time, expressed in clock cycles, in the same range than the code size, expressed in number of symbols, which describes it.* ◇

Example 1.3 *Solving the 2nd degree equation is done running this fragment of* **complex** *code:*

```
a = float(input('␣coeficientul␣a␣este:'))
b = float(input('␣coeficientul␣b␣este:'))
c = float(input('␣coeficientul␣c␣este:'))
gama = 2*a
eta = -b/gama
```

```
    delta = cmath.sqrt(b*b - 4*a*c)/gama
    x_1 = eta - delta
    x_2 = eta + delta
    print('Solutia X1 este:', x_1)
    print('Solutia X2 este:', x_2)
```

The execution time is proportional to the number of lines justifying the characterization of a complex program.

◇

Definition 1.14 *Intense computation is characterized by an execution time, expressed in clock cycles, much bigger than the code size, expressed in number of symbols, which describes it. Or: the computation is intense if the execution time is in $O(f(n))$ and the code size is in $O(1)$.*

◇

Example 1.4 *Matrix multiplication is an example of intense computation.*

```
def multiply_matrix(A,B):
  global C
  if  A.shape[1] == B.shape[0]:
    C = np.zeros((A.shape[0],B.shape[1]),dtype = int)
    for row in range(rows):
        for col in range(cols):
            for elt in range(len(B)):
              C[row, col] += A[row, elt] * B[elt, col]
    return C
  else:
    return "Sorry, cannot multiply A and B."
```

The running time for this code depends on the size of the two arrays which can be arbitrarily large.

◇

The uniformity of the computational patterns assumed by the intensive calculation presupposes the uniformity of the structural patterns of some parallel structures. If a mono-core structure lends itself competently for both complex and intensive computing, effectively only many-core structures, of the type presented in the two previous chapters, can perform intensive computing.

If we can afford to continue focusing on the Turing computation for complex computation, for the intensive one a parallel structure inspired by the Kleene computation is recommended. Consequently, an efficient structure will have to be thought of as containing two structures, a HOST for intensive computation, which includes the control of the entire system, and one for intensive computation that runs under the command of the HOST. In this way, a heterogeneous system is defined that uses two distinct computational paradigms to obtain the efficient adequacy to the complex-intensive distinction in computation.

1.4 Gordon Moore's Law and Complexity

Gordon Moore talks twice about the evolution of integrated circuit technology. In addition to the internal report from 1965 [33], the paper from 1975 [34] provides important new insights. If the first intervention is mainly about the density of components and the efficiency with which the integrated circuits were produced, in the 1975 paper are explicitly added two other dimensions for evaluating the evolution of the domain of integrated circuits. Besides the density of components on the silicon chips, the chip size an "circuit cleverness" are considered. Besides, density & size given by technology, organization & architecture provide functional improvements, "cleverness" in Moore terms.

The way in which the "circuit cleverness" and die size evolves is dictated equally by the organization and architecture of the system that grows exponentially in size. Both organization and architecture must allow the acquisition of functional facilities to justify the increase of both chip size and component density. How can the exponential increase of the number of components per chip be capitalized in the conditions in which the accepted increase of the complexity of the structure is much slower (according to our point of view it is logarithmic takeing into account Conjecture 1.1, and its corollary)? Only a programmable cellular structure in which the data is interleaved with the execution elements represents the solution.

1.4.1 How Modularity Supports Parallelism

Conjecture 1.1, which works intuitively, is supported by a cellular increase of the structure with identical cells having a controllable complexity. The uniformity of the cellular network keeps the exponentially growing size at a constant complexity. Only the cells are supposed to be complex, but they are not submitted to Gordon Moore's Law.

In this way, it is understandable how we maintain complexity at a controllable growth level. But, the problem that arises is how the structural resources of parallelism thus obtained can be put to the service of computation. Two ways are envisaged:

1. using programming techniques and algorithmic thinking that adapt the geometric uniformity of the cellular network to the requirements of the computation to be performed

2. organizing the cellular structure so that it corresponds to a parallel mathematical model of computability that allows algorithmic thinking and coding in a friendly and efficient programming environment (from the point of view of performance, energy and silicon area (price)).

In the first case, adapting to the computational requirements faces difficulties in achieving efficiency, both in programming and in use. In the second case, convergence between structural possibilities and computational requirements can be ensured so that programmability and efficiency in use can be optimized.

The two ways of exploiting parallel hardware resources are practiced with different weights in the corporate and academic space. The first method is used with predilection in the corporate space while the second is practiced more in the academic one.

From a theoretical point of view, the adaptation of the structural potential to the functionality pursued is illustrated by many concerns among which we quote:

- the *functional forms* of John Backus [3] which presents at a high theoretical level the idea of programmability independent of the hardware support, in a manner that highlights structural requirements in a way that very naturally introduces the concept of parallelism (see section 11 of the cited text)

- *"The Landscape of Parallel Computing Research: A View from Berkeley"* [1, 2] highlights the functional areas that parallel computing must cover, such as dense linear algebra, sparse linear algebra, ...
- patterns for efficient computation emphasized in [32], a book that studies parallel computing from the perspective of computational patterns that hardware systems must support efficiently

1.4.2 Heterogeneous Computation

In any real application we are challenged with a mixture of complex computation and intense computation. In order to optimize the computational system it is imposed the segregation between the complex and intense using a ***heterogeneous computing system*** whose block schematic is represented in Figure 1.1, where HOST COMPUTER is responsible for control and complex part of the program, while ACCELERATOR runs the intense part of the application.

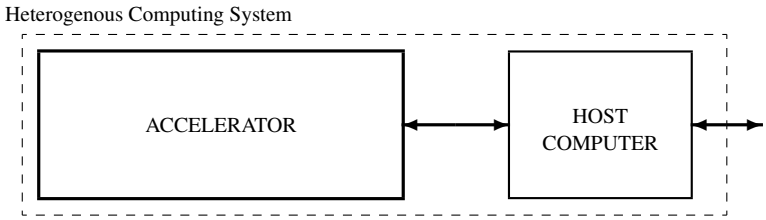

Figure 1.1: Heterogenous computing system.

The host is a Turing-based computer (Processor + Memories + I/Os), while the accelerator is many-cell Kleene-based engine. The modular structure of the accelerator allows the exponential increase of the size, while the complexity of the computation is based on the programmability. The overall programming environment of a heterogeneous system is organized on two levels:

1. at the host computer level, a standard or an application specific library is implemented in a high-level language using the associated kernel-library developed at the accelerator's level
2. on the accelerator level, the kernel-library, developed in assembly language for data structures limited to the size of the accelerator, expressed by p, number of cells, and m, the size of memory in each cell.

This approach avoids the development of a specific language for the lowest level of the hierarchy, while for the host level there are high-level languages. The kernel-library running on the accelerator works at the host's level like an extension of its instruction set architecture, ISA. This approach, which focuses mainly on a hierarchical function architecture, is also encouraged by the group proposing *The Berkeley view of the parallel computing landscape*:

> *The software span connecting applications to hardware relies more on parallel software architectures than on parallel programming languages. Instead of traditional optimizing compilers, we depend on autotuners, using a combination of empirical search and performance modelling to create highly optimized libraries tailored to specific machines.* [2]

The two-level hierarchical hardware organization supposes a hierarchy of architectures through which each level is integrated into the next higher level through a set of functions structured as a function library. Let us take the example of a linear algebra oriented library. The accelerator, programmed in assembly, implements a kernel-library for functions defined on arrays with sizes limited by number of cells in accelerator. Then, the host, programmed in a high-level language, manages the linear algebra algorithms for data of any size by tiling problems to be solved by the kernel-library running on the accelerator.

The structural hierarchy, accompanied by the architectural one, facilitates distinct levels at which programming is practiced. In this sense, we continue the previous quote:

> *By splitting the software stack into a productivity layer and an efficiency layer and targeting them at domain experts and programming experts respectively, we hope to bring parallel computing to all programmers while keeping domain experts productive and allowing expert programmers to achieve maximum efficiency.* [2]

Instead of optimizing compilers, we must optimize kernel-libraries for parameterizable and configurable accelerators. In this early stage of developing parallel accelerators, auto-tuning mechanisms deserve much attention for the process leading us toward optimal structures and optimally defined and implemented kernels of functions.

1.4.3 Configurability

Heterogeneous computing has two extreme implementation approaches:

1. accelerators based on parallel computing: the cellular structure is configured as a programmable parallel computing machine instead of the clock speed increase (the clock speed race stopped around 2002, see Figure 1.2

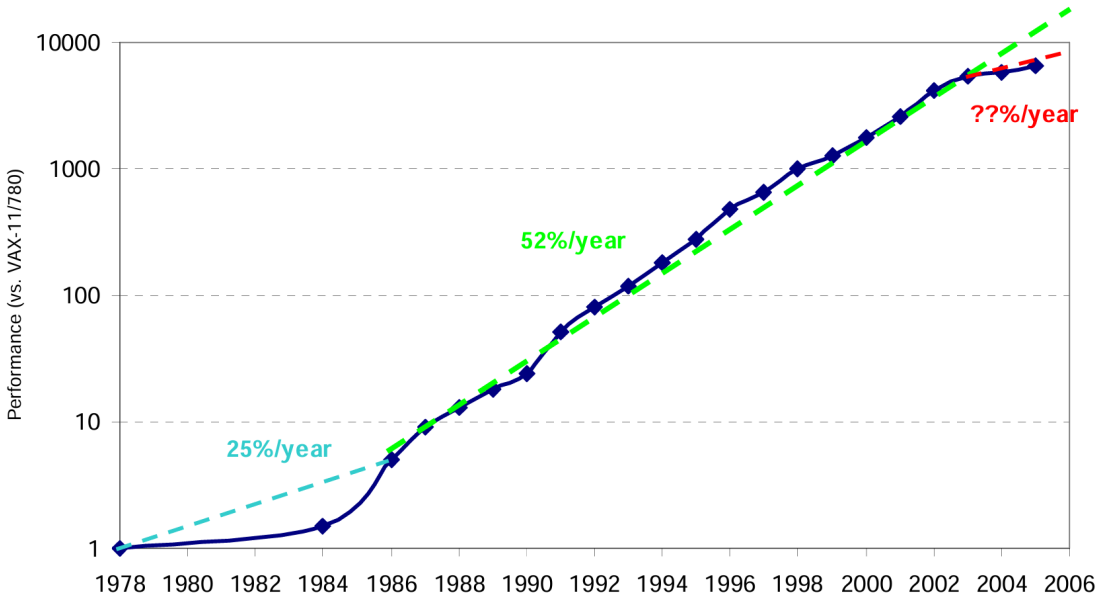

Figure 1.2: Processor performance improvement between 1978 and 2006 [1].

2. accelerators based on ***reconfigurability*** using FPGA-based techniques, in two versions:

   (a) user-made design using HDL which ***configures*** the accelerator as a circuit that performs a specific function very efficiently

   (b) synthesis techniques based on high-level synthesis, HLS, which starts from the description of functions in the high-level language in which the program is written for the host and dynamically generates circuit structures which, implemented in the accelerator's FPGA, increase processing performance

If until around the year 2000 we relied on increasing the system clock frequency (in the age of clock speed), starting with the 3rd millennium we are forced to rely on the possibilities offered by parallelism (in the age of parallelism) to increase the performance of complex computation execution (see Figures 1.3).

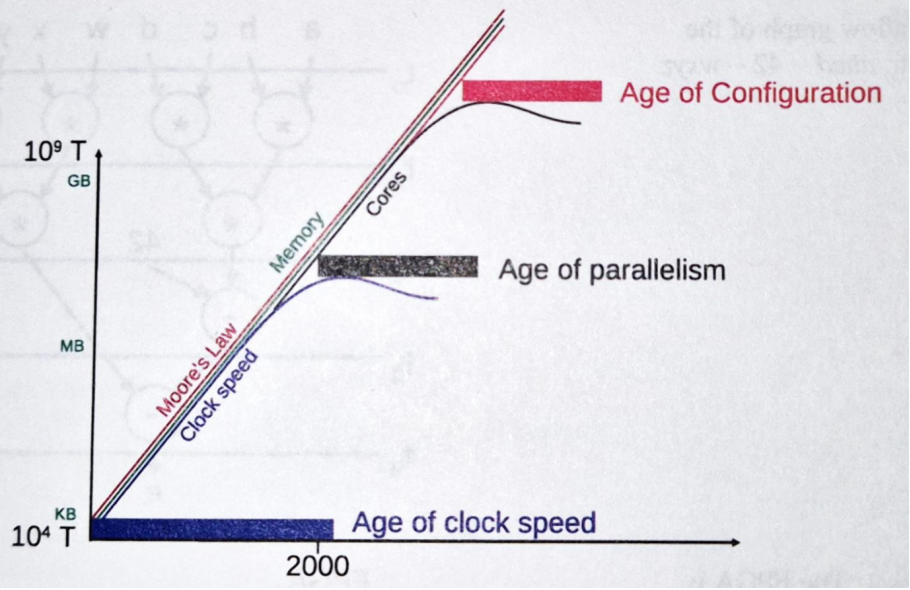

Figure 1.3: From age of clock speed to the age of configuration through the age of parallelism [25].

The age of configurability is gradually starting to manifest itself. Indeed, with hundreds of billions of components on silicon, very intensive and complex functions at the same time will be able to be implemented using HDL techniques. What we do not know today is what will be the performances and efficiency with which they will be obtained both from the point of view of energy, area but also of a friendly development environment.

1.4.4 Pseudo-Reconfigurability

In the prediction presented in Figure A.1, limiting the number of cores does not mean giving up modularity. Modularity will not only refer to a repetitive pattern, it will be associated with complex network configurations (hierarchical, almost certainly) but configured to a controllable complexity. Hierarchy, as a superior form of modularity, will allow functional realizations in a context that will exceed the possibilities of current programming languages.

Configurability can manifest itself above a certain level only against the background of some configurational "matrices" that will allow reaching complexities achieved through self-organization mechanisms based on higher-level autonomies allowed by higher-level loops. We can speak of forms of pseudo-reconfigurability.

One of the first and simplest forms of pseudo-reconfigurableness can be represented by a parallel programmable system that is ***parameterizable and configurable through HLS-type mechanisms***. An HLS design system based on an efficient parallel structure that can be optimized through parameterization and configuration will allow for efficiency in a friendly programming environment.

Configurability starts from the circuit, while pseudo-configurability starts from a parametrizable and configurable parallel structure. How can the difference in flexibility between the two solutions be reduced? The circuit is more malleable than a parallel structure, so it can be adapted more efficiently to perform a certain function.

It is worth looking for a solution because:

1. the functions we want to accelerate are part of well-defined conceptual classes, which allows the consideration of solutions subject to specific configurations
2. the parallel structure can be configured according to a computational model specific to parallelism, thus allowing an adaptation that matches the computational performance of the circuit.

Pseudo-configurability represents a realistic solution for the era of configurability because it can represent a compromise between the huge size and the complexity we want to achieve. Pseudo-configurability leaves room for ***information*** to act to provide the functional diversity we want to achieve as the size of physical structures increases.

1.5 Data vs. Information

We *know* about some things, we are *informed* about some events, we have all the *data* related to a certain reality. So there are three distinct ways in which we relate to those around us. *Knowledge – Information – Data* is a challenging and sometimes a confusing triad. The middle term seems to be the hardest to catch in a unanimously accepted definition. Dictionaries agree relatively easily on *knowledge* and *data*, but it is very difficult to identify the slightest attempt to provide an acceptable definition for the concept of information. Any attempt slips very quickly into quantitative assessments of an entity considered, by an unconfessed guilty consensus, as known. We do not ignore some successful attempts to provide qualitative interpretations of the concept of information, but we consider it necessary to deepen these encouraging beginnings to the level at which we can provide an effective qualitative and quantitative theory. Our approach is oriented in this direction.

1.5.1 Mihai Drăgănescu's General Information

Mihai Drăgănescu in [17] outlines a general theory that provides a theoretical framework for the concept of information.

Definition 1.15 *The* generalized information *is:* $N =< S, \mathscr{M} >$ *where: S is the set of objects characterized by a syntactical relation, and* $\mathscr{M}$ *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 1.16 *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.

The first actual information is the semantic information.

Definition 1.17 *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 signification. There are two types of significations:

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

thus, we can write: $\mathbf{S} =< R, C >$.

Definition 1.18 *Let us call the* reference information*:* $N_{11} =< S, R >$. $\diamond$

Definition 1.19 *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 1.20 *Let us call the* phenomenological *information:* $N_2 =< S, \sigma >$*, where:* σ *are senses.* $\diamond$

Attention! The entity σ is not a set.

Definition 1.21 *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 real 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.

1.5.2 The Meaning Acting in Context

Let us expand the generalized information defined by Mihai Drăgănescu by adding a new component and its relation with the meaning, $\mathscr{M}$, associated to *S* in Definition 1.15.

Definition 1.22 *The* generalized information *is:* $N =< S, \mathscr{C}\mathscr{M} >$ *where: S is the set of objects characterized by an internal syntactical order, and* $\mathscr{M}$ *is the* meaning *of S which is used to* ***act*** *on the context* $\mathscr{C}$. $\diamond$

We consider that information must be defined in the context where it acts. In this way we differentiate information from data. Data is a passive entity while information is an active one. In the following subsections we will exemplify the information in computers, in nature and, only speculatively, in existence.

A notable distinction appears from the beginning with the definition of the Universal Turing Machine (UTM) [45]. By defining UTM, the tape of a Turing Machine (TM), on which a string of symbols is recorded, has been split by Alan Turing in two. One portion contains the description of a particular TM, M, and the second portion represents the contents of the string, D, on which M is working. The finite automaton (FA) of UTM uses the description M to modify the content of D. Thus, we can say that, in the context of UTM, FA uses the *meanings* associated with the string M ***to act*** on the contents of the string D. Both strings, M and D, are structured according to a syntactic order, but in UTM they play completely distinct roles. M describes an action that is performed by the FA modifying the string D. M has a meaning at the UTM level, while D is a passive structure that supports the action defined by M and performed by the FA. In this sense we say that M ***acts*** on D in the ***context*** of UTM.

Consequently, the abstract computer models (von Neumann and Harvard) contain two symbolic structures in their memory/memories: ***programs*** and *data*, corresponding respectively to the M and D zones of the UTM's band.

In a computer, the *information is represented only by programs*, and the data can represent information for the computer's user in the context in which the computer is used. So, it is fair to say that a computer processes data through program's information. The computer processes data, not information. The result of the computation can be instantiated in information only at user level.

Example 1.5 *Let be the instruction format in a RISC processor:*

```
instruction ::= {function, result, leftOperand, rightOperand}
```

Then the instruction stored in the program memory of the computer at the address `1324`*:*

```
programMemory[1342] = {add, reg5, reg12, reg4}
```

will act on the content of the Register File (RF) adding in register 5 the content of register 12 with the content of register 4 **no matter** *what values were stored in these registers. Thus, the content of the register file is* **passive** *data while the instruction stored in the program memory represent the* **acting** *information.*

In this example the context, $\mathscr{C}$*, is provided by an Arithmetic & Logic Unit (ALU) loop connected with a RF and the content of RF. The instruction selects with its four fields the operation performed by ALU, the destination register, the two operands in RF. The context,* $\mathscr{C}$*, is designed and filled up with data in concordance with the meaning associated to the symbols used in the instruction's fields. Outside of this context, the meaning associated with the fields of instruction cannot act in any way.*

$\diamond$

In the previous example, the values contained in registers 5, 12 and 4 could be information in the context in which the computer is used. For the arithmetic addition operation in the context provided by the considered processor these values have no meaning, but they can be, and usually are, significant in a broader context.

Example 1.6 *In the Lisp language data and programs are represented by S-expressions. Because Lisp programs are able to manipulate source code as a data structure, they are characterized by interchangeability of programs and data. The distinction between data and programs is done only in the EVAL process which consists in reducing an input S-expression to an output S-expression. In this process sometimes a big stack memory is used to deal with the recursive evaluations. During the EVAL process, which starts with an empty stack, the content of the stack is used as temporary data. At the end of EVAL the stack remains empty.*

The context, $\mathscr{C}$, in this case, consists of an EVAL engine and a big STACK, which can be called Lisp Machine *(LM). During the evaluation process we can expect a large data structure to expand in the stack memory, the data structure that is resorbed until the end of the process. Instead of the data structure contained in the RF in the previous example, the data structure in a Lisp machine has an ephemeral character. This is due to the coexistence of data and program (information) within the S-expresses that LM reduces.*

◇

The interleaving data and information in a LM is a natural step in the process of externalizing our mental abilities. We do not believe that the symbolic representations used by the human mind clearly differentiate data from information. The ability of the human mind to play different language games [**?**], simultaneously or in rapid succession, requires and makes the symbolic structures with which it operates to be able to switch their status quickly. In a certain language game a certain symbolic structure has the role of data, while in another language game it can have the role of information. We are dealing with very subtle mutual interactions whose modeling can be facilitated by the definition and use of S-expressions in LM.

Example 1.7 *Let be, in an artificial neural network, a fully connected m-input layer of n neurons. It is defined by the associated weight matrix $M_{n\times m}$. The layer of neurons is subjected to two distinct processes: training and inference. In the training process it receives at the input a series of vectors that allow the configuration of the weight matrix. In the inference process, the network receives vectors at the input, vectors that determine a certain response of the network that is according to the training to which it has been subjected.*

In the training process, $\mathscr{C}_{training}$, the content of the matrix $M_{n\times m}$ represents a data structure that is configured according to the content of the stream, S, of training vectors. In this process the input stream S represents information, because its content acts configuring the weight matrix.

In the inference process, $\mathscr{C}_{inference}$, the weight matrix is multiplied with input vectors. Now the content of the matrix $M_{n\times m}$ represents information, while the input stream of vectors represents data. The syntactic order in the matrix $M_{n\times m}$ is not obvious because it is established in a training process and is the consequence of a very subtle mechanism of identifying hidden patterns in the training stream S.

Now, the same symbolic structure, $M_{n\times m}$, play two roles depending on the context; for $\mathscr{C}_{training}$ it is data, while for $\mathscr{C}_{inference}$ it becomes information because it acts as a program established by training.

◇

While in the first example the meaning of the program is obvious because it is built applying explicit rules, in the third example the meaning associated with the content of the matrix $M_{n\times m}$ is not explicit, it exists, but in a form inaccessible to human mind. The meaning of $M_{n\times m}$ is related to the training information provided in $\mathscr{C}_{training}$ context.

1.6 Defining Functional Electronics

Functional electronics is the path proposed by Professor Mihai Drăgănescu for obtaining maximum functional complexity on a structural support with minimum complexity. The solution involved the interaction of circuit structures with information structures on the same support.

In 2003 I have participated in San Jose at the annual conference on embedded systems[7]. After three days, during which the latest solutions were presented, through which a chip can perform the most sophisticated functions required by the electronic products market, I suddenly realized that I was at a conference on *functional electronics*. A quarter of a century ago, in 1978, Professor Mihai Drăgănescu initiated the course on functional electronics at the Faculty of Electronics and Telecommunications of the Bucharest Polytechnic Institute. And even today there are still some who ask me, not without a trace of malice, what are you doing there in the course on functional electronics? I can answer more clearly than ever: *"Embedded systems, sir! Now that's how I call it. Yes, those who do embedded systems today can say, like Monsieur Jourdain, that they do functional electronics without knowing it."*

Indeed, the American term for what we consider functional electronics is *embedded systems*.

Systems in which computing is **embedded** can achieve very high behavioral complexities due to the particularly flexible way in which their **function** is implemented. Since electronic technologies allowed the integration of a processor and its associated memory on a silicon chip, we have had the possibility of integrating complex behaviors into technical objects.

The initial forms of implementing the solutions of the functional electronics have supposed one-chip micro-controllers embedded in various physical structure and programmed to interact with them in performing a useful, sometimes complex functionality. The action of Moore Law allows in the last decade to involve in the design of embedded, besides the mono-core computational engines, parallel computational structure as accelerators for critical computations. Thus, in what follows, we consider the functional electronics based on heterogeneous computing systems.

[7]he Conference of Embedded Technology. Embedded Processors Forum, San Jose, CA, 16-19 June, 2003.

Chapter 2

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.

2.1 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 2.1 *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 2.2 *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 2.3 *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 2.4 *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 2.5 *The language generated by the grammar G is the set $L(G) = \{p \mid n_0 \Rightarrow p\}$.* $\diamond$

Example 2.1 *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 2.2 *The grammar G_2 is used for generating well formed infix 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.

◇

Example 2.3 *The grammar* G_2' *is used for generating well formed postfix algebraic expressions.*

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

where:

$$P = \{S \to SM+ \mid M, M \to MF* \mid F, F \to a \mid S\}$$

◇

Example 2.4 *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 as *followed by a string of* m bs.

◇

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

Example 2.5 *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 2.6

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

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 2.7 *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:

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

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.

$\diamond$

2.2 Translation Using Generative Grammars

In computer science, the main application of the generative grammars is the translation between two formal languages, from a source language (the source code) to a destination language (usually the machine code). The translation process, in its simplest form, involves two stages:

- identification of the grammatical rules that allowed the generation of the source code starting from the start symbol in the grammar of the source language
- application of the corresponding rules in the grammar of the target language starting from the start symbol

Example 2.8 *Let us translate a sequence generated in $L(G_2)$ into the corresponding sequence in $L(G_2')$. For translation to be possible, a list of correspondences between the rules of the two languages, the source and the target, is required. This list is as follows:*

- $S \to S+M \;\leftrightarrow\; S \to SM+$
- $S \to M \;\leftrightarrow\; S \to M$
- $M \to M*F \;\leftrightarrow\; M \to MF*$

- $M \to F \;\leftrightarrow\; M \to F$
- $F \to a \;\leftrightarrow\; F \to a$
- $F \to (S) \;\leftrightarrow\; F \to S$

By applying the corresponding rules starting with S in $L(G_2')$ we get:

$S \to SM+;$	**is applied** $S \to SM+$ **corresponding to** $S \to S+M$
$SM+ \to MM+;$	*is applied* $S \to M$ *corresponding to* $S \to M$
$MM+ \to FM+;$	*is applied* $M \to F$ *corresponding to* $M \to F$
$FM+ \to aM+;$	*is applied* $F \to a$ *corresponding to* $F \to a$
$aM+ \to aMF*+;$	**is applied** $M \to MF*$ **corresponding to** $M \to M*F$
$aMF*+ \to aFF*+;$	*is applied* $M \to F$ *corresponding to* $M \to F$
$aFF*+ \to aaF*+;$	*is applied* $F \to a$ *corresponding to* $F \to a$
$aaF*+ \to aaS*+;$	**is applied** $F \to S$ **corresponding to** $F \to (S)$
$aaS*+ \to aaSM+*+;$	**is applied** $S \to SM+$ **corresponding to** $S \to S+M$
$aaSM+*+ \to aaMM+*+;$	*is applied* $S \to M$ *corresponding to* $S \to M$
$aaMM+*+ \to aaFM+*+;$	*is applied* $M \to F$ *corresponding to* $M \to F$
$aaFM+*+ \to aaaM+*+;$	*is applied* $F \to a$ *corresponding to* $F \to a$
$aaaM+*+ \to aaaF+*+;$	*is applied* $M \to F$ *corresponding to* $M \to F$
$aaaF+*+ \to aaaa+*+;$	*is applied* $F \to a$ *corresponding to* $F \to a$

◇

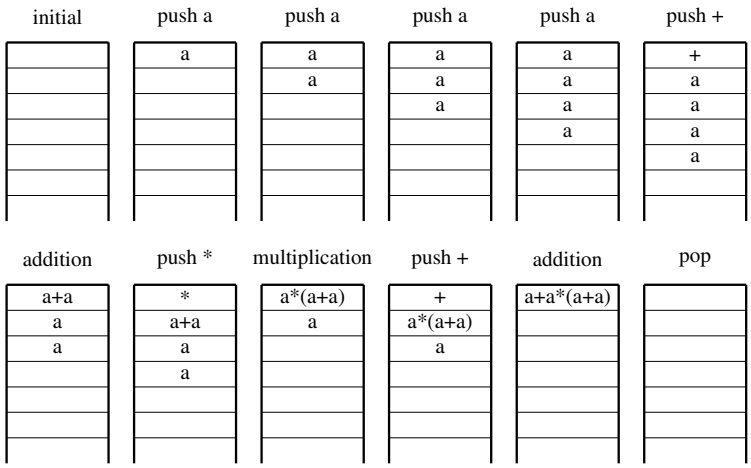

Figure 2.1: The sequence of operations for evaluation of the postfix expression (exemplified by `aaaa+*+`) using a stack-based system.

What is the advantage of using postfix expression instead of infix expression? The advantage comes from the grouping of variables separated form the operands. This grouping allow simplify the evaluation if an appropriate hardware is used. It is about using a stack-type evaluation as is illustrated in Figure 2.1. When in top of stack an operand is identified, it is popped out, is applied to the operands popped from stack and the result is pushed back in stack.

2.3 Chomsky's Hierarchy of Generative Grammars

In [9] [10] [11] Noam Chomsky, the founder of computational linguistics, introduced the concept of ***hierarchy of grammars*** based on the restrictions imposed to the restrictions imposed on the rules of generation.

Definition 2.6 *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 2.7 *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**. $\diamond$

Definition 2.8 *The language $L(G)$ is a type-i language, if the grammar G is a type-i grammar, for $i = 0,1,2,3$.* $\diamond$

Definition 2.9 *The set $\mathscr{L}_i$ is the set of type-i languages, for $i = 0,1,2,3$.* $\diamond$

Theorem 2.1 $\mathscr{L}_0 \supset \mathscr{L}_1 \supset \mathscr{L}_2 \supset \mathscr{L}_3$.

$\diamond$

Proof 2.1 *Directly, using the Definition 2.7*

$\diamond$

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 2.9 *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 3

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 3.1 *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.

3.1 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 3.2 *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 3.3 *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 3.4 *The lowest order of a system that implements any finite automaton is two.*

◇

Proof 3.1 *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 3.1, where:*

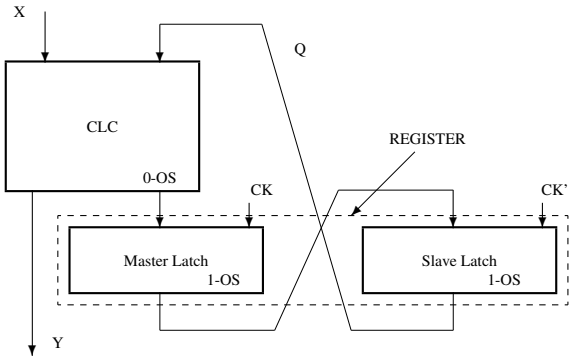

Figure 3.1: The internal structure of a Mealy automaton

- 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* **Q**. Slave Latch *has only an electrical role, allowing only the synchronous transition of the system under the control of the clock signal.*

◇

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.

3.2 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 3.2, 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.

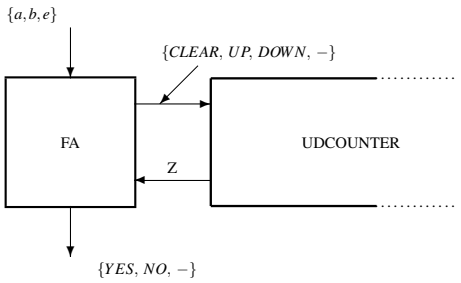

Figure 3.2: Finite automaton with counter - a 3-OS that recognizes the language $\{a^n b^n | n > 0\}$

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 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 3.5 *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 3.6 *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 3.7 *The lowest order of a system that implements a push-down automaton is 3.*

◇

Proof 3.2 *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.

3.3 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 3.1 *The* linear bounded automaton *(LBA) is a finite automaton (FA) loop connected with a linear bounded memory (see Figure 3.3) 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.

◇

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

◇

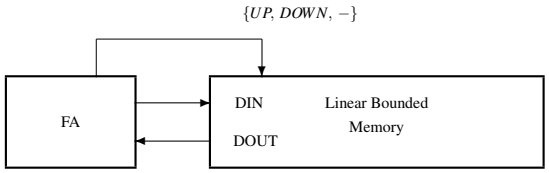

Figure 3.3: Linear Bounded Automata

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 3.9 *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 3.10 *The lowest order of a system that implements a linear bounded memory automaton is 4.*

◇

Proof 3.3 *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 3.4).

◇

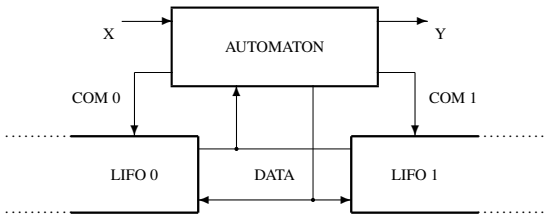

Figure 3.4: Push-down automaton with an additional stack memory

An equivalent structure for *LBA* is presented in Figure 3.5, 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

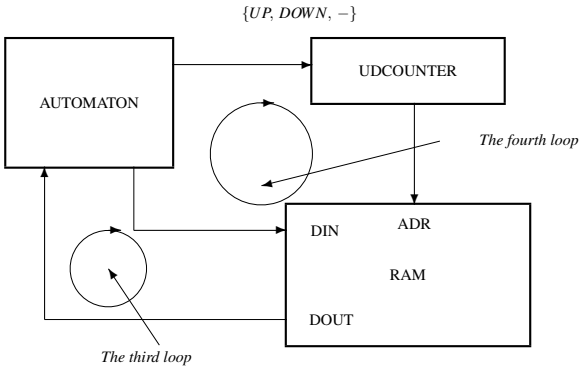

Figure 3.5: Automaton with Linear Bounded Memory

- *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.

3.4 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 3.2 *Turing Machine (TM) is a finte automaton (FA) loop connected with an infinite memory (Figure 3.6). 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.

◇

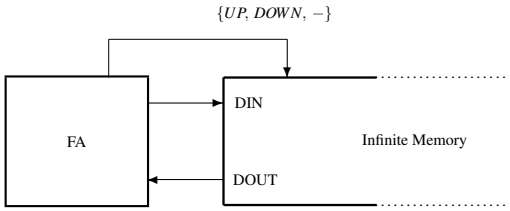

Figure 3.6: The Turing Machine

A detailed structure of TM is presented in Figure 3.7 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.

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.

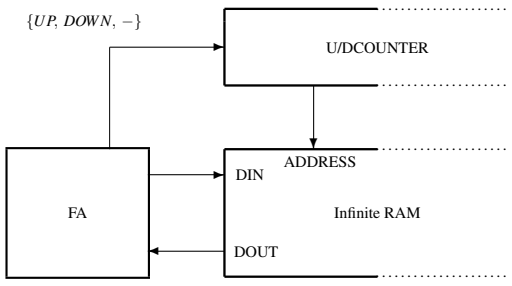

Figure 3.7: The structure of a Turing Machine

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.

3.5 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 3.3 *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.*

$\diamond$

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*.

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**!).

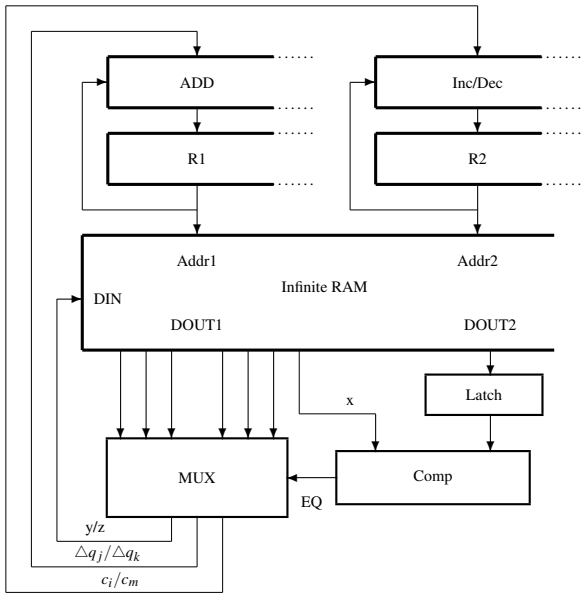

Figure 3.8: The structure of a recursive defined Universal Turing Machine

The structure of UTM in the most segregated form is presented in Figure 3.8, 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 3.8), 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 3.11 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. $\diamond$

Proof 3.4 *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.*

$\diamond$

3.5.1 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 3.12 *The function $H(< e(M), T >)$ is uncomputable.* $\diamond$

Proof 3.5 *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*.

3.6 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 4

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.

4.1 Definitions of Information

4.1.1 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.

4.1.2 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 4.1 *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. ◇

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 4.1 *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 4.2 *The machine M (see Figure 4.1) has the following characteristics:*

- *three tapes (memories) as follows:*

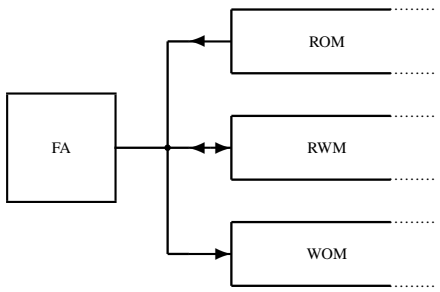

Figure 4.1: The machine M

  - *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 4.1 *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).* $\diamond$

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

Definition 4.3 *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).* $\diamond$

Example 4.2 *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 4.4 *The* algorithmic entropy *of the binary string* s *is* $H(s) = -log_2 P(s)$. $\diamond$

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

Definition 4.5 *The* bf algorithmic information *of the string* s *is* $I(s) = min(H(s))$*, i.e. the shortest program written for the best machine.* $\diamond$

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 4.3 *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 4.6 *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 4.4 *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 4.2 *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 4.7 *$H(t|s)$ 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 4.3 $H(s) \leq H(t,s) + c,\ c \in O(1)$.◇

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

Theorem 4.4 $H(s,t) = H(t,s) + c,\ c \in O(1)$. ◇

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 4.5 $H(s,t) \leq H(s)+H(t)+c$, $c \in O(1)$.$\diamond$

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

Theorem 4.6 $H(t|s) \leq H(t)+c$, $c \in O(1)$.$\diamond$

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 4.8 *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 4.7 *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 4.8 *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.

4.1.3 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 4.9 *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 4.10 *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 4.11 *The signification can be emphasized using a* formal semantical relation *in which each signifiers has one or more signifieds.* $\diamond$

Definition 4.12 *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 4.13 *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 4.14 *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 4.5 *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 4.15 *The* semantic information *is:*

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

where: S is a syntactical set, and **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 4.16 *Let us call the* reference information*: $N_{11} =< S, R >$.* $\diamond$

Definition 4.17 *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 4.18 *Let us call the* phenomenological *information:* $N_2 =< S, \sigma >$*, where:* σ *are senses.* $\diamond$

Attention! The entity σ is not a set.

Definition 4.19 *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.

4.2 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.

4.2.1 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 4.20 *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 4.21 *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 4.6 *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.

4.2.2 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 4.22 *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.$\diamond$

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.

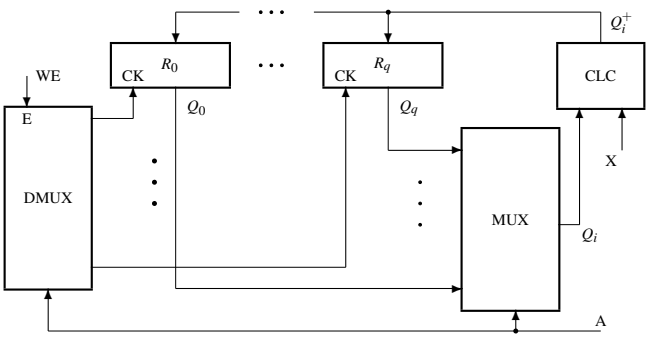

Figure 4.2: The structured state space automaton as a multiple register structure.

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 4.2) 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 4.2 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 4.3). 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.

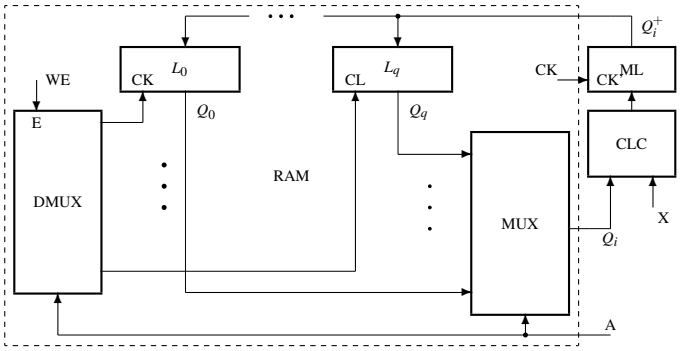

Figure 4.3: The structured state space automaton as a single master-latch.

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 4.4. 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.

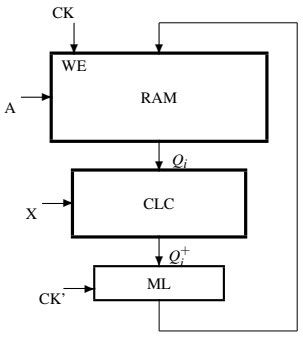

Figure 4.4: The structured state space automaton with RAM and master latch.

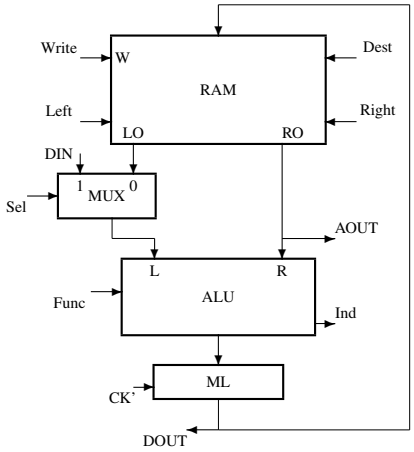

Figure 4.5: An exemplu of structured state space automaton: Registers with ALU (RALU)

Example 4.7 *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 4.5). 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 4.3 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. ◇

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.

4.2.3 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 4.23 *The* **informational structure** *is a* structured state *that has a* meaning *correlated with the* functional loop *of an automaton.*◇

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!

4.2.4 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 4.24 *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 4.6 (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*

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)*

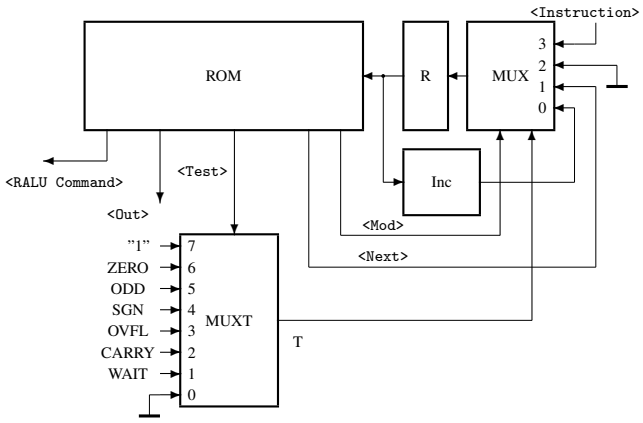

Figure 4.6: CROM

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 4.8 *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 4.25 *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.* ◇

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 4.9 *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.* ◇

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 4.26 *The* functional information *is an informational structure which generates strings of symbols specifying the actual function of a digital system.* $\diamond$

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.

4.2.5 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 4.27 *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 4.28 *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 4.29 *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 4.30 *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.

4.3 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.

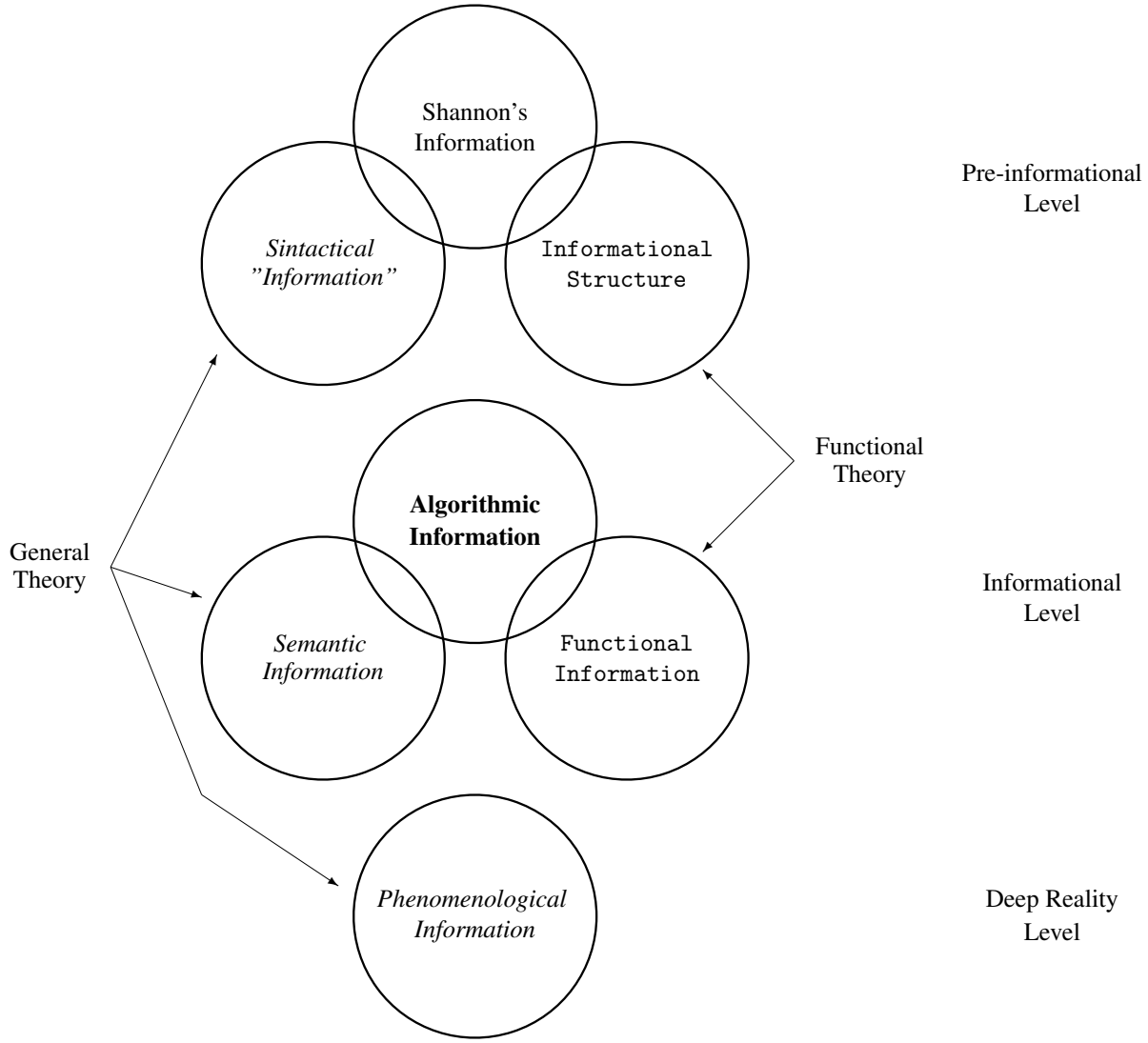

Figure 4.7: The levels of information

Part II

The Parallel Engine

Chapter 5

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

5.1 From Kleene's model to MapReduce engine

In this section the way from the Kleene's model to MapReduce engine is presented [**?**]. In Appendix B 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 B.1).

◇

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

◇

5.1.1 *Kleene Machine*: a Parallel Model of Computation

Because, according to Theorems B.1 and B.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 5.1 ***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 5.1) 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.*

◇

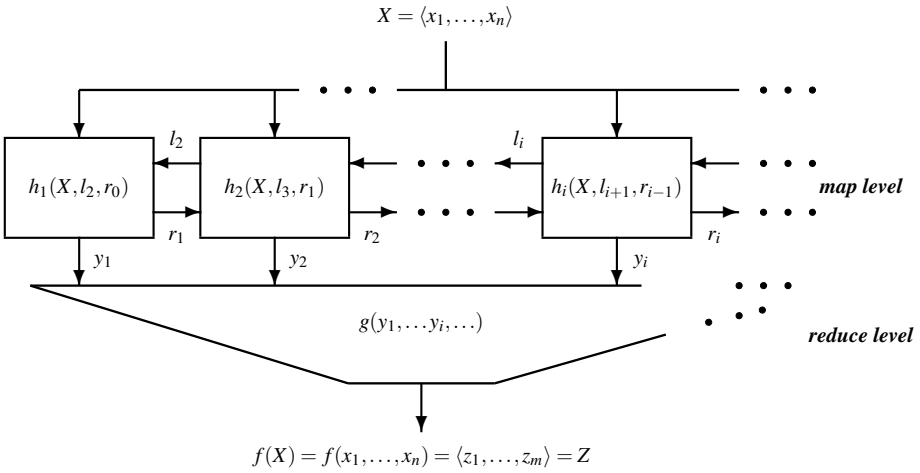

Figure 5.1: **Kleene Machine**. The ***synchronic parallelism*** is performed on the **map** level and the ***diachronic parallelism*** works between the **map** level and the **reduce** level.

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 **??**, 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 5.1 ***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.*

◇

5.1.2 *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 5.2 ***Universal Kleene Machine*** *(Figure 5.2) 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:*

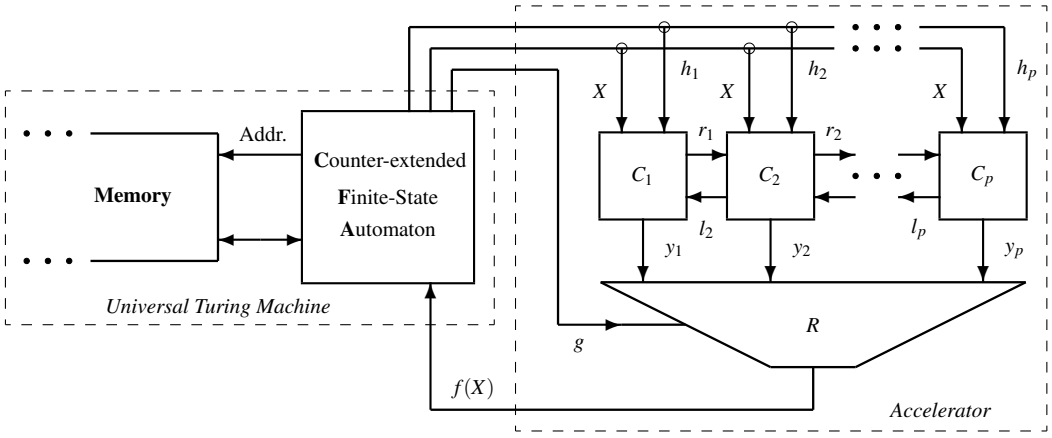

Figure 5.2: **Universal Kleene Machine**: seen as an Accelerated Universal Turing Machine.

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

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:

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

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*

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

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.

5.2 *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 5.3 *A computation model is* ***competent*** *if the computation it supports ends in a finite number of steps.*

◇

Definition 5.4 *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.

5.2.1 Forms of Parallelism

Five forms of simplified parallelism (see Figure 5.3) are emphasized as the meaningful set of particular compositions able to provide the transition from a competent model to a performant one.

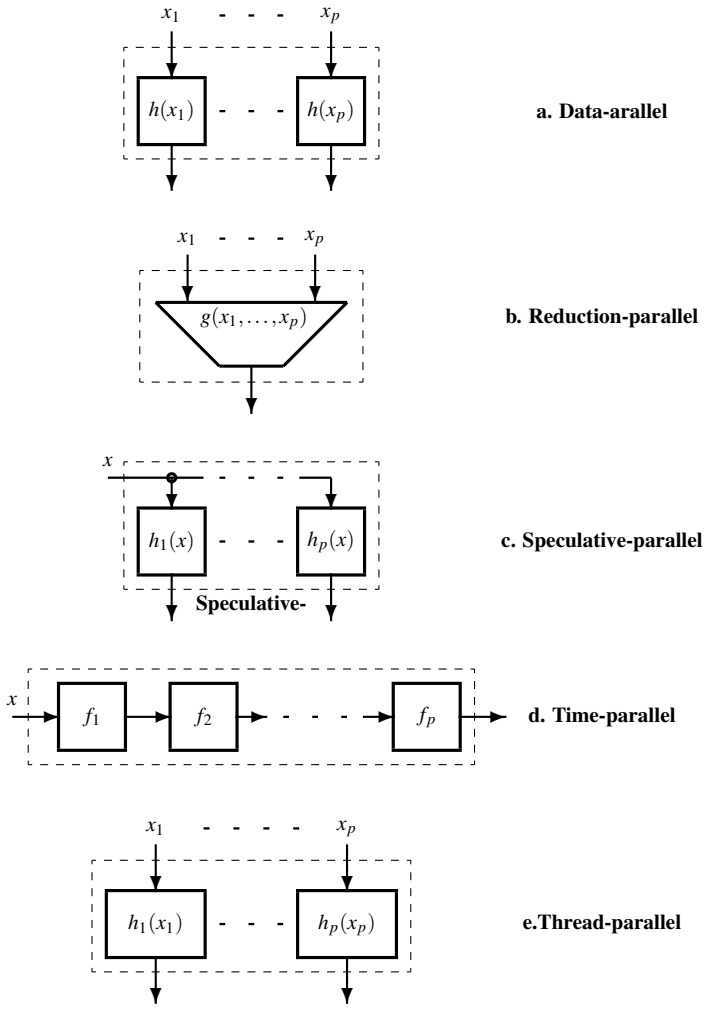

Figure 5.3: Five types of parallelism as particular forms of composition (see Figure 5.2)

Definition 5.5 Data-parallel *computation is defined for MC computation (see Definition B.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 5.6 Reduction-parallel *computation is defined for RC computation (see Definition B.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 5.7 Speculative-parallel *computation is defined for MC computation when* $n = 1$.

◇

Each function h_i has the same input variable x_1.

Definition 5.8 Thread-parallel *computation is defined for MC computation when* $n = m$ *with* $h_i(x_1,\ldots,x_n) = h_i(x_i)$, *for* $i = 1,\ldots,n$.

◇

Each cell performs a specific function on different data.

Definition 5.9 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:

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

5.2.2 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 5.4, the MapReduce recursive parallel abstract model for parallel computation is presented. It consists of:

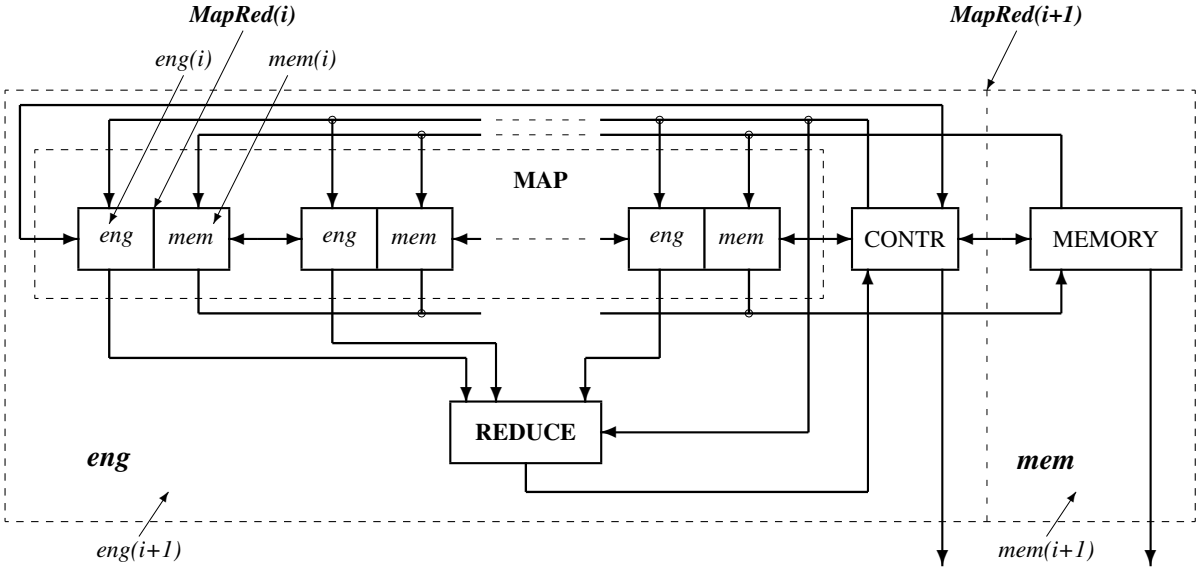

Figure 5.4: ***MapReduce recursive abstract model*** for parallel computation.

- 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
  - *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 5.4 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.

5.3 A Programming Model

5.3.1 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 [3], 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 [3], 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):

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

  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 $REV \equiv (bu\ perm < \bar{2}, \bar{1} >)$

5.3.2 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	↔	**Backus's functional forms**
data-parallel	↔	`apply to all`
reduction-parallel	↔	`insert`
speculative-parallel	↔	`construction`
time-parallel	↔	`composition`
thread-parallel	↔	`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.

5.3.3 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:

$$\textbf{Def}\, CRA \equiv (/+)\circ(\alpha((leq\circ(bu\,del1))\rightarrow(id\circ 1);\bar{0}))\circ trans$$

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`.

5.3.4 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 5.1 *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
```

◇

Chapter 6

The Generic Parallel Engine

6.1 The General Description of the Hybrid System

The structure of the hybrid system we consider (see Figure 6.1) consists of:

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

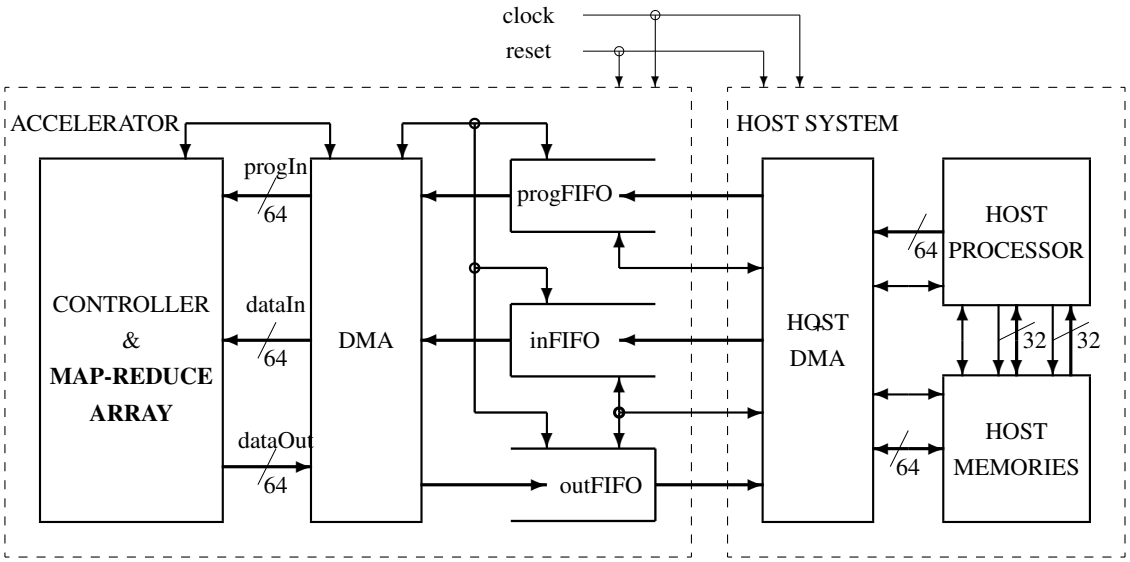

Figure 6.1: The hybrid system: HOST SYSTEM & ACCELERATOR.

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.

6.2 Host System

6.2.1 Host System's Structure

The host system (see Figure 6.1) 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.

6.2.2 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 (see `0_DEFINES.hv` in C.2.1).

6.3 Accelerator

6.3.1 Accelerator's Structure

The accelerator (see Figure 6.1) 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.

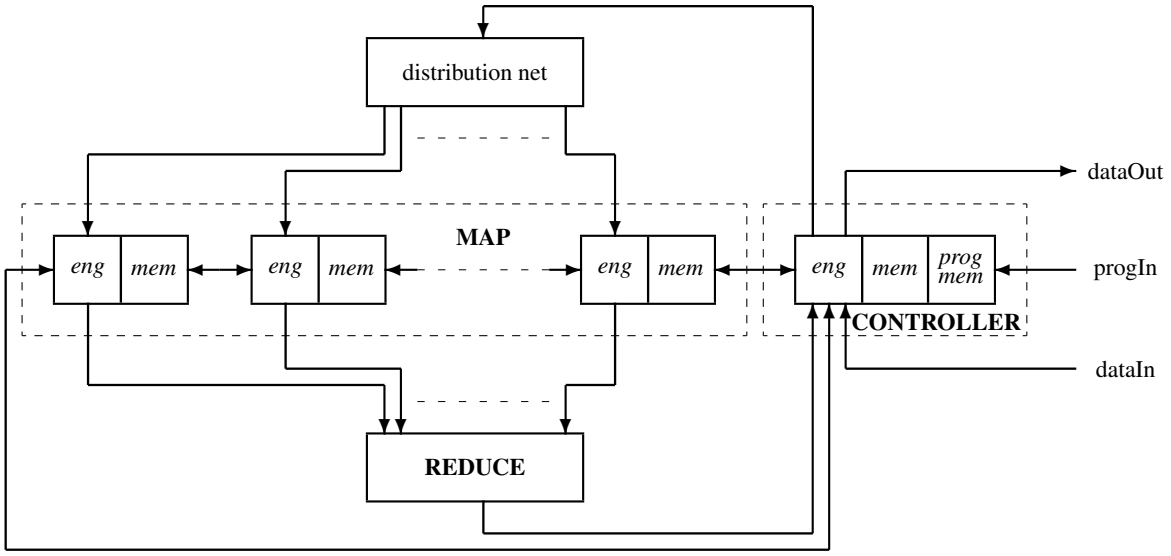

Figure 6.2: The functional organization of the accelerator's core.

The computational part of the accelerator (see Figure 6.2) 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 defined in the first three lines in `0_DEFINES.hv` (see C.2.1).

6.3.2 The User's Architectural Image

The user's image of the accelerator system is presented in Figure 6.3. 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]`
  - 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

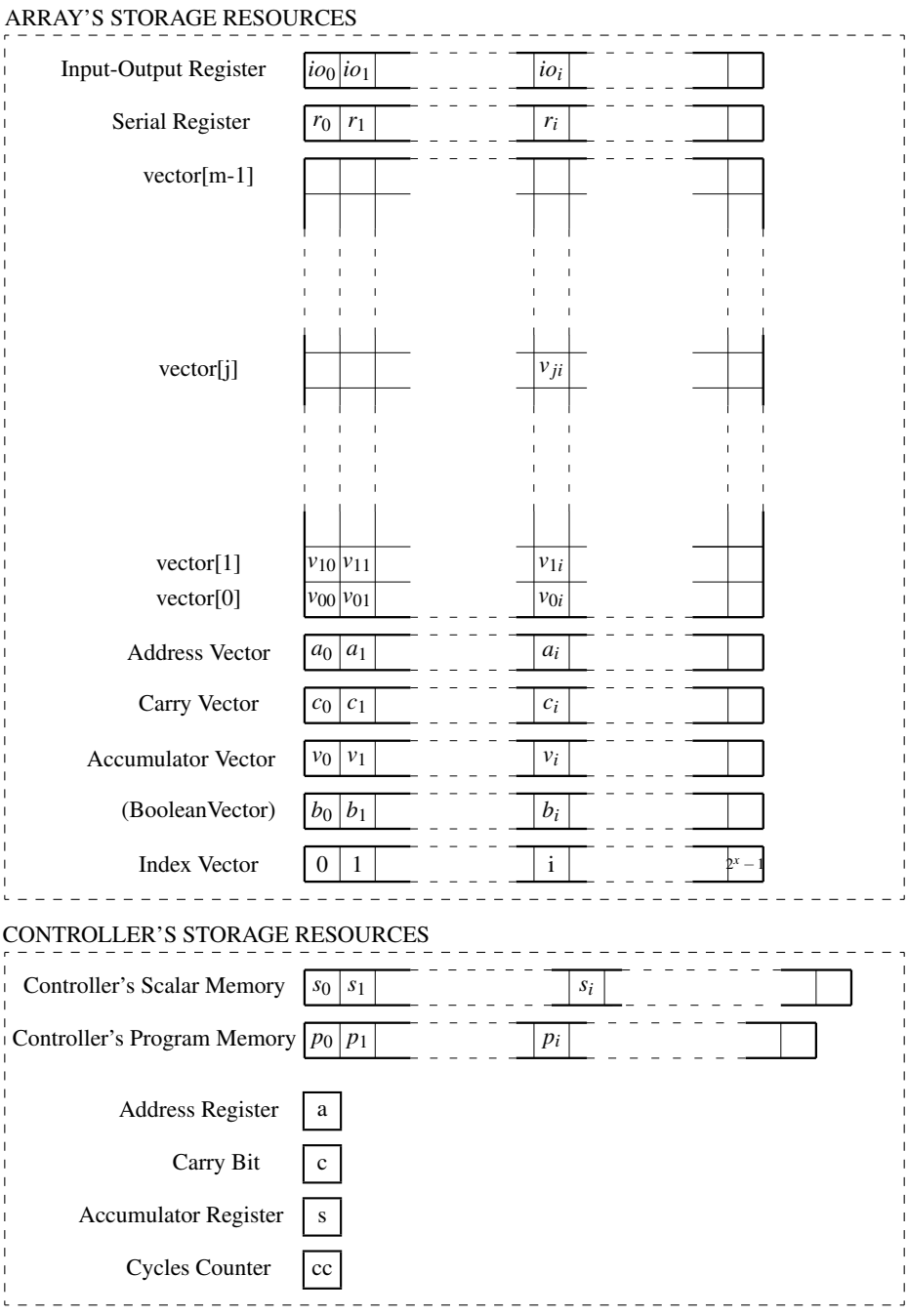

Figure 6.3: The users view of the ACCELERATOR's architecture.

    - 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 6.1) 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 $\lambda(p) = 1 + 0.5log_2 p$. 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`

6.3.3 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[63:0] = {controllerInstr, arrayInstr} =
                      {{instr[4:0], operand[2:0], value[23:0]},
                       {instr[4:0], operand[2:0], value[23:0]}}
```

where:

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

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

`value[23:0]` : 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 instructions are tabulated in Table 6.1, Table 6.2, Table 6.3, Table 6.4.

opC\op	val	imm	rel	rei	cop	cim	crl
CONTR & ARRAY	**for ARRAY the code is placed in the right column for CONTROLLER the code is placed in the left column for CONTROLLER the prefix 'c' is added**						
add	VADD(s) nop	ADD(s) pseudo-instruction codded as VADD(0)	RADD(s)	RIADD(s)	CADD	CAADD	CRADD
addc	VADDC(s)	ADDC(s)	RADDC(s)	RIADDC(s)	CADDC	CAADDC	CRADDC
sub	VSUB(s)	SUB(s)	RSUB(s)	RISUB(s)	CSUB	CASUB	CRSUB
rsub	VRSUB(s)	RSUB(s)	RRSUB(s)	RIRSUB(s)	CRSUB	CARSUB	CRRSUB
subc	VSUBC(s)	SUBC(s)	RSUBC(s)	RISUBC(s)	CSUBC	CASUBC	CRSUBC
rsubc	VRSUBC(s)	RSUBC(s)	RRSUBC(s)	RIRSUBC(s)	CRSUBC	CARSUBC	CRRSUBC
mult	VMULT(s)	MULT(s)	RMULT(s)	RIMULT(s)	CMULT	CAMULT	CRMULT
bwand	VAND(s)	AND(s)	RAND(s)	RIAND(s)	CAND	CAAND	CRAND
bwor	VOR(s)	OR(s)	ROR(s)	RIOR(s)	COR	CAOR	CROR
bwxor	VXOR(s)	XOR(s)	RXOR(s)	RIXOR(s)	CXOR	CAXOR	CRXOR
load	VLOAD(s)	LOAD(s)	RLOAD(s)	RILOAD(s)	CLOAD	CALOAD	CRLOAD
store		STORE(s)	RSTORE(s)	RISTORE(s)	CSTORE	CASTORE	CRSTORE
ARRAY							
search	VSEARCH(s)				SEARCH		
csearch	VCSEARCH(s)				CSEARCH		
insert	VINSERT(s)				INSERT		

Table 6.1: Instructions with operand from the Instruction Set Architecture.

Instructions in Table 6.1 of form `XXX(s)` are defined for a numerical value, `s`, which is a scalar `cVal` for CONTROLLER, or `aVal` for ARRAY.

opCode\operand	COMMENTARIES	ISA_ctl
CONTROLLER & ARRAY	**for ARRAY the code is placed in the right column for CONTROLLER the code is placed in the left column for CONTROLLER the prefix 'c' is added**	
shift_rotate operations	shift right one bit position	SHR
	shift right arithmetic one bit position	ASHR
	shift right one bit position with carry	SHRC
	shift left one bit position	SHL
	shift right one bit position with carry	SHLC
	rotate right arithmetic one bit position	ROTR
	rotate left arithmetic one bit position	ROTL
insval	acc/acc[i] <= {(acc/acc[i] << 8), value[7:0]}	INSVAL
misc_store_load	addr/addr[i] <= acc/acc[i]	ADDRLD

Table 6.2: Instruction Set Architecture with no operand.

opCode\operand	COMMENTARIES	ISA_ctl
ARRAY ONLY		
misc_store_load	acc[i] <= i	IXLOADd
	acc[i] <= serialReg[i]	GETSR
	serialReg[i] <= acc[i]	SENDSR
	aMem[i][s] <= ioReg[i]	GETIO
	ioReg[i] <= aMem[i][s]	SENDIO
spatial_select		WHEREZERO
		WHERECARRY
		WHERENEGATIVE
		WHEREPREVACT
		WHEREFIRST
		WHERENEXT
		WHEREPREV
		WHERENZERO
		WHERENCARRY
		WHERENNEGATIVE
		WHERENPREVACT
		WHERENFIRST
		WHERENNEXT
		WHERENPREV
		ELSEWHERE
		SELSHIFT
		ACTIVATE
set_reduction		REDADD
		REDMIN
		REDMAX
		REDOR
delete		DELETE

Table 6.3: Array's Instruction Set Architecture with no operand.

CONTROLLER ONLY		
global_shift	1-position global right shift	GRSHIFT
	1-position global left shift	GLSHIFT
	reduction output insert right	RREDINS
	reduction output insert left	LREDINS
	1-position global right rotate	GRROTATE
	1-position global left rotate	GLROTATE
jmp	relative jump to label s	cJMP(s)
	absolute jump to label s	cAJMP(s)
	jump to label s if acc=0	cBRZ(s)
	jump to label s if acc!=0	cBRNZ(s)
	jump to label s if acc=0; acc<=acc-1	cBRZDEC(s)
	jump to label s if acc!=0; acc<=acc-1	cBRNZDEC(s)
		cHALT
misc_testing		cSTART
		cSTOP

Table 6.4: Controller's Instruction Set Architecture with no operand.

Part III

Berkeley view of parallel computing

Chapter 7

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" [1]. 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 [1] 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 8

The First Dwarf: Dense Linear Algebra

The problems approached in this chapter are:

- matrix transpose
- matrix-vector multiplication
- matrix-matrix multiplication
- matrix move
- matrix inverse

8.1 Matrix Transpose

8.1.1 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;
================================================================================
```

8.1.2 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]
//=============================================================================================
```

8.1.3 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 8.1 *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.

◇

8.2 Matrix-Vector Multiplication

The algorithm is standard.

8.2.1 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.

8.3 Matrix-Matrix Multiplication

The algorithm is standard.

8.3.1 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
```

8.3.2 The Verification

Example 8.2 *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:* $T_{vectMatrixMult} = 462$.

◇

8.4 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)
//=========================================================================
```

8.5 Matrix Inverse Using Gauss-Jordan Elimination

```
/*************************************************************************
Algorithm:  matrix inverse
   input:  n×n matrix A
   output:  n×n matrix A^-1
*************************************************************************/

    for (i=0; i<n; i=i+1)
        read column i
        select the first diagonal component: s[i,i]
        divide the i-th row, R[i}, by s[i,i]: R[i] <= R[i]/s[i,i]
        for (j=0; j<n; j=j+1)
            if ~(i=j) R[j} <= R[j} - R[i] x s[j]
```

Example 8.3 *Let us take a* 3×3 *matrix [6] and apply the algorithm for matrix inverse.*

$$A = \begin{bmatrix} 13 & 2 & 5 \\ 14 & 8 & 3 \\ 11 & 6 & 10 \end{bmatrix}$$

Write the augmented matrix adding, separated by bars, right columns representing the identity matrix I:

$$\left[\begin{array}{ccc|ccc} 13 & 2 & 5 & 1 & 0 & 0 \\ 14 & 8 & 3 & 0 & 1 & 0 \\ 11 & 6 & 10 & 0 & 0 & 1 \end{array}\right]$$

Step 1: divide R1 by 13 to obtain the 1 in the first diagonal

$$\left[\begin{array}{ccc|ccc} \mathbf{1} & 0.1538 & 0.3846 & 0.0769 & 0 & 0 \\ 14 & 8 & 3 & 0 & 1 & 0 \\ 11 & 6 & 10 & 0 & 0 & 1 \end{array}\right]$$

Step 2: R2 - 14R1 to obtain the first zero on the first column

$$\left[\begin{array}{ccc|ccc} 1 & 0.1538 & 0.3846 & 0.0769 & 0 & 0 \\ \mathbf{0} & 5.8462 & -2.3846 & -1.0769 & 1 & 0 \\ 11 & 6 & 10 & 0 & 0 & 1 \end{array}\right]$$

Step 3: R3 - 11R1

$$\left[\begin{array}{ccc|ccc} 1 & 0.1538 & 0.3846 & 0.0769 & 0 & 0 \\ 0 & 5.8462 & -2.3846 & -1.0769 & 1 & 0 \\ \mathbf{0} & 4.3077 & 5.7692 & -0.8462 & 0 & 1 \end{array}\right]$$

Step 4: divide R2 by 5.8462

$$\left[\begin{array}{ccc|ccc} 1 & 0.1538 & 0.3846 & 0.0769 & 0 & 0 \\ 0 & \mathbf{1} & -0.4079 & -0.1842 & 0.1711 & 0 \\ 0 & 4.3077 & 5.7692 & -0.8462 & 0 & 1 \end{array}\right]$$

Step 5: R1 - 0.1538R2

$$\left[\begin{array}{ccc|ccc} 1 & \mathbf{0} & 0.4474 & 0.1053 & -0.0263 & 0 \\ 0 & 1 & -0.4079 & -0.1842 & 0.1711 & 0 \\ 0 & 4.3077 & 5.7692 & -0.8462 & 0 & 1 \end{array}\right]$$

Step 6: R3 - 4.3077R2

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0.4474 & 0.1053 & -0.0263 & 0 \\ 0 & 1 & -0.4079 & -0.1842 & 0.1711 & 0 \\ 0 & \mathbf{0} & 7.5263 & -0.0526 & -0.7368 & 1 \end{array}\right]$$

Step 7: divide R3 by 7.5263

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0.4474 & 0.1053 & -0.0263 & 0 \\ 0 & 1 & -0.4079 & -0.1842 & 0.1711 & 0 \\ 0 & 0 & \mathbf{1} & -0.007 & -0.0979 & 0.1329 \end{array}\right]$$

Step 8: R1 - 0.4474R3

$$\left[\begin{array}{ccc|ccc} 1 & 0 & \mathbf{0} & 0.1084 & 0.0175 & -0.0594 \\ 0 & 1 & -0.4079 & -0.1842 & 0.1711 & 0 \\ 0 & 0 & 1 & -0.007 & -0.0979 & 0.1329 \end{array}\right]$$

Step 9: R2 + 0.4079R3

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0.1084 & 0.0175 & -0.0594 \\ 0 & 1 & \mathbf{0} & -0.1871 & 0.1311 & 0.0542 \\ 0 & 0 & 1 & -0.007 & -0.0979 & 0.1329 \end{array}\right]$$

Then:

$$A^{-1} = \left[\begin{array}{ccc} 0.1084 & 0.0175 & -0.0594 \\ -0.1871 & 0.1311 & 0.0542 \\ -0.007 & -0.0979 & 0.1329 \end{array}\right]$$

Chapter 9

The Second Dwarf: Sparse Linear Algebra

Part IV

Machine Learning

Chapter 10

What is Machine Learning?

Machine Learning is the science and art of using computers so they can *learn from data* without being explicitly programmed.

Instead of *writing rules* we must *train* according to an algorithm.

Applying ML techniques to dig into large amounts of data can help discover patterns that were not immediately apparent. This is called *data mining*.

There are so many different types of Machine Learning systems that it is useful to classify them in broad categories based on:

- Whether or not they are trained with human supervision (supervised, unsupervised, semisupervised, and Reinforcement Learning)
- Whether or not they can learn incrementally on the fly (online versus batch learning)
- Whether they work by simply comparing new data points to known data points, or instead detect patterns in the training data and build a predictive model, much like scientists do (instance-based versus model-based learning)

Chapter 11

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

11.1 K-means

K-means [**?**] is an unsupervised learning algorithms that solve the clustering problem.

11.1.1 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 11.1 *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:

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

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

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

Once loaded, the three 1023-component vectors are considered as 341 $3\times$ matrices. The $3\times$ 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.

$\diamond$

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.

11.1.2 Conceptual k-means clustering

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

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

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:

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

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: $X = \{x_1, \ldots, x_n\}$
- a set of attribute associated to the objects: $x_i = \langle b_{i1}, \ldots, b_{im} \rangle$ for $i = 1, \ldots, n$, 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:

$$\mathbf{S} = \begin{bmatrix} s_{11} & \ldots & s_{1n} \\ \vdots & \ddots & \vdots \\ s_{n1} & \ldots & s_{nn} \end{bmatrix}$$

by starting from the vector $[x_1, \ldots, x_n]$ with the computation of the symmetric matrix

$$\mathbf{XOR} = \begin{bmatrix} x_1 \oplus x_1 & x_1 \oplus x_2 & \ldots & x_1 \oplus x_n \\ x_2 \oplus x_1 & x_2 \oplus x_2 & \ldots & x_2 \oplus x_n \\ \vdots & \vdots & \ddots & \vdots \\ x_n \oplus x_1 & x_n \oplus x_2 & \ldots & x_n \oplus x_n \end{bmatrix}$$

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 **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 **S**,

   $$\mathbf{L} = \mathbf{D} - \mathbf{S}$$

   where **D** is the *degree matrix* of **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:

   $$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

   where **A** is an $n \times n$ matrix, **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 **A**. The vector **v** which corresponds to this value is the associated ***eigenvector***. The above equation is transformed as follows:

   $$\mathbf{A}\mathbf{v} - \lambda\mathbf{v} = 0$$

   $$\mathbf{A}\mathbf{v} - \lambda\mathbf{I}\mathbf{v} = 0$$

   $$(\mathbf{A}\mathbf{v} - \lambda\mathbf{I})\mathbf{v} = 0$$

   and, if **v** is a non-zero vector, then the equation has a solution if

   $$|\mathbf{A}\mathbf{v} - \lambda\mathbf{I}| = 0$$

   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 [**?**]:

   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

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

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

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

Example 11.2 *Let's consider a simple example (following* [**?**]*) of 6 objects:* $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ *defined by the the graph represented in Figure 11.1, 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:*

$$\mathbf{S} = \begin{bmatrix} 0 & 0.8 & 0.6 & 0 & 0.1 & 0 \\ 0.8 & 0 & 0.8 & 0 & 0 & 0 \\ 0.6 & 0.8 & 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0.2 & 0 & 0.8 & 0.7 \\ 0.1 & 0 & 0 & 0.8 & 0 & 0.8 \\ 0 & 0 & 0 & 0.7 & 0.8 & 0 \end{bmatrix}$$

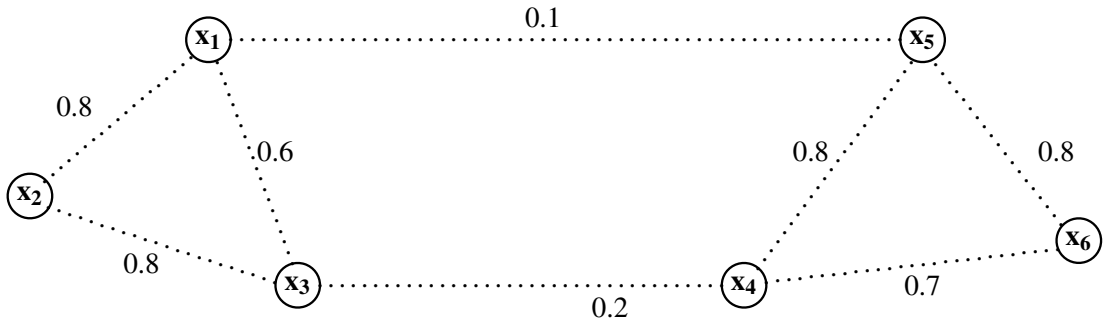

Figure 11.1:

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)

$$\mathbf{Y}^{(1)} = \frac{\mathbf{X}^{(1)}}{max(\mathbf{X}^{(1)})} = \frac{1}{1.6} \begin{bmatrix} -0.8 \\ 1.6 \\ -1 \\ 1 \\ -1.6 \\ 0.8 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 1 \\ -0.625 \\ 0.625 \\ -1 \\ 0.5 \end{bmatrix}$$

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 $\{\{x_1, x_2, x_3\}, \{x_4, x_5, x_6\}\}$

$\diamond$

11.2 Hierarchical clustering

11.3 Fuzzy C-means

11.4 Mixture of Gaussians

Chapter 12

Regression

Regression is an important algorithm used in machine learning [**?**]. 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 $x_1, ...$).

12.1 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 12.1 *In Figure 12.1 there are 16 points in a two-dimension space.*

$$\langle (4,3),(1,5),(5,5),(7,7),(10,7),(2,8),(5,10),(12,10),$$
$$(3,12),(8,13),(12,13),(10,14),(15,14),(13,16),(10,17),(16,18)\rangle$$

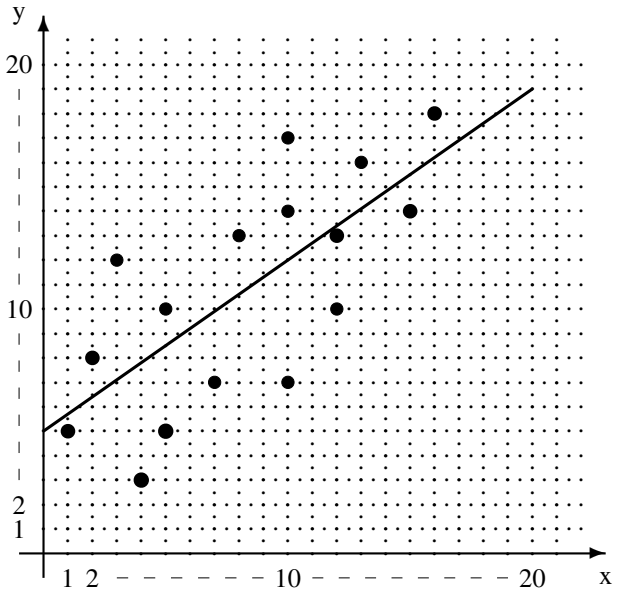

Figure 12.1:

```
/*********************************************************************
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
    RUN('MLOAD, 34, 112, 16, 1);     // load second vector with n/2 pairs
    hRUN('REG);                      // call  REG running on ACCELERATOR
    RUN('MSTORE, 37, 90, 4, 1);      // store the 4-scalar vector in HOST
    hRUN('EOP);                      // call END OF PROGRAM
    hWAITACC;                        // wait for end of program
    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 12.1 is drown.

◇

12.2 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 [**?**]) 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 12.2 *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.*

◇

Chapter 13

Markov Models

A Markov Model is a stochastic model which models temporal or sequential data. It is used to model randomly changing systems. It is assumed that the next state depend only on the current state, not on the events that occurred before it. Generally, this assumption enables reasoning and computation with the model that would otherwise be intractable. Markov models are named after their creator, Andrey Markov (1856–1922), a Russian mathematician [**?**] [**?**] [**?**] [**?**].

Hidden Markov Model is an unsupervised Machine Learning Algorithm.

Markov Model could be assimilated with a no-input stochastic finite half-automaton. The inputs to be tested in each state are substituted by the probabilities of transition from a state in another. The initial state is given be a vector of probabilities.

Hidden Markov Model could be assimilated with a no-input stochastic finite automaton.

13.1 Markov Models

Definition 13.1 *A first-order Markov Model is a n-node graph where each node is a state $s_i \in S = \{s_1, \ldots s_n\}$ and each edge a_{ij} represents the probability of going from the state s_i to the state s_j. It is completely described by the pair:*

$$MM = (\mathbf{A}, \Pi(0))$$

where:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix}$$

*is the $n \times n$ **state transition matrix** with $a_{ij} = P(s_j|s_i)$, and*

$$\Pi(0) = \begin{bmatrix} p_1 & p_2 & \ldots & p_n \end{bmatrix}$$

*is the initial **state distribution vector** with $p_i = P(s_i)$.*

◇

MM is a first-order Markov Model because, if the state at the moment t is $\sigma_t \in S$, then the probability of being in a state at time t depends only on the probability of the state at time $t-1$:

$$P(\sigma_t|\sigma_0, \sigma_1, \ldots \sigma_{t-1}) = P(\sigma_t|\sigma_{t-1})$$

Having a MM, the main application is to compute the probability distribution of the state at the moment t. Because the state distribution vector at the moment t is: $p(t) = p(t-1)\mathbf{A}$, related to the initial state distribution vector we have:

$$\Pi(t) = ((((\Pi(0)\mathbf{A})\mathbf{A})\ldots)\mathbf{A}) = \Pi(0)\mathbf{A}^t$$

Example 13.1 *Let be $S = \{sun, cloud, rain\}$ with the Markov Model represented as the graph from Figure 13.1 [**?**].*

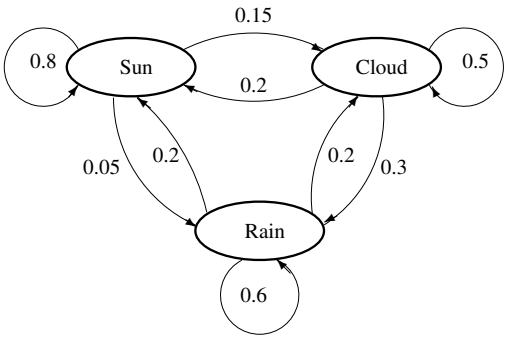

Figure 13.1: Example of Markov Model.

The transition probability matrix is:

$$\mathbf{A} = \begin{bmatrix} P(sun|sun) & P(cloud|sun) & P(rain|sun) \\ P(sun|cloud) & P(cloud|cloud) & P(rain|cloud) \\ P(sun|rain) & P(cloud|rain) & P(rain|rain) \end{bmatrix} = \begin{bmatrix} 0.8 & 0.15 & 0.05 \\ 0.2 & 0.5 & 0.3 \\ 0.2 & 0.2 & 0.6 \end{bmatrix}$$

where: $P(sun|sun)$ is the probability to have a sunny day followed by another sunny day, $P(cloud|sun)$ is the probability to have a cloudy day preceded by a sunny day, and so on.

If $\Pi(0) = [1,0,0]$ (it is a sunny day) what is $\Pi(1)$ (the forecast for tomorrow)? The answer is:

$$\Pi(1) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0.8 & 0.15 & 0.05 \\ 0.2 & 0.5 & 0.3 \\ 0.2 & 0.2 & 0.6 \end{bmatrix} = \begin{bmatrix} 0.8 & 0.15 & 0.05 \end{bmatrix}$$

It will be a sunny day with the probability of 0.8, a cloudy day with the probability of 0.15, or a rainy day with the probability of 0.05.

The forecast for the day after tomorrow is:

$$\Pi(2) = \Pi(0)\mathbf{A}^2 = \begin{bmatrix} 0.68 & 0.205 & 0.115 \end{bmatrix}$$

$\diamond$

The main computational challenge imposed by using a Markov Model is the vector-matrix multiplication. Working with probabilities the fix point arithmetic is recommended.

13.1.1 Eigenvector & Eigenvalue

Consider we start with a state distribution vector $\Pi(t)$, and we multiply by the matrix $\mathbf{A}$, and we end up with the same state distribution vector:

$$\Pi(t) = \Pi(t)\mathbf{A}$$

This is what we call a *stationary distribution*, because no matter how many times we make the transition from this state distribution, we still have the same state distribution.

Definition 13.2 *If*

$$\Pi\mathbf{A} = \lambda\Pi$$

then the vector Π *is called the* eigenvector *of the matrix* $\mathbf{A}$ *and* λ *is the* eigenvalue*.*

◇

13.2 Hidden Markov Models

HMM is a probabilistic finite state automaton, with probabilistic outputs. The Hidden Markov Model is a process that has two levels:

- ***embedded level***: is a Markov process and has the unobservable states $S = \{s_1, \ldots, s_n\}$ with the unobservable series of state $\vec{\sigma} = \{\sigma_1, \sigma_2, \ldots, \sigma_T\}$, with $\sigma_i \in S$ (it could be associated to a half-automaton-like engine)
- ***observable level***: is what we can actually observe – that is the output that internal states emit – the series of observed outputs $\vec{\omega} = \{\omega_1, \omega_2, \ldots, \omega_T\}$ with $\omega_i \in O = \{o_1, \ldots, o_m\}$ where O is the output alphabet (could be associated to an automaton-like engine whose internal states are unobservable).

Definition 13.3 *A Hidden Markov Model is defined by the following pair:*

$$HMM = (MM, \mathbf{B})$$

where: MM is a Markov Model (see Definition 13.1), and $\mathbf{B}$ *describe the observable behavior of MM defined by elements of the* ***output alphabet*** $O = \{o_1, \ldots, o_m\}$

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \ldots & b_{1n} \\ b_{21} & b_{22} & \ldots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \ldots & b_{mn} \end{bmatrix}$$

where: $b_{jk} = P(\omega_t = o_k | \sigma_t = s_j)$*, with* $\omega_t \in O$ *the value at any moment t in the observed series*

$$\vec{\omega} = \{\omega_1, \omega_2, \ldots, \omega_T\}$$

◇

Example 13.2 *Let be a HMM with* $O = \{o_1, \ldots, o_8\}$*. A possible observed output is:*

$$\vec{\omega} = \{\omega_1 = o_5, \omega_2 = o_1, \omega_3 = o_7, \omega_4 = o_2, \omega_5 = o_1, \omega_6 = o_5\}$$

◇

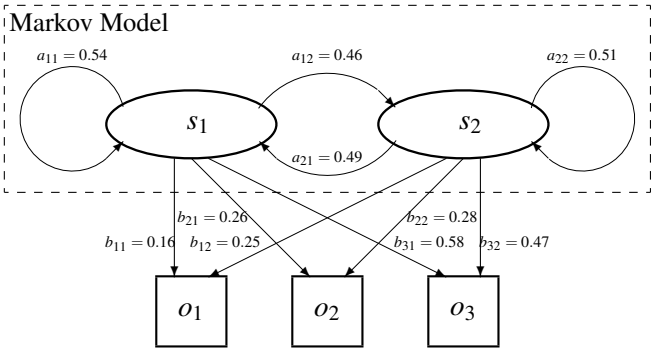

Figure 13.2: Example of Hidden Markov Model.

Example 13.3 *Let be HMM* $= (\mathbf{A}, \mathbf{B}, \Pi(0))$ *represented in Figure 13.2 defined by:*

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 0.54 & 0.46 \\ 0.49 & 0.51 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} 0.16 & 0.25 \\ 0.26 & 0.28 \\ 0.58 & 0.47 \end{bmatrix}$$

$$\Pi(0) = \begin{bmatrix} p_1 & p_2 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}$$

with $S = \{s_1, s_2\}$ *and* $O = \{o_1, o_2, o_3\}$.

◇

There are three problems related to the Hidden Markov Model:

- ***Evaluation***: since the state of Markov Model is hidden, we cannot actually be sure that it did generate the outcome. We have a model and we have a sequence we observe, but what are the probabilities that what we can see is really an emission product? We cannot directly observe the state, so how can we know if our assumption is accurate? To calculate our 'belief state' we use the *Forward Backward Algorithm*
- ***Learning***: provides what model has the highest probability of generating the observed outcome? What are the parameters that generated the observed sequences? For this problem, we use the *Baum-Welch Algorithm*
- ***Decoding***: means to find what is the most likely hidden and unobserved sequence, based on what we can observe? Which state has the highest possibility of being accurate? To decode what state the output refers to we use the *Viterbi Algorithm*

13.2.1 Evaluation problem

Given $M = (\mathbf{A}, \mathbf{B}, \Pi(0))$ and the observation sequence $\vec{\omega} = \{\omega_1, \omega_2, \ldots, \omega_T\}$, calculate the probability that $\vec{\omega}$ is generated by M.

Forward recursion

Definition 13.4 *The variable $\alpha_t(i) = P(\omega_1\omega_2\ldots\omega_t, \sigma_t = s_i)$ is defined as the joint probability of the partial observation sequence $\{\omega_1\omega_2\ldots\omega_t\}$ and that the hidden state at time t is s_i.*

For each t is defined the vector: $\alpha_t = [\alpha_t(1)\alpha_t(2)\ldots\alpha_t(n)]$.

◇

Definition 13.5 *The Hadamard (entrywise) product is defined as follows:*

- *for two $n \times m$-element matrices $P_{n\times m}$ and $R_{n\times m}$:*

$$H_{n\times m} = P \circ R = R \circ P = \begin{bmatrix} p_{11} & \ldots & p_{1m} \\ \vdots & \ddots & \vdots \\ p_{n1} & \ldots & p_{nm} \end{bmatrix} \circ \begin{bmatrix} r_{11} & \ldots & r_{1m} \\ \vdots & \ddots & \vdots \\ r_{n1} & \ldots & r_{nm} \end{bmatrix} = \begin{bmatrix} r_{11}p_{11} & \ldots & r_{1m}p_{1m} \\ \vdots & \ddots & \vdots \\ r_{n1}p_{n1} & \ldots & r_{nm}p_{nm} \end{bmatrix}$$

- *for $P_{n\times 1}$ and $R_{n\times m}$:*

$$H_{n\times m} = P \circ R = R \circ P = \begin{bmatrix} p_{11} \\ \vdots \\ p_{n1} \end{bmatrix} \circ \begin{bmatrix} r_{11} & \ldots & r_{1m} \\ \vdots & \ddots & \vdots \\ r_{n1} & \ldots & r_{nm} \end{bmatrix} = \begin{bmatrix} r_{11}p_{11} & \ldots & r_{1m}p_{11} \\ \vdots & \ddots & \vdots \\ r_{n1}p_{n1} & \ldots & r_{nm}p_{n1} \end{bmatrix}$$

- *for $P_{n\times m}$ and $R_{1\times m}$:*

$$H_{n\times m} = P \circ R = R \circ P = \begin{bmatrix} p_{11} & \ldots & p_{1m} \\ \vdots & \ddots & \vdots \\ p_{n1} & \ldots & p_{nm} \end{bmatrix} \circ \begin{bmatrix} r_{11} & \ldots & r_{1m} \end{bmatrix} = \begin{bmatrix} r_{11}p_{11} & \ldots & r_{1m}p_{1m} \\ \vdots & \ddots & \vdots \\ r_{11}p_{n1} & \ldots & r_{1m}p_{nm} \end{bmatrix}$$

◇

Example 13.4

$$\begin{bmatrix} 2 & 4 & 6 \end{bmatrix} \circ \begin{bmatrix} 3 & 5 & 7 \end{bmatrix} = \begin{bmatrix} 6 & 40 & 42 \end{bmatrix}$$

◇

Definition 13.6 *The Hadamard (entrywise) division, $\oslash$, is defined similarly for:*

- *for two $n \times m$-element matrices $P_{n\times m}$ and $R_{n\times m}$: $H_{n\times m} = P \oslash R$*
- *for $P_{n\times m}$ and $R_{n\times 1}$: $H_{n\times m} = P \oslash R$*
- *for $P_{n\times 1}$ and $R_{1\times m}$: $H_{n\times m} = P \oslash R$*

◇

Definition 13.7 *The reduction sum applied to the vector $V = [v_1\ldots v_n]$ is*

$$redSum(V) = \Sigma_1^n v_i$$

◇

Forward Algorithm: let us consider $M = (\mathbf{A}, \mathbf{B}, \Pi(0))$ and the observed outputs $\vec{\omega} = \{\omega_1, \omega_2, \ldots, \omega_T\}$. The probability that M generates $\vec{\omega}$ starting from $\Pi(0)$ is computed as follows:

Initialization :

$$\alpha_1 = \big[\alpha_1(1)\alpha_1(2)\ldots\alpha_1(n)\big] = \big[p_1 p_2 \ldots p_n\big] \circ \big[B(\omega_1,1)B(\omega_1,2)\ldots B(\omega_1,n)\big] = \Pi(0) \circ \mathbf{B}(\omega_1,-)$$

where $\mathbf{B}(\omega_1,-)$ is the line in the matrix $\mathbf{B}$ selected by ω_1

Forward recursion :

$$\alpha_{t+1} = (\alpha_t \mathbf{A}) \circ \mathbf{B}(\omega_{t+1},-)$$

for $t = 1\ldots(T-1)$ where $\mathbf{B}(\omega_{t+1},-)$ is the line in the matrix $\mathbf{B}$ selected by ω_{t+1}

Termination : $P(\vec{\omega}) = redSum(\alpha_T)$.

The execution time is in $O(n^2 T)$.

Example 13.5 *Let be HMM* $= (\mathbf{A}, \mathbf{B}, \Pi(0))$ *defined in Example 13.3 and* $\vec{\omega} = \{o_1, o_2, o_3\}$.
The forward algorithm runs as follow:

initialization :

$$\alpha_1 = \Pi(0) \circ B(1,-) = \begin{bmatrix}0.5 & 0.5\end{bmatrix} \circ \begin{bmatrix}0.16 & 0.25\end{bmatrix} = \begin{bmatrix}0.08 & 0.125\end{bmatrix}$$

forward recursion :

first step :

$$\alpha_2 = (\alpha_1 \mathbf{A}) \circ B(2,-) = (\begin{bmatrix}0.08 & 0.12\end{bmatrix} \begin{bmatrix}0.54 & 0.46\\ 0.49 & 0.51\end{bmatrix}) \circ \begin{bmatrix}0.26 & 0.28\end{bmatrix} = \begin{bmatrix}0.0271 & 0.0281\end{bmatrix}$$

second step :

$$\alpha_3 = (\alpha_2 \mathbf{A}) \circ B(3,-) = (\begin{bmatrix}0.0271 & 0.0281\end{bmatrix} \begin{bmatrix}0.54 & 0.46\\ 0.49 & 0.51\end{bmatrix}) \circ \begin{bmatrix}0.58 & 0.47\end{bmatrix} =$$

$$= \begin{bmatrix}0.0165069392 & 0.0126198572\end{bmatrix}$$

termination : $P(\{1,2,3\}) = 0.0165069392 + 0.0126198572$

◇

Backward recursion

Backward algorithm is the time-reversed version of the forward algorithm. In backward algorithm we find the probability that the machine will be in hidden state s_i at time step t and will generate the remaining part of the sequence of the visible symbol $\vec{\omega}$.

Definition 13.8 *Define the variable* $\beta_t(i)$ *as the joint probability of the partial observation sequence* $\omega_{t+1}\omega_{t+2}\ldots\omega_T$ *given that the hidden state at time t is* s_i*:*

$$\beta_t(i) = P(\omega_{t+1}\omega_{t+2}\ldots\omega_T)|\sigma_t = s_i)$$

For each t is defined the vector: $\beta_t = [\beta_t(1)\beta_t(2)\ldots\beta_t(n)]$.

◇

Backward Algorithm: let us consider $M = (\mathbf{A}, \mathbf{B}, \Pi(0))$ and the observed outputs $\vec{\omega} = \{\omega_1, \omega_2, \ldots, \omega_T\}$. The probability that M generates $\vec{\omega}$ starting from $\Pi(0)$ is computed as follows:

Initialization :

$$\beta_T = \left[\beta_T(1)\beta_T(2)\ldots\beta_T(n)\right] = \left[11\ldots1\right]$$

Backward recursion :

$$\beta_t = (\beta_{t+1} \circ \mathbf{B}(\omega_{t+1}, -))\mathbf{A}'$$

for $t = T-1\ldots1$ where $\mathbf{A}'$ is the matrix $\mathbf{A}$ transposed ($a_{ij} \Leftarrow a_{ji}$)

Termination : $P(\vec{\omega}) = redSum(\beta_1 \circ \mathbf{B}(\omega_1, -) \circ \Pi(0))$.

The execution time is in $O(n^2T)$.

Example 13.6 *Let be $M = (\mathbf{A}, \mathbf{B}, \Pi 0)$ defined in Example 13.5, and $\vec{\omega} = \{o_2, o_1, o_2\}$ [?]. The backward algorithm runs as follow:*

initialization :

$$\beta_4 = \left[11\right]$$

backward recursion :

first step :

$$\beta_3 = (\beta_4 \circ B(2,-))\mathbf{A} = (\begin{bmatrix}1 & 1\end{bmatrix} \circ \begin{bmatrix}0.58 & 0.47\end{bmatrix}) \begin{bmatrix}0.54 & 0.49\\ 0.46 & 0.51\end{bmatrix} = \begin{bmatrix}0.5294 & 0.5239\end{bmatrix}$$

second step :

$$\beta_2 = (\beta_3 \circ B(1,-))\mathbf{A} = (\begin{bmatrix}0.5294 & 0.5239\end{bmatrix} \circ \begin{bmatrix}0.26 & 0.28\end{bmatrix}) \begin{bmatrix}0.54 & 0.49\\ 0.46 & 0.51\end{bmatrix} =$$

$$\begin{bmatrix}0.14180608 & 0.14225848\end{bmatrix}$$

third step :

$$\beta_1 = (\beta_2 \circ B(2,-))\mathbf{A} = (\begin{bmatrix}0.14180608 & 0.14225848\end{bmatrix} \circ \begin{bmatrix}0.58 & 0.47\end{bmatrix}) \begin{bmatrix}0.54 & 0.49\\ 0.46 & 0.51\end{bmatrix} =$$

$$= \begin{bmatrix}0.0751699476 & 0.0744006456\end{bmatrix}$$

termination :

$$P(\vec{x}) = redSum(\begin{bmatrix}0.07516994 & 0.07440064\end{bmatrix} \circ \begin{bmatrix}0.58 & 0.47\end{bmatrix} \circ \begin{bmatrix}0.5 & 0.5\end{bmatrix}) = 0.039283$$

$\diamond$

13.2.2 Learning problem

Given some training observation sequences $\vec{\omega} = \{\omega_1, \omega_2, \ldots, \omega_T\}$ and the numbers of hidden variable and of visible outputs, determine the parameters of $MM = (\mathbf{A}, \mathbf{B}, \Pi(0)$ that best fit training data, i.e., maximize $P(\vec{\omega}|MM)$. There is no algorithm producing optimal parameter values. The iterative expectation-maximization algorithm is used to find a local maximum of $P(\vec{\omega}|MM)$. It is called *Baum-Welch algorithm*.

Definition 13.9 *The probability of the transition between state s_i and s_j at time t given the observations $\{\omega_1, \omega_2, \ldots, \omega_t\}$ is proportional to:*

$$g_t(i,j) = \alpha_t(i) a_{ij} b_{j\omega_t} \beta_{t+1}(j)$$

◇

Definition 13.10 *The maximum likelihood for transition form s_i to s_j at any time is:*

$$a_{ij} = \frac{\Sigma_T g_t(i,j)}{\Sigma_j \Sigma_T g_t(i,j)}$$

◇

Definition 13.11 *The maximum likelihood for o_k to be generated by the state s_j at any time is:*

$$b_{jk} = \frac{\Sigma_i \Sigma_T 1\{\omega_t = o_k\} g_t(i,j)}{\Sigma_i \Sigma_T g_t(i,j)}$$

◇

Learning Algorithm

Initialization : initialize the matrices **A** and **B** with equal probabilities or randomly distributed probabilities.

Repeat until convergence :

Compute expectation : run the Forward and Backward algorithms to compute α_i and β_i for $i = 1 \ldots n$, and compute the matrices

$$\mathbf{G}_t = \begin{bmatrix} g_t(1,1) & \ldots & g_t(1,n) \\ g_t(2,1) & \ldots & g_t(2,n) \\ \vdots & \ddots & \vdots \\ g_t(n,1) & \ldots & g_t(n,n) \end{bmatrix} = \alpha_t' \circ \mathbf{A} \circ (\mathbf{B}(\omega_t, -) \circ \beta_{t+1})$$

for $t = 1 \ldots T$.

Re-estimate : the probability matrices **A** and **B** are recomputed as follows:

for state transition matrix :

- add the matrices $\mathbf{G}_t$

$$\mathbf{G} = \Sigma_T \mathbf{G}_t$$

- generate the vector C, as a $n \times 1$ matrix, by summing all the elements of each row of **G**

$$C = \Sigma_j \mathbf{G}(i,j)$$

- instantiate the new **A** by entrywise dividing (see Definition 13.6) each column of **G** by vector **C**

$$\mathbf{A} \Leftarrow \mathbf{G} \oslash C$$

for output transition matrix :

- compute the matrices

$$\mathbf{G}^k = \Sigma_T \mathbf{G}_t(\omega_t = o_k)$$

for $k = 1 \dots m$

- construct the matrix:

$$\Gamma = \begin{bmatrix} \Sigma_i \mathbf{G}^1 \\ \Sigma_i \mathbf{G}^2 \\ \vdots \\ \Sigma_i \mathbf{G}^m \end{bmatrix}$$

where $\Sigma_i \mathbf{G}^k = [\Sigma_i g^k(i,1), \Sigma_i g^k(i,2), \dots, \Sigma_i g^k(i,n)]$

- generate the vector R, as a $1 \times n$ matrix, by summing all the elements of each column of **G**

$$R = \Sigma_i \mathbf{G}(i,j)$$

- instantiate the new **B** by entrywise dividing the matrix Γ with the vector R:

$$\mathbf{B} \Leftarrow \Gamma \oslash R$$

13.2.3 Decoding problem

Given $M = (\mathbf{A}, \mathbf{B}, \Pi(0))$ and the observation sequence $\vec{\omega} = \{\omega_1, \omega_2, \dots, \omega_T\}$, we must calculate the most likely sequence of $\vec{\sigma}$ that produced $\vec{\omega}$. This algorithm is similar to the forward recursion of evaluation problem, with *redSum* operations replaced by *redMax* and additional backtracking.

Definition 13.12 *The pseudo-multiplication of a matrix $P_{n\times m}$ with a matrix $R_{m\times q}$, $P \bullet R$, is performed substituting, in the multiplication algorithm, the reduction sum operation, performed in the dot products of the m-component vector, with the selection of the maximum value from those of the vectors resulting from the Hadamard product.*

◇

Example 13.7

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \bullet \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} max(1\times 1, 2\times 2) \\ max(1\times 3, 2\times 4) \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \end{bmatrix}$$

◇

Definition 13.13 *Let be $\alpha(t) = [\alpha_1(t), \alpha_2(t), \dots, \alpha_n(t)]$, where $\alpha_i(t)$ is the probability to reach state s_i via an optimal sequence of states after emitting the first t observations:*

$$\alpha_i(t) = max_{j=1}^{n}(\alpha_j(t-1)A(j,i)B(\omega_t, i))$$

◇

Viterbi Algorithm

Initialization :

$$\alpha(1) = \Pi(0) \circ \mathbf{B}(\omega_1, -)$$

Iteration :

$$\alpha(t) = (\alpha(t-1) \bullet \mathbf{A}) \circ \mathbf{B}(\omega_t, -)$$

$$\vec{\sigma}(t) \Leftarrow \{\vec{\sigma}(t-1), s_{indexMax(\alpha(t+1))}\}$$

where: $indexMax(\alpha(i))$ is the index of the maximum value in $\alpha(i)$.

Termination : $\vec{\sigma}(T)$

Example 13.8 *Let be:*

$$\mathbf{A} = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 0.5 & 0.1 \\ 0.4 & 0.3 \\ 0.1 & 0.6 \end{bmatrix}$$

$$\Pi(0) = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix}$$

and the observed sequence $\vec{\omega} = \{o_1, o_2, o_3\}$ *[?].*

Then, the Viterbi algorithm runs as follows:

initialization *:*

$$\alpha(1) = \Pi(0) \circ B(o_1, -) = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix} \circ \begin{bmatrix} 0.5 & 0.1 \end{bmatrix} = \begin{bmatrix} \mathbf{0.3} & 0.04 \end{bmatrix}$$

because $indexMax(\alpha(1) = 1$

iteration *:*

first iteration *:*

$$\alpha(2) = (\alpha(1) \bullet \mathbf{A}) \circ B(o_2, -) =$$

$$= \begin{bmatrix} 0.3 & 0.04 \end{bmatrix} \bullet \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \circ \begin{bmatrix} 0.4 & 0.3 \end{bmatrix} =$$

$$= \begin{bmatrix} max(0.21, 0.016) & max(0.09, 0.024) \end{bmatrix} \circ \begin{bmatrix} 0.4 & 0.3 \end{bmatrix} = \begin{bmatrix} \mathbf{0.084} & 0.027 \end{bmatrix}$$

second iteration *:*

$$\alpha(3) = \begin{bmatrix} 0.084 & 0.027 \end{bmatrix} \bullet \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \circ \begin{bmatrix} 0.1 & 0.6 \end{bmatrix} =$$

$$= \begin{bmatrix} max(0.058, 0.0108) & max(0.0252, 0.0162) \end{bmatrix} \circ \begin{bmatrix} 0.1 & 0.6 \end{bmatrix} = \begin{bmatrix} 0.0058 & \mathbf{0.01512} \end{bmatrix}$$

termination *:* $\vec{\sigma}(T) = \{s_1, s_1, s_2\}$

$\diamond$

Part V

Neural Networks & Machine Learning

Chapter 14

Artificial Neural Network

Artificial **neural network** (NN) is a technical construct inspired from the biological neural networks. NN are composed of interconnected artificial **neurons**. An artificial neuron is a programmed or circuit construct that mimic the property of a biological neuron. A multi-layer NN is used as a connectionist computational model. The introductory text [47] is used for a short presentation of the concept of NN.

14.1 The neuron

The artificial neuron (see Figure 14.1) receives the inputs $x_1, \ldots, x_n$ (corresponding to *n dendrites*) and process them to produce an output o (*synapse*). The sums of each node are weighted, using the weight vector $w_1, \ldots, w_n$ and the sum, *net*, is passed through a ***non-linear function***, $f(net)$, called activation function or transfer function. The transfer functions usually have a sigmoid shape (see Figure 14.2) or step functions.

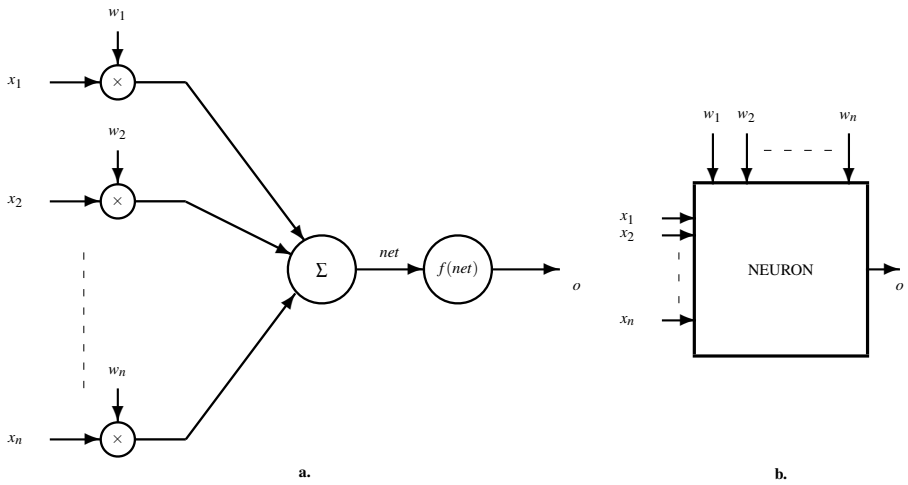

Figure 14.1: **The general form of a neuron. a.** The circuit structure of a *n*-input neuron. **b.** The logic symbol.

Formally, the transfer function of a neuron:

$$o = f(\sum_{i=1}^{n} w_i x_i) = f(net)$$

where f, the typical activation function, is:

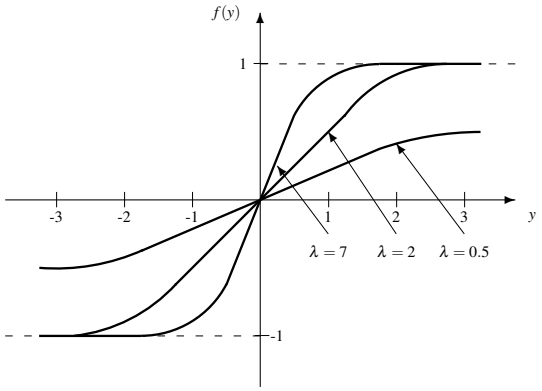

Figure 14.2: **The activation function.**

$$f(y) = \frac{2}{1 + exp(-\lambda y)} - 1$$

The parameter λ determines the steepness of the continuous function f. For big value of λ the function f becomes:

$$f(y) = sgn(y)$$

The neuron works as a combinational circuit performing the scalar product of the input vector

$$\mathbf{x} = [x_1\ x_2\ \ldots\ x_n]$$

with the weight vector

$$\mathbf{w} = [w_1\ w_2\ \ldots\ w_n]$$

followed by the application of the activation function. The activation function f is simply implemented using as a look-up table using a Read-Only Memory.

14.2 The feedforward neural network

A feedforward NN is a collection of m n-input neurons (see Figure 14.3). Each neuron receives the same input vector

$$\mathbf{x} = [x_1\ x_2\ \ldots\ x_m]$$

and is characterized by its own weight vector

$$\mathbf{w}_i = [w_1\ w_2\ \ldots\ w_m]$$

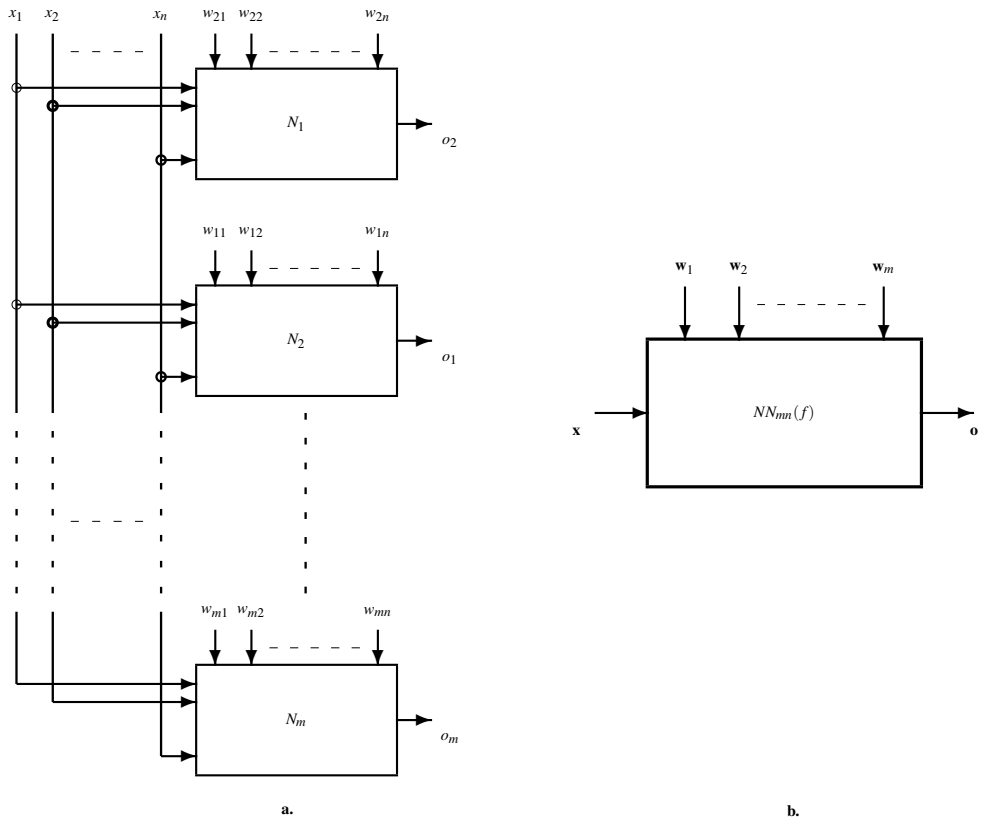

Figure 14.3: **The single-layer feedforward neural network. a.** The organization of a feedforward NN having m n-input neurons. **b.** The logic symbol.

The entire NN provides the output vector

$$\mathbf{o} = [o_1 \; o_2 \; \ldots \; o_m]^t$$

The activation function is the same for each neuron.

Each NN is characterized by the weight matrix

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \ldots & w_{1n} \\ w_{21} & w_{22} & \ldots & w_{2n} \\ \vdots & \vdots & \ldots & \vdots \\ w_{m1} & w_{m2} & \ldots & w_{mn} \end{bmatrix}$$

having for each output a line, while for each input it has a column. The transition function of the NN is:

$$\mathbf{o}(t) = \Gamma[\mathbf{W}\mathbf{x}(t)]$$

where:

$$\Gamma[\cdot] = \begin{bmatrix} f(\cdot) & 0 & \dots & 0 \\ 0 & f(\cdot) & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & f(\cdot) \end{bmatrix}$$

The feedforwaed NN is of "instantaneous" type, i.e., it behaves as a combinational circuit which provides the result in the same "cycle". The propagation time associated do not involve storage elements.

Example 14.1 *The shaded area in Figure 14.4 must be recognized by a two-layer feedforward NN. Four conditions must be met to define the surface:*

$$\begin{aligned} &x_1 - 1 > 0 \rightarrow sgn(x_1 - 1) = 1 \\ &x_1 - 2 < 0 \rightarrow sgn(-x_1 + 2) = 1 \\ &x_2 > 0 \quad \rightarrow sgn(x_2) = 1 \\ &x_2 - 3 < 0 \rightarrow sgn(-x_2 + 3) \end{aligned}$$

For each condition a neuron from the first layer is used. The second layer determines whether all the conditions tested by the first layer are fulfilled.

The first layer is characterized the weight matrix

$$\mathbf{W}_{43} = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & -1 & -3 \end{bmatrix}$$

The weight vector for the second layer is

$$\mathbf{W} = [1 \;\; 1 \;\; 1 \;\; 1 \;\; 3.5]$$

On both layers the activation function is sgn.

◇

14.3 The feedback neural network

The feedback NN is a sequential system. It provides the output with a delay of a number of clock cycles after the initialization with the input vector **x**. The structure of a feedback NN is presented in Figure 14.5. The multiplexor **mux** is used to initialize the loop closed through **register**. If *init = 1* the vector **x** is applied to $NN_{mn}(f)$ one clock cycle, then *init* is switched to 0 and the loop is closed.

In the circuit approach of this concept, after the initialization cycle the output of the network is applied to the input through the feedback register. The transition function is:

$$\mathbf{o}(t + T_{clock}) = \Gamma[\mathbf{W}\mathbf{o}(t)]$$

where T_{clock} (the clock period) is the delay on the loop. After k clock cycles the state of the network is described by:

$$\mathbf{o}(t + k \times T_{clock}) = \Gamma[\mathbf{W}\Gamma[\dots\Gamma[\mathbf{W}\mathbf{o}(t)]\dots]]$$

A feedback NN can be considered as an initial *automaton* with few final states mapping disjoint subsets of inputs.

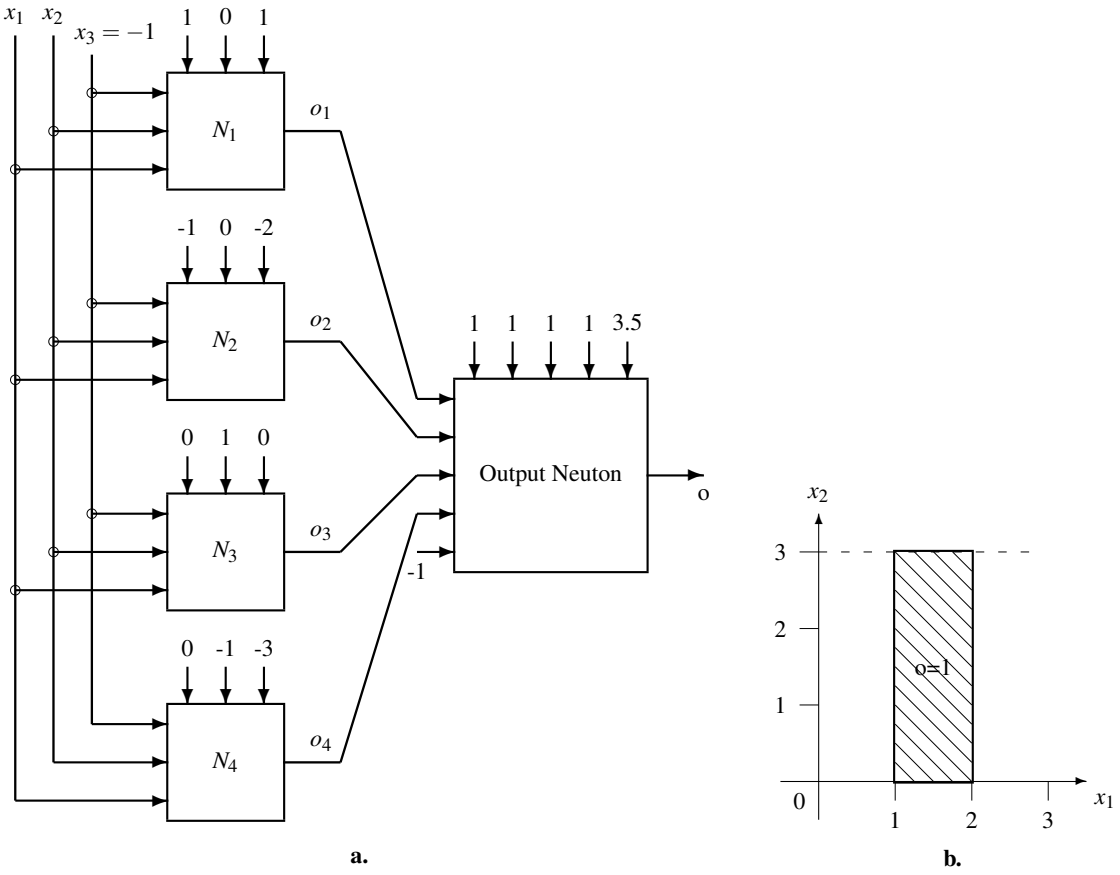

Figure 14.4: **A two-layer feedforward NN. a.** The structure. **b.** The two-dimension space mapping.

Example 14.2 *Let be a feedback NN with 4 4-input neurons with one-bit inputs and outputs. The activation function is* sgn*. The feedback NN can be initialized with any 4-bit binary configuration from*
x = `[-1 -1 -1 -1]`
to
x = `[1 1 1 1]`
and the system has two final states:
$\mathbf{o}_{14}$ = `[1 1 1 -1]`
$\mathbf{o}_1$ = `[-1 -1 -1 1]`
reached in a number of clock cycles after the initialization.

The resulting discrete-time recurrent network has the following weight matrix:

$$\mathbf{W}_{44} = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

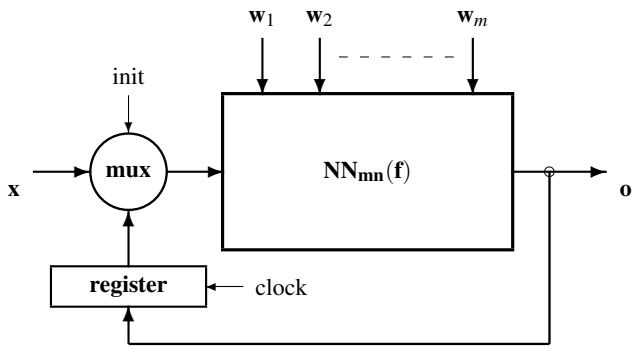

Figure 14.5: **The single-layer feedback neural network.**

The resulting structure of the NN is represented in Figure 14.6, where the weight matrix is applied on the four 4-bit inputs destined for the weight vectors.

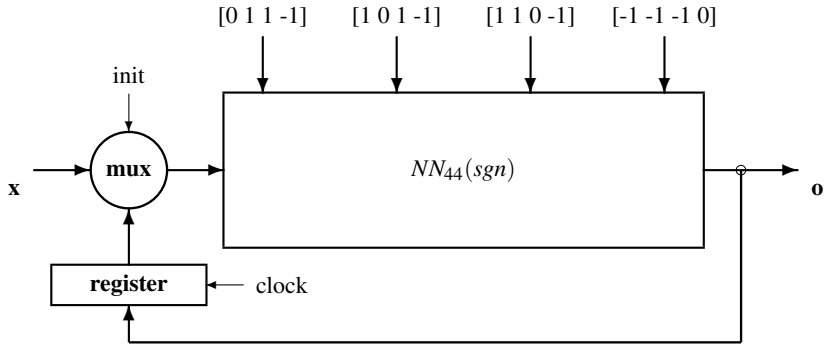

Figure 14.6: **The feedback NN with two final states.**

The sequence of transitions are computed using the form:

$$\mathbf{o}(t+1) = [sgn(net_1(t))\ \ sgn(net_2(t))\ \ sgn(net_3(t))\ \ sgn(net_4(t))]$$

Some sequences end in $\mathbf{o}_{14}$ `= [1 1 1 -1]`*, while others in* $\mathbf{o}_1$ `= [-1 -1 -1 1]`.

◇

14.4 The learning process

The learning process is used to determine the actual form of the matrix **W**. The learning process is an iterative one. In each iteration, for each neuron the weight vector **w** is adjusted with $\Delta\mathbf{w}$, which is

proportional with the input vector $\mathbf{x}$ and the learning signal r. The general form of the learning signal is:

$$r = r(\mathbf{w}, \mathbf{x}, d)$$

where d is the desired response (the teacher's signal). Thus, in each step the weight vector is adjusted as follows:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + c \times r(\mathbf{w}(t), \mathbf{x}(t), d(t)) \times \mathbf{x}(t)$$

where c is the *learning constant*. The learning process starts from an initial form of the weight vector (established randomly or by a simple "hand calculation") and uses as a set of training input vectors.

There are two types of learning:

unsupervised learning :

$$r = r(\mathbf{w}, \mathbf{x})$$

the desired behavior is not known; the network will adapt its response by "discovering" the appropriate values for the weight vectors by *self-organization*

supervised learning :

$$r = r(\mathbf{w}, \mathbf{x}, d)$$

the desired behavior, d, is known and can be compared with the actual behavior of the neuron in order to find how to adjust the weight vector.

In the following both types will be exemplified using the *Hebbian rule* and the *perceptron rule*.

14.4.1 Unsupervised learning: Hebbian rule

The learning signal is the output of the neuron. In each step the vector $\mathbf{w}$ will be adjusted (see Figure 14.7) as follows:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + c \times f(\mathbf{w}(t), \mathbf{x}(t)) \times \mathbf{x}(t)$$

The learning process starts with small random values for w_i.

Example 14.3 *Let be a four-input neuron with the activation function sgn. The initial weight vector is:*

$$\mathbf{w}(t_0) = [1 \ -1 \ 0 \ 0.5]$$

The training inputs are:
$\mathbf{x}_1 = [1 \ -2 \ 1.5 \ 0]$,
$\mathbf{x}_2 = [1 \ -0.5 \ -2 \ -1.5]$,
$\mathbf{x}_3 = [0 \ 1 \ -1 \ 1.5]$

Applying by turn the three training input vectors for $c = 1$ we obtain:
$\mathbf{w}(t_0+1) = \mathbf{w}(t_0) + sgn(net) \times \mathbf{x}_1 = \mathbf{w}(t_0) + sgn(3) \times \mathbf{x}_1 = \mathbf{w}(t_0) + \mathbf{x}_1 = [2 \ -3 \ 1.5 \ 0.5]$
$\mathbf{w}(t_0+2) = \mathbf{w}(t_0+1) + sgn(net) \times \mathbf{x}_2 = \mathbf{w}(t_0+1) + sgn(-0.25) \times \mathbf{x}_2 = \mathbf{w}(t_0+1) - \mathbf{x}_2 = [1 \ -2.5 \ 3.5 \ 2]$
$\mathbf{w}(t_0+3) = \mathbf{w}(t_0+2) + sgn(net) \times \mathbf{x}_3 = \mathbf{w}(t_0+2) + sgn(-3) \times \mathbf{x}_3 = \mathbf{w}(t_0+2) - \mathbf{x}_3 = [1 \ -3.5 \ 4.5 \ 0.5]$

◇

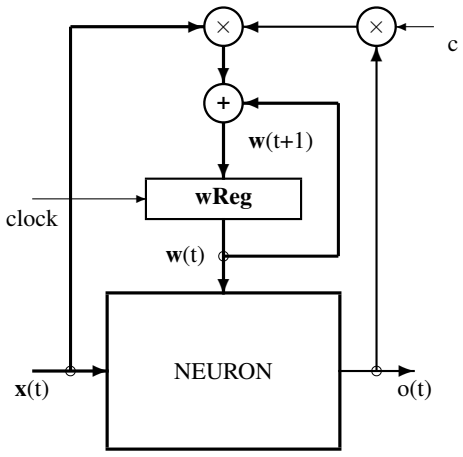

Figure 14.7: **The Hebian learning rule**

14.4.2 Supervised learning: perceptron rule

The perceptron rule performs a supervised learning. The learning is guided by the difference between the desired output and the actual output. Thus, the learning signal for each neuron is:

$$r = d - o$$

In each step the weight vector is updated (see Figure 14.8) according to the relation:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + c \times (d(t) - f(\mathbf{w}(t), \mathbf{x}(t))) \times \mathbf{x}(t)$$

The initial value for **w** does not matter.

Example 14.4 *Let be a four-input neuron with the activation function sgn. The initial weight vector is:*

$$\mathbf{w}(t_0) = [1 \ -1 \ 0 \ 0.5]$$

The training inputs are:
$\mathbf{x}_1 = [1 \ -2 \ 0 \ -1]$,
$\mathbf{x}_2 = [0 \ 1.5 \ -0.5 \ -1]$,
$\mathbf{x}_3 = [-1 \ 1 \ 0.5 \ -1]$
and the desired output for the three input vectors are: $d_1 = -1$, $d_2 = -1$, $d_3 = 1$. *The learning constant is* $c = 0.1$.

Applying by turn the three training input vectors for $c = 1$ *we obtain:*

step 1 *: because* $(d - sgn(net)) \neq 0$
$\mathbf{w}(t_0 + 1) = \mathbf{w}(t_0) + 0.1 \times (-1 + sgn(net)) \times \mathbf{x}_1 = \mathbf{w}(t_0) + 0.1 \times (-1 - 1) \times \mathbf{x}_1 = [0.8 \ -0.6 \ 0 \ 0.7]$

step 2 *: because* $(d - sgn(net) \neq 0)$ *no correction is needed in this step*
$\mathbf{w}(t_0 + 2) = \mathbf{w}(t_0 + 1)$

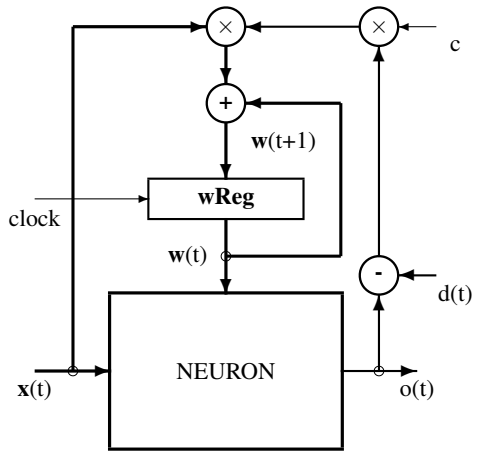

Figure 14.8: **The perceptron learning rule**

step 3 *: because* $(d - sgn(net)) = 2$
$\mathbf{w}(t_0+3) = \mathbf{w}(t_0+2) + 0.1 \times 2 \times \mathbf{x}_3 = [0.6\ -0.4\ 0.1\ 0.5]$

◇

14.5 Neural processing

NN are currently used to model complex relationships between inputs and outputs or to find patterns in streams of data. Although NN has the full power of a Universal Turing Machine (some people claim that the use of irrational values for weights results in a machine with *"super-Turing"* power), the real application of this paradigm are limited only to few functions involving specific complex memory functions (please do not use this paradigm to implement a text editor). They are grouped in the following categories:

- auto-association: the input (even a degraded input pattern) is associated to the closest stored pattern
- hetero-association: the association is made between pais of patterns; distorted input patterns are accepted
- classification: divides the input patterns into a number of classes; each class is indicated by a number (can be understood as a special case of hetero-association which returns a number)
- recognition: is a sort of classification with input patterns which do not exactly correspond to any of the patterns in the set
- generalization: is a sort of interpolation of new data applied to the input.

What is specific for this computational paradigm is that its "program" – the set of weight matrices generated in the learning process – do not provide explicit information about the functionality of the net. The content of the weight matrices can not be read and understood as we read and understand the

program performed by a conventional processor built by a register file, an ALU, ... The representation of an actual function of a NN defies any pattern based interpretation. Maybe this is the price we must pay for the complexity of the functions performed by NN.

Chapter 15

Convolutional Neural Network

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s. In the last few years, thanks to the increase in computational power, CNNs have managed to achieve superhuman performance. They are also successful at other tasks: voice recognition or natural language processing (NLP).

David H. Hubel and Torsten Wiesel performed a series of experiments on cats and monkeys starting from 1958, giving crucial insights on the structure of the visual cortex. They showed that many neurons in the visual cortex have a small *local receptive field*, meaning they react only to visual stimuli located in a limited region of the visual field. The receptive fields of different neurons may overlap, and together they tile the whole visual field. The receptive fields provide *low-level patterns*.

Some neurons have *larger receptive fields*, and they react to more complex patterns that are combinations of the lower-level patterns. We conclude that the higher-level neurons are based on the outputs of neighboring lower-level neurons. These studies of the visual cortex inspired the idea of we now call *convolutional neural networks*. The structure of this kind of network has fully *connected layers* with various activation functions, *convolutional layers*, and *pooling layers*.

15.1 Convolutional Layer

Neurons in a convolutional layer are not connected to every single pixel in the input layer but only to pixels in their receptive fields.

Rather than using neurons to look at the entire input at a time, a convolution layer "scans" the input, crossing over the entire input with a small, $k \times k$, receptive field.

Let us consider the two-dimension input plan represented in the following matrix:

$$I = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{p1} & x_{p2} & \dots & x_{pp} \end{bmatrix} \tag{15.1}$$

where x_{ij} are scalars (to make the story short and simple we considered a square matrix). For example, I represents the 8-bit pixels of one of the RGB plans associated with a color image. The image will be

scanned looking each time to a $k \times k$ *receptive field* of the following form:

$$R_{ij} = \begin{bmatrix} x_{ij} & x_{i(j+1)} & \cdots & x_{i(j+k-1)} \\ x_{(i+1)j} & x_{(i+1)(j+1)} & \cdots & x_{(i+1)(j+k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{(i+k-1)j} & x_{(i+k-1)(j+1)} & \cdots & x_{(i+k-1)(j+k-1)} \end{bmatrix} \tag{15.2}$$

Starting from the top left corner of the input plane, the first receptive field is R_{11}. Additional receptive fields are considered with a stride s horizontally and vertically:

$$i = 1, (1+s), (1+2s), \ldots, (1+((p-k)/s)s) = 1, (1+s), (1+2s), \ldots, (1+p-k)$$

$$j = 1, (1+s), (1+2s), \ldots, (1+p-k)$$

where the stride could take values $s = 1, \ldots, k$ (the stride cannot be bigger than k because the entire image must be scanned). If needed, the matrix I will be padded with zeroes to have $(p-k)/s = integer$.

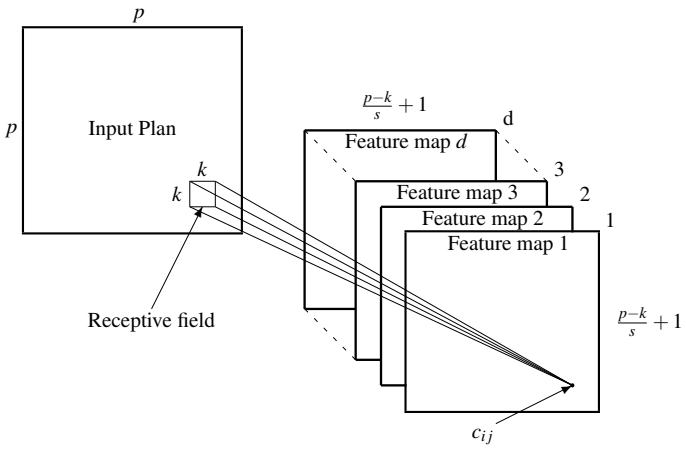

Figure 15.1: Convolution.

The neuron is the same during the scan of the entire input plan. This is called a *filter* and is defined as a matrix having the same size with the receptive field. For each input plane d filters are defined:

$$F^y = \begin{bmatrix} f^y_{11} & f^y_{12} & \cdots & f^y_{1k} \\ f^y_{21} & f^y_{22} & \cdots & f^y_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ f^y_{k1} & f^y_{k2} & \cdots & f^y_{kk} \end{bmatrix} \tag{15.3}$$

for $y = 1, 2, \ldots, d$. Each filter investigates the input plane "looking" for a specific feature, thus generating a *Feature map* (see Figure 15.1). The filter F^y applied to the receptive field R_{ij} provides c^y_{ij} where:

$$c^y_{ij} = \sum_{m=1}^{k} \sum_{l=1}^{k} f^y_{lm} \times x_{(i+l-1)(j+m-1)} \tag{15.4}$$

Thus, the application of the filter F^y with stride s provides the matrix:

$$C^y = \begin{bmatrix} c^y_{11} & c^y_{12} & \cdots & c^y_{1(((p-k)/s)+1)} \\ c^y_{21} & c^y_{22} & \cdots & c^y_{2(((p-k)/s)+1)} \\ \vdots & \vdots & \ddots & \vdots \\ c^y_{(((p-k)/s)+1)1} & c^y_{(((p-k)/s)+1)2} & \cdots & c^y_{(((p-k)/s)+1)(((p-k)/s)+1)} \end{bmatrix} \tag{15.5}$$

A convolutional layer consists of the application of d filters on the input plan generating a three dimensional array of $(((p-k)/s)+1) \times (((p-k)/s)+1) \times d$ scalars (see Figure 15.1). For each filter a feature plan is generated with a scalar for every receptive field.

15.2 Pooling layer

The pooling layer is used to reduce the size of a feature plan substituting (usually) a square *pooling window* of $q \times q$ scalars with only one scalar, which characterizes the entire pooling window. The scalar could be the maximum value from the pooling window, the sum of the values from the pooling window, or another value that is able to synthesize the content of the pooling window. The pooling windows are considered (usually) with a stride q in both directions in order to cover the entire feature plan. A stride smaller than q is possible, but it is not frequently considered.

Starting from a feature plan provided by a convolution, the pooling operation provides the pooled plan. Let us consider defining the pooling function with the same input I of $p \times p$ scalars. If the pooling window is $q \times q$ and the stride q (the usual case) the resulting plan is a $p/q \times p/q$ matrix of scalars P_q.

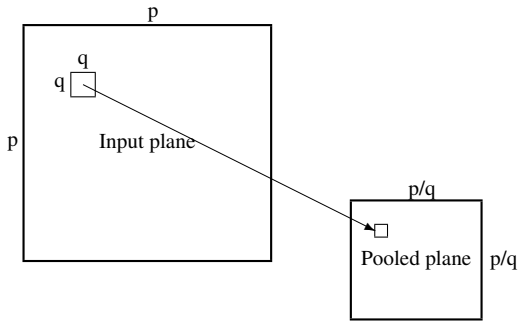

Figure 15.2: The pooling operation: starting from a $p \times p$ matrix, results in a $p/q \times p/q$ matrix.

$$P_q = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1(p/q)} \\ y_{21} & y_{22} & \cdots & y_{2(p/q)} \\ \vdots & \vdots & \ddots & \vdots \\ y_{(p/q)1} & y_{(p/q)2} & \cdots & y_{(p/q)(p/q)} \end{bmatrix} \tag{15.6}$$

where y_{ij} is computed usually in two ways (see Figure 15.3) for $q \times q$ matrices:

- by adding all the $q \times q$ values
- by taking the maximum value from the $q \times q$ values

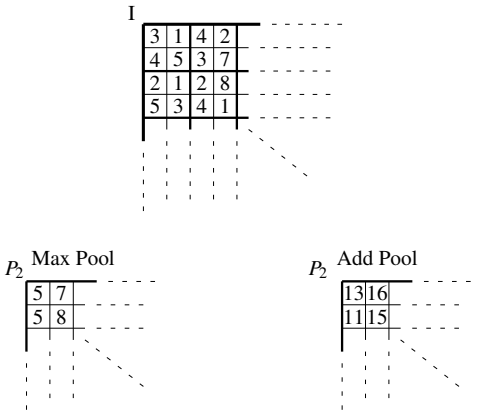

Figure 15.3: Examples of pooling for 2×2 pooling windows and stride 2. The Max Pool operation takes from the window the maximum value, while the Add Pool operation sums all the values from the window.

Pooling window of $q \times q$ scalars in the matrix I results in a scalar in the P_q matrix (see Figure 15.2)

In current applications, the value of q is 2 or 4. In Figure 15.3 two examples for $q = 2$ are presented. One with *Max* as pooling function, and another with *Add* as pooling function. Each 2×2 matrix of scalars is substituted with their maximum or their sum.

15.3 Softmax layer

The softmax layer is used for multi-category classification, in order to emphasise the most probable candidate as result. It is applied to a n-component vector $V = \langle x_1, x_2, \ldots, x_n \rangle$. Its value is determined by the standard exponential function on each component, divided by the sum of the exponential function applied to each component, as a normalizing constant. Therefore, the output components sum to 1:

$$\sigma_i(V) = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}} \tag{15.7}$$

Results:

$$S_i(V) = \langle \sigma_1(V), \sigma_2(V), \ldots, \sigma_n(V) \rangle \tag{15.8}$$

In [**?**] the computation is simplified by avoiding the divide operation and by reducing the domain of the exponent. The first step is to down-scale the exponentiation:

$$\sigma_i(V) = \frac{e^{x_i}/e^{x_{max}}}{(\sum_{i=1}^{n} e^{x_i})/e^{x_{max}}} = \frac{e^{x_i - x_{max}}}{\sum_{i=1}^{n} e^{x_i - x_{max}}} \tag{15.9}$$

The second step is to compute the natural logarithm:

$$ln(\sigma_i(V)) = (x_i - x_{max}) - ln(\sum_{i=1}^{n} e^{x_i - x_{max}}) \tag{15.10}$$

While the sum in Expression 15.7 is susceptible to overflow because the values generated by exponentiation are high, and the divide operation is resource and time consuming, the Expression 15.9 works with smaller numbers and avoids the division. Both, ln and exp operations are performed using LUTs.

15.4 Putting all together

An example of DNN is shown in Figure 15.4, where all the previously presented functions are used to define a particular network. The network has 5 layers, hence named Lenet-5: three sets of convolution layers with a combination of average pooling and two fully connected layers. At last, a Softmax classifier which classifies the images.

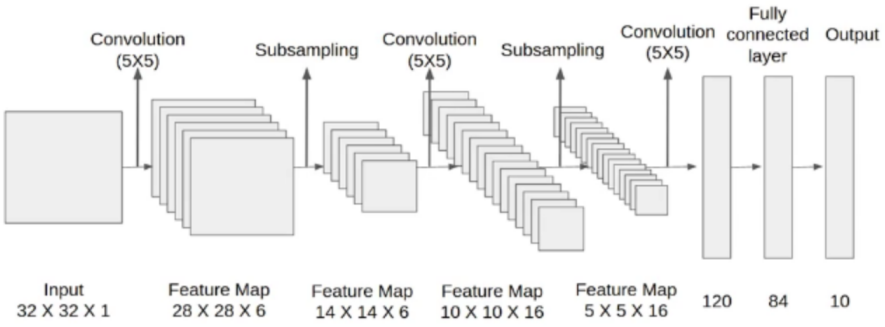

Figure 15.4: An example of DNN: Lenet-5 proposed by Yann LeCun [28] [29] [40].

While the first convolutions are used to inspect the input to identify specific *local* features, the last fully connected layers provide a *global* analysis, and the softmax output layer emphasizes the most probable result. The Lenet-5 has the following characteristics on each of its layers:

input with one channel of 32×32 gray-scale image

first convolution operation with 6 filter of size 5×5. Therefore, the feature map is $28 \times 28 \times 6$ because the padding is not used (the number of channels is equal to the number of filters applied).

average pooling with stride 2 and the size of the feature map is reduced by half. Note that, the number of channels is intact.

second convolution layer with 16 filters of size 5×5.

average pooling with stride 2 reduces the size of the feature map by half i.e $5 \times 5 \times 16$

third convolution layer of size 5×5 with 120 filters leaving the feature map size $1 \times 1 \times 120$, i.e., 120 values

first fully connected layer with 84 neurons

second fully connected layer with 10 neurons with activation function softmax, unlike the others layers which have tanh activation function.

The number of trainable parameters is 60000.

Chapter 16

Recurrent Neural Network

In this chapter, we are going to discuss about *recurrent neural networks* (RNN). RNN is a class of neural networks that can predict, up to a point, the future. They can analyze *time series data* (for example stock prices)g. They can anticipate car trajectories and help avoid accidents. They can work on sequences of arbitrary lengths, rather than on fixed-sized inputs like all the nets we have discussed so far.

16.1 Recurrent Neural Network Structure

16.1.1 Recurrent Neuron

Unlike the neuron used until now, described by

$$y_{(t)} = \phi(\mathbf{x}_{(t)}^T \cdot \mathbf{w}_x + b)$$

the recurrent neuron is described by [24]:

$$y_{(t)} = \phi(\mathbf{x}_{(t)}^T \cdot \mathbf{w}_x + \mathbf{y}_{(t-1)}^T \cdot \mathbf{w}_y + b)$$

because of the loop closed from its output, $\mathbf{y}$, to its input which now is a concatenation between the current input, $\mathbf{x}_{(t)}$, and the output generated in the previous cycle, $\mathbf{y}_{(t-1)}$.

16.1.2 Recurrent Layer of Neurons

For a layer of recurrent neurons we write:

$$\mathbf{Y}_{(t)} = \phi(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b}) = \phi([\mathbf{X}_{(t)} \;\; \mathbf{Y}_{(t-1)}] \cdot \mathbf{W} + \mathbf{b})$$

with:

$$\mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}$$

where:

$\mathbf{Y}_{(t)}$ is an $m \times n_{neurons}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and $n_{neurons}$ is the number of neurons).

$\mathbf{X}_{(t)}$ is an $m \times n_{inputs}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).

$\mathbf{W}_x$ is an $n_{inputs} \times n_{neurons}$ matrix containing the connection weights for the inputs of the current time step.

$\mathbf{W}_y$ is an $n_{neurons} \times n_{neurons}$ matrix containing the connection weights for the outputs of the previous time step. The weight matrices $\mathbf{W}_x$ and $\mathbf{W}_y$ are often concatenated into a single weight matrix $\mathbf{W}$ of shape $(n_{inputs} + n_{neurons}) \times n_{neurons}$.

$\mathbf{b}$ is a vector of size $n_{neurons}$ containing each neuron's bias term.

This layer of neurons has the output $\mathbf{y}_{(t)}$ depending on the inputs from the moment $t = 0$. Likewise, output behavior, with constant input, can predict subsequent behaviors.

16.1.3 Internal State of a Cell

A cell's internal state at time t is

$$h_(t) = f(h_{(t-1)}, x_t)$$

while the output at the step t is

$$y_(t) = g(h_{(t-1)}, x_t)$$

which means in each cell there is a memory for the value of $h_(t)$.

For a layer of recurrent neurons the internal state is a *vector* $H_{(t)}$ and evolves triggered by the *sequence* of input vectors, $X_{(0)}, X_{(1)}, \ldots X_{(i)}, \ldots$ according to:

$$H_(t) = f(H_{(t-1)}, X_t)$$

while the output vector $Y_{(t)}$ evolves triggered by the *sequence* of input vectors, $X_{(0)}, X_{(1)}, \ldots X_{(i)}, \ldots$ according to:

$$Y_(t) = g(H_{(t-1)}, X_t)$$

generating the sequence of vectors $Y_{(0)}, Y_{(1)}, \ldots Y_{(i)}, \ldots$.

16.2 Operation Modes

16.2.1 Sequence of vectors to sequence of vectors

The network is fed by a sequence of vectors and provides a sequence of vectors. For example in predicting time series such as stock prices.

$$X_{(0)}, X_{(1)}, \ldots X_{(i)} \Rightarrow Y_{(0)}, Y_{(1)}, \ldots Y_{(i)}$$

16.2.2 Sequence of vectors to vector

The network is fed by a sequence of vectors, for example key words associated to a process. The network "accumulates" the information and provides a "score" associated to the process. The outputs of the network are ignored from $t = 0$ to $t = i - 1$.

$$X_{(0)}, X_{(1)}, \ldots X_{(i)} \Rightarrow Y_{(i)}$$

16.2.3 Vector to sequence of vectors

The network is triggered by an input vector and is let to evolve a number of cycles generating a description for $X_{(0)}$.

$$X_{(0)} \Rightarrow Y_{(0)}, Y_{(1)}, \ldots Y_{(i)}$$

16.2.4 Delayed sequence of vectors to sequence of vectors

The network is triggered as a sequence-to-vector network, performing the operation called *encoder*, followed by a vector-to-sequence network, performing the operation called *decoder*. For example, this mode is used for translating a sentence from one language to another.

$$X_{(0)}, X_{(1)}, \ldots X_{(i)} \Rightarrow Y_{(i+1)}, Y_{(i+2)}, \ldots Y_{(i+j)}$$

The output is ignored for the first cycles in the process of *encoding*, then the input is ignored for the last cycles in the process of *decoding*.

Chapter 17

Autoencoders

An autoencoder looks at the inputs, converts them to an efficient internal representation, and then generates a representation that looks very close to the inputs. An autoencoder is always composed of two parts: an *encoder*, which is a recognition network that converts the inputs to an internal representation, followed by a *decoder* which is a generative network that converts the internal representation and outputs it.

The output are often called the *reconstruction* since the autoencoder tries to reconstruct the input, and the cost function contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.

The internal representation has a lower dimensionality than the input data (it is 2D instead of 3D). Thus the autoencoder is said to be *undercomplete*. An autoencoder cannot simply copy its inputs to the codings. It must find a way to output a copy of its inputs. Therefore, it is forced to learn the most important features in the input data and drop the unimportant ones.

Chapter 18

Reinforcement Learning

Part VI

ANNEXES

Appendix A

History

The history of computation consists of few independent threads, starting in Antiquity. The history starts with an imaginary thread, a conceptual thread and a factual thread. Initially, the concepts and the objects evolved independently. At their mature stage, stimulated by the sad event of World War II, the conceptual evolution interferes with the physical implementation and the IT era begins. Application-driven history gradually emerges around 1971 when the conceptual approach reaches a maturity that slows down the theoretical approaches. In parallel with these threads, along the history, has been manifested and it still manifests also an imaginary thread. One of the main driving force in any domain is the human will and imagination. Therefore, we cannot ignore an ever developing imaginary history of the computing technology and its applications.

A.1 Imaginary history

At the beginning is always an image, a dream.

A.1.1 Antiquity

Hephaestus & Vulcan

Greek god Hephaestus is the god of technology, blacksmiths, craftsmen and artisans. Hephaestus made a bronze giant called *Talos* who protected Crete from pirates and invaders. It would patrol around the island and throw rocks at enemy ships.

A roman counterpart of Hephaestus is Vulcan: made slave-girls of gold for himself.

Pygmalion

Pygmalion was a mythical character who, in search of perfection, sculpted in ivory the image of a perfect woman with whom he later fell in love and the goddess Aphrodite gave life to the statue.

We are always dealing with the human being's aspiration to correct the imperfections of nature through artifacts. Pygmalion seems to be a transhumanist *avant la lettre*.

A.1.2 Middle Ages

There were several stories and legends in the Middle Ages that involved the creation of artificial beings or creatures. These stories often reflected the beliefs and fears of medieval society regarding the power

of human beings to create and control life.

Golem

Golem (Prague, ∼1500) is a metaphor for a mindless entity that serves man under controlled conditions, but is hostile to him under certain conditions. The earliest known written account of how a golem is created can be found in Jewish tradition. The most famous golem narrative involves Judah Loew ben Bezalel, the late 16th-century rabbi of Prague.

Artificial animals and creatures at the court of Emperor Frederick II

There were also stories of artificial animals and creatures, such as the legendary mechanical eagle of Holy Roman Emperor Frederick II (1194-1250), which was said to be able to fly and to emit various sounds and cries.

Brazen Head

Another famous example is the story of the Brazen Head, a mechanical or artificial head that was said to be able to answer any question put to it. According to legend, the head was created by the medieval scholar and philosopher Roger Bacon (1220-1292), who was said to have used his knowledge of natural philosophy to imbue the head with the power of speech and reason.

Homunculus

Another example is the legend of the Homunculus, a tiny human-like creature that was said to be created by alchemists through the use of special substances and rituals. The Homunculus was believed to possess magical powers and to be able to perform various tasks, including the transmutation of metals and the creation of life. Paracelsus (1493–1541) is credited with the first mention of Homunculus in *De homunculis* (c. 1529–1532), and *De natura rerum* (1537)

Overall, these stories and legends reflected the fascination and curiosity of medieval society with the idea of artificial life and the power of human beings to create and control it.

A.1.3 Modernity

Frankenstein‘s Creature

Mary Shelley (1797-1851, wife of the poet Percy Shelley and daughter of Mary Wollstonecraft a founding figure of feminism) published in 1818 the novel *Frankenstein* about a brilliant but unorthodox scientist, Dr. Victor Frankenstein, who rejects the artificial man he created; Creature escapes and later swears revenge.

Offenbach’s Olympia

Jacques Offenbach (1819-1880) in his *The Tales of Hoffmann* opera finished in 1880 introduced the character *Olympia*, a mechanical or an animatronical doll.

Karel Capek's Robota

Robot (*robota* in Russian) is coined by Karel Capek 1920 R.U.R. is a 1920 science fiction play by the Czech writer Karel Čapek. R.U.R. stands for *Rossumovi Univerzální Roboti* (Rossum's Universal Robots). The English phrase "Rossum's Universal Robots" has been used as a subtitle.

Fritz Lang's Metropolis

In 1927, German film maker Fritz Lang made the science fiction film Metropolis. The script contains the construction of a robot that acquires perfect human appearance and behavior. The artificial product has the ability to disrupt the behavior of the masses. It's far beyond what was imagined for Capek's robot.

A.1.4 Contemporary

Scary Science Fiction (SF) scenarios about Artificial Intelligence (AI) [44].

Max Tegmark: Life 1.0 referring to biological origins, Life 2.0 referring to cultural developments in humanity, and Life 3.0 referring to the technological age of humans.

We must make distinctions between the three main human brain behaviors: Spiritually – Imaginary – Rationally. AI refers mainly to the third.

A.2 Conceptual history

A.2.1 Binary Arithmetic to the Chinese

In *Discourse on the Natural Theology of the Chinese*, Gottfried Wilhelm von Leibniz (1646-1716) mentioned that the 64 hexagrams of *I Ching* ($\sim$1000 BC) represent the binary arithmetic used a few thousand years ago in China. Leibnitz consider the mythical King and philosopher Fuxi, who is believed to have lived more than 4000 years ago, as the inventor of binary notation.

Binar	Trigram	Meaning
111	☰	Heaven, creativity
000	☷	Earth, receptivity
010	☵	Water, dabger
101	☲	Fire, clarity
100	☶	Mountain, stillness
110	☴	Wind/wood, flexibility
011	☱	Lake, joy
001	☳	Thunder, movement

Figure A.1: Figure of the Eight Cova attributed to Fuxi.

Fuxi is considered the originator of the methods of divination that were passed down through the ages before the *I Ching*.

Figure A.2: Guo Xu (1456-1529) depicts Fuxi as he looks at the binary symbols he invented (dated 1503).

A.2.2 Programming & Algorithms in Babylon [49]

Floating-point notation

In Babylonia the base-60 number system is introduced. The sequels of this system can be found in the 60 minutes of the hour, the 60 degrees of the angles of the equilateral triangle, ...

Instructions to describe to compute

Descriptions were found on the cuneiform tablets in the form of instructions for making certain calculations.

A.2.3 Epimenides of Crete (late 7th century - 6th century BC

Karl Jaspers (1883–1969) introduced the concept of an Axial Age in his book *The Origin and Goal of History*, published in 1949. It refers to broad changes in religious and philosophical thought that occurred in a variety of locations from about the 8th to the 3rd century BCE. During this period new ways of thinking emerged in Persia, India, China, Greece and Roman Empire, in a singular synchronous development, without any effective direct cultural contact between all of the Eurasian cultures. Jaspers emphasized prominent thinkers from this period who had a profound influence on future sciences, philosophies and religions.

In this Axial Age, around 7th or 6th century BC, Epimenides of Cnossos (Crete) was a semi-mythical Greek seer and philosopher-poet which started the conceptual development leading to the contemporary computer science. In one day he uttered a sentence which troubled the inquisitive minds from everywhere for the next two and half millennia:

"Cretans, always liars."

The sentence is equivalent with *"I lie"*, and is undecidable: its truth value can not be decided.

A.2.4 Liar's paradox in Middle Ages

In the Middle Ages, the paradox was studied and commented on by many philosophers and theologians, who tried to resolve the contradiction it presents.

Anicius Manlius Severinus Boethius (∼480-524)

Boethius is one of the earliest recorded commentaries on the paradox in the Middle Ages. He is a Roman philosopher who lived in the 6th century CE. In his work "Consolation of Philosophy," Boethius discusses the paradox and argues that it arises from a confusion of terms and concepts.

Peter Abelard (1079-1142)

In the 12th century, the paradox was further discussed by the French philosopher and theologian Peter Abelard, who used it to criticize the doctrine of divine omnipotence. Abelard argued that the paradox shows that there are limits to what even an omnipotent God can do, since he cannot make a statement that is both true and false at the same time.

William of Ockham (1285-1347)

In the 14th century, the English logician William of Ockham used the paradox to argue against the idea of universal propositions. Ockham argued that the paradox shows that there are no universal propositions that can be true or false in all cases, since there are some statements that cannot be consistently evaluated as true or false.

Overall, the paradox of the liar was a topic of interest and debate among medieval philosophers and theologians, who used it to explore the limits of language, logic, and the nature of truth.

A.2.5 Mohammed Al-Khoresmi (780-850)

Active in the city of Baghdad, not far from the ancient city of Babylon, Al-Khoresmi works as a court mathematician around the year 800.

Algebra

His work *Kitâ al-jabr wa'l-muqabâla* gives the name to the mathematical discipline algebra.

Decimal notation

The lost work, translated into Latin under the name *Algorithmi de numero Indiorum* introduces the positional number system we use today: a base-10 system including a symbol for zero.

Algorithm

Al-Khoresmi's name also gave the currently used name, that of *algorithm*, for the procedure associated with a sequence of operations that describes a computation.

A.2.6 Gottfried Wilhelm von Leibniz

Binary representation

In 1703, Leibniz published in the *Mémories de l'Académie Royale des Sciences* his essay "Explication de l'arithmétique binaire, qui se sert des seules caractères 0 & 1; avec des remarques sur son utilité, et sur ce qu'elle donne de sens des anciennes figures chinoises de Fohy" [30] where he explains how to perform addition, subtraction, multiplication and division using the binary representation for numbers.

Calculus ratiocinator

The *Calculus ratiocinator* is a a concept introduced by Leibniz related to *characteristica universalis*, an universal conceptual language. This concept could be related to both the hardware and software aspects of the modern digital computer.

A.2.7 George Boole

In 1847 George Boole (1815-1864) published *Mathematical Analysis of Logic* and in 1854 *An Investigation into the Laws of Thought, on which are Founded the Mathematical Theories of Logic and Probabilities* which underpins what we now call *Boolean algebra*, a successful attempt to formalize Aristotelian logic. It is thus made available to innovators an instrument that will be used to substantiate the science of calculus as a ***decision*** tool in the first place, and only through a second approach as a calculation tool. Indeed, computation is mainly about deciding. Numerical computation comes only as a consequence. At first it was the true/false alternative, and only then the 0/1 alternative.

A.2.8 1900-1928: David Hilbert

David Hilbert (1862-1943) one of the most influential and universal mathematicians of the 19th and early 20th centuries.

At the International Congress of Mathematicians held in Bologna, Hilbert revisited to the second of the 23 problems posed in his 1900 paper *Mathematische Probleme* [21], asking [23]:

1. Was its set of rules complete, so that any statement could be proved (or disproved) using only the rules of the system?

2. Was it consistent, so that no statement could be proved true and also proved false?

3. Was there some procedure that could determine whether a particular statement was provable, rather than allowing the possibility that some statements (such as enduring math riddles like Fermat's last theorem, Goldbach's conjecture, or the Collatz conjecture) were destined to remain in undecidable limbo?

Hilbert thought that the answer to the first two questions was yes, making the third one moot [23].

In mathematics and computer science, the *Entscheidungsproblem* (German for "decision problem") is a challenge posed by David Hilbert and Wilhelm Ackermann in 1928 [22]. By the completeness theorem of first-order logic, a statement is universally valid if and only if it can be deduced from the axioms, so the Entscheidungsproblem can also be viewed as asking for an algorithm to decide whether a given statement is provable from the axioms using the rules of logic.

As late as 1930, Hilbert believed that there would be no such thing as an unsolvable problem.

The *Entscheidungsproblem* is related to Hilbert's tenth problem (from Hilbert's address of 1900 to the International Congress of Mathematicians in Paris [21]), which asks for an algorithm to decide whether Diophantine equations have a solution. The non-existence of such an algorithm, established by Yuri Matiyasevich in 1970, also implies a negative answer to the *Entscheidungsproblem*.

Hilbert's address of 1900 to the International Congress of Mathematicians in Paris is perhaps the most influential speech ever given to mathematicians, given by a mathematician, or given about mathematics. In it, Hilbert outlined 23 major mathematical problems to be studied in the coming century.

A.2.9 1931: Kurt Gödel

Kurt Friedrich Gödel (1906-1978) The *logician* Gödel published his two incompleteness theorems in 1931 when he was 25 years old, one year after finishing his doctorate at the University of Vienna. The first incompleteness theorem states that for any self-consistent recursive axiomatic system powerful enough to describe the arithmetic of the natural numbers (for example Peano arithmetic), there are true propositions about the naturals that cannot be proved from the axioms. To prove this theorem, Gödel developed a technique now known as Gödel numbering, which codes formal expressions as natural numbers.

The Austrian-born logician Kurt Gödel polished off **the first two** Hilbert's questions with unexpected answers: no and no. In his "incompleteness theorem", he showed that there existed statements that could be neither proved nor disproved.

A.2.10 1936: Church – Kleene – Post – Turing

What a synchronicity! Indeed, the logician Gödel's approach triggered four *mathematicians* to provide independently mathematical versions to the logical challenge raised by the *Entscheidungsproblem* (the third of Hilbert's questions).

Alonzo Church

Alonzo Church (1903-1995): The lambda calculus emerged in his 1936 paper showing the unsolvability of the *Entscheidungsproblem*. This result preceded Alan Turing's work on the halting problem, which also demonstrated the existence of a problem unsolvable by mechanical means. Church and Turing then showed that the lambda calculus and the Turing machine used in Turing's halting problem were equivalent in capabilities, and subsequently demonstrated a variety of alternative "mechanical processes for computation". This resulted in the Church–Turing thesis.

The lambda calculus influenced the design of the LISP programming language and functional programming languages in general.

Stephen Kleene

Stephen Cole Kleene (1909-1994) is best known as a founder of the branch of mathematical logic known as recursion theory, which subsequently helped to provide the foundations of theoretical computer science.

Emil Post

Emil Leon Post (1897-1957) developed in 1936, independently of Alan Turing, a mathematical model of computation that was essentially equivalent to the Turing machine model. This model is sometimes

called ”Post’s machine” or a Post–Turing machine.

Alan Turing

“When the great Cambridge math professor Max Newman taught Turing about Hilbert’s questions, the way he expressed the *Entscheidungsproblem* was this: Is there a “mechanical process” that can be used to determine whether a particular logical statement is provable?” [23]

Alan Mathison Turing (1912-1954) in 1936 published his paper ”On Computable Numbers, with an Application to the *Entscheidungsproblem*”. It was published in the Proceedings of the London Mathematical Society journal in two parts, the first on 30 November and the second on 23 December. In this paper, Turing reformulated Kurt Gödel’s 1931 results on the limits of proof and computation, replacing Gödel’s universal arithmetic-based formal language with the formal and simple hypothetical devices that became known as Turing machines. The *Entscheidungsproblem* (decision problem) was originally posed by German mathematician David Hilbert in 1928. Turing proved that his ”universal computing machine” would be capable of performing any conceivable mathematical computation if it were representable as an algorithm. He went on to prove that there was no solution to the decision problem by first showing that the halting problem for Turing machines is undecidable: It is not possible to decide algorithmically whether a Turing machine will ever halt.

A.2.11 1940s: abstract models of computation

The transition from a mathematical model of computation to a realizable physical structure was enabled by the abstract models of computers. Purely mathematical models contain descriptions that assume concepts that have no physical counterpart, such as infinity. For this reason abstract models were necessary.

1943: Neural nets

Warren S. McCulloch, Walter H. Pitts introduced the neural network model for computation [38].

1944: Harvard abstract model

The term originated from the Harvard Mark I, or IBM Automatic Sequence Controlled Calculator (ASCC), an electromechanical computer, which stored instructions on punched tape and data in electromechanical counters.

1945: von Neumann abstract model

John von Neumann wrote up a description titled *First Draft of a Report on the EDVAC* [46] based on the work of Eckert and Mauchly. It was unfinished when his colleague Herman Goldstine circulated it, and bore only von Neumann’s name (to the consternation of Eckert and Mauchly).

A.3 Factual history

A.3.1 Antikythera mechanism

The Antikythera mechanism is believed to be designed to predict eclipses. It has been designed and constructed by Greeks and is dated to about 200 BC to 80 BC. It is a clockwork mechanism composed of more than 30 engaged bronze gears.

A.3.2 Hero of Alexandria

Hero of Alexandria (c.10-c.70) was the first to build a vending machine; when a coin was introduced via a slot on the top of the machine, a set amount of holy water was dispensed.

Hero described the construction of the aeolipile (a version of which is known as Hero's engine) which was a rocket-like reaction engine and the first-recorded steam engine

A.3.3 Gerbert of Aurillac

In 996 A.D., Gerbert of Aurillac (Pope Sylvester II from 999) (946-1003) invented the first weight-driven mechanical pendulum clock at a monastery in Magdeburg in Germany. The clock's mechanism would ring bells at regular intervals throughout the day to call his fellow monks to prayer.

Gerbert took the idea of the abacus calculator from a Spanish Arab. But the calculations with his abacus were extremely difficult, because the people of his day used only Roman numerals.

A.3.4 Wilhelm Schickard

Johannes Kepler, claimed that the drawings of a calculating clock, predating the public release of Pascal's calculator by twenty years, had been discovered in two unknown letters written by Wilhelm Schickard (1592-1635) to him in 1623 and 1624.

A.3.5 Blaise Pascal

Pascaline: Blaise Pascal (1623-1662) was led to develop a calculator by the laborious arithmetical calculations required by his father's work as the supervisor of taxes in Rouen. He designed the machine to add and subtract two numbers directly and to perform multiplication and division through repeated addition or subtraction.

A.3.6 Gottfried Wilhelm von Leibniz

In *Machina arithmetica in qua non additio tantum et subtractio sed et multiplicatio nullo, diviso vero paene nullo animi labore peragantur*, written in 1685, Gottfried Wilhelm (von) Leibniz (1646-1716) described an arithmetic machine he had invented that was made by linking two separate machines, one to perform additions/subtractions and one for multiplications/divisions.

A.3.7 Joseph Marie Charles *dit* Jacquard

The Joseph Jacquard (1752-1834) Loom is a mechanical loom that uses pasteboard cards with punched holes, each card corresponding to one row of the design. Multiple rows of holes are punched in the cards and the many cards that compose the design of the textile are strung together in order.

A.3.8 Charles Babbage

Difference engine

Charles Babbage (1791-1871) began in 1822 with what he called the difference engine, made to compute values of polynomial functions. It was created to calculate a series of values automatically. By using the method of finite differences, it was possible to avoid the need for multiplication and division.

Analytical Engine

The Analytical Engine marks the transition from mechanised arithmetic to fully-fledged general purpose computation. It is largely on it that Babbage's standing as computer pioneer rests.

The major innovation was that the Analytical Engine was to be programmed using punched cards: the Engine was intended to use loops of Jacquard's punched cards to control a mechanical calculator, which could use as input the results of preceding computations.[157][158] The machine was also intended to employ several features subsequently used in modern computers, including sequential control, branching and looping. It would have been the first mechanical device to be, in principle, Turing-complete.

A.3.9 Ada Byron, Countess of Lovelace

Augusta Ada King, Countess of Lovelace (née Byron; 1815–1852) chiefly known for her work on Charles Babbage's proposed mechanical general-purpose computer, the Analytical Engine. She was the first to recognise that the machine had applications beyond pure calculation, and published the first algorithm intended to be carried out by such a machine. As a result, she is sometimes regarded as the first to recognise the full potential of a "computing machine" and one of the first computer programmers.

A.3.10 Herman Hollerith

Herman Hollerith (1860-1929) developed an electromechanical tabulating machine for punched cards to assist in summarizing information and, later, in accounting. His invention of the punched card tabulating machine, patented in 1889, marks the beginning of the era of semiautomatic data processing systems, and his concept dominated that landscape for nearly a century. He was the founder of the Tabulating Machine Company that was amalgamated (via stock acquisition) in 1911 with three other companies to form a fifth company, the Computing-Tabulating-Recording Company, which was renamed IBM in 1924. Hollerith is regarded as one of the seminal figures in the development of data processing.

A.3.11 Claude Shannon & Thomas Flowers

Implementing electro-mechanically Boolean functions

Claude Elwood Shannon (1916-2001) known as "the father of information theory". Shannon is noted for having founded information theory with a landmark paper, *A Mathematical Theory of Communication*, that he published in 1948.

He is also well known for founding digital circuit design theory in 1937, when -— as a 21-year-old master's degree student at the Massachusetts Institute of Technology (MIT) -— he wrote his thesis demonstrating that electrical applications of Boolean algebra could construct any logical numerical relationship.

Implementing electronically Boolean functions

Thomas Harold Flowers (1905-1998). From 1935 onward, he explored the use of electronics for telephone exchanges and by 1939, he was convinced that an all-electronic system was possible. A background in switching electronics would prove crucial for his computer designs [13].

A.4 Merged history

The triad *Math & Logic – War – Technology* (Ethos – Pathos – Logos) provides the context of the emergence of the Information Technology (IT) era.

WWII made the turbulent transition toward IT industry.

A.4.1 Konrad Zuse (1910-1995)

Z1 to Z4 electro-mechanical first programmable computers

Plankalkül first high-level programming language

A.4.2 Colossus

Colossus was a set of computers developed by British code-breakers in the years 1943–1945 to help in the cryptanalysis of the Lorenz cipher. Colossus used thermionic valves (vacuum tubes) to perform Boolean and counting operations. Colossus is thus regarded as the world's first programmable, electronic, digital computer, although it was programmed by switches and plugs and not by a stored program.

A Colossus computer was thus not a fully Turing complete machine. The notion of a computer as a general purpose machine — that is, as more than a calculator devoted to solving difficult but specific problems — did not become prominent until after World War II.

A.4.3 ENIAC – EDVAC

ENIAC

Electronic Numerical Integrator and Computer was the first electronic general-purpose computer. It was Turing-complete, digital and able to solve "a large class of numerical problems" through reprogramming.

ENIAC was completed in 1945 and first put to work for practical purposes on December 10, 1945. ENIAC was designed by *John Mauchly* and *J. Presper Eckert* of the University of Pennsylvania, U.S.

By the end of its operation in 1956, ENIAC contained 20,000 vacuum tubes; 7,200 crystal diodes; 1,500 relays; 70,000 resistors; 10,000 capacitors; and approximately 5,000,000 hand-soldered joints. It weighed more than 27 t, was roughly $2.4m \times 0.9m \times 30m$ in size, occupied $167m^2$ and consumed $150kW$ of electricity.

EDVAC

Electronic Discrete Variable Automatic Computer: unlike its predecessor, the ENIAC, it was binary rather than decimal, and was designed to be a stored-program computer. Functionally, EDVAC was a binary serial computer with automatic addition, subtraction, multiplication, programmed division and automatic checking with an ultrasonic serial memory[1] capacity of 1,000 34-bit words. EDVAC's average addition time was 864 microseconds and its average multiplication time was 2,900 microseconds.

ENIAC inventors John Mauchly and J. Presper Eckert proposed EDVAC's construction in August 1944, and design work for EDVAC commenced before ENIAC was fully operational. The design would implement a number of important architectural and logical improvements conceived during the ENIAC's construction and would incorporate a high-speed serial-access memory. Like the ENIAC, the EDVAC was built for the U.S. Army's Ballistics Research Laboratory at the Aberdeen Proving Ground by the University of Pennsylvania's Moore School of Electrical Engineering. Eckert and Mauchly and the other

ENIAC designers were joined by John von Neumann in a consulting role; von Neumann summarized and discussed logical design developments in the 1945 *First Draft of a Report on the EDVAC*.

A.4.4 Princeton computer

The IAS machine was the first electronic computer to be built at the Institute for Advanced Study (IAS) in Princeton, New Jersey. It is sometimes called the von Neumann machine, since the paper describing its design was edited by John von Neumann, a mathematics professor at both Princeton University and IAS. The computer was built from late 1945 until 1951 under his direction.

The IAS machine was a binary computer with a 40-bit word, storing two 20-bit instructions in each word. The memory was 1,024 words (5.1 kilobytes). Negative numbers were represented in "two's complement" format. It had two general-purpose registers available: the Accumulator (AC) and Multiplier/Quotient (MQ). It used 1,700 vacuum tubes. The memory was originally designed for about 2,300 RCA Selectron vacuum tubes.

It weighed about 1,000 pounds (450 kg).[11]

It was an asynchronous machine, meaning that there was no central clock regulating the timing of the instructions. One instruction started executing when the previous one finished. The addition time was 62 microseconds and the multiplication time was 713 microseconds.

A.4.5 IBM entered the scene

The IBM 701 Electronic Data Processing Machine, known as the Defense Calculator while in development, was IBM's first commercial scientific computer, which was announced to the public on April 29, 1952. It was designed by Nathaniel Rochester and based on the IAS machine at Princeton.

A.4.6 John Backus: first involvement

John Warner Backus (1924-2007): directed the team that invented and implemented FORTRAN, the first widely used high-level programming language, and was the inventor of the Backus–Naur form (BNF), a widely used notation to define formal language syntax.

A.4.7 LISP language & Artificial Intelligence

LISP (LISt Processor) language is the language of AI. Lisp was originally created influenced by the notation of Alonzo Church's lambda calculus. Data structure and source code are lists in LISP. John McCarthy began developing Lisp in 1958 while he was at the Massachusetts Institute of Technology.

Alan Turing was the first to conduct substantial research in the field that he called artificial intelligence machine intelligence. The field went through multiple cycles of optimism followed by disappointment and loss of funding.

There have been two well-known "AI winters" in the history of artificial intelligence:

First AI Winter (1970s-1980s): The term "AI winter" was first coined during this period. It occurred due to overhyped expectations and unfulfilled promises of AI capabilities. Funding for AI research was reduced as progress did not meet the ambitious goals set earlier.

Second AI Winter (late 1980s-early 1990s): Another AI winter followed as a result of similar issues—high expectations that were not met, coupled with limited progress in AI research. Funding decreased, and interest in AI waned during this time.

It's worth noting that while these periods were challenging for the field of AI, they also led to valuable lessons and paved the way for more realistic expectations and sustainable progress in subsequent years. The field has seen a resurgence and significant advancements since the late 1990s, and AI is now a rapidly evolving and influential area of technology. [ChatGPT]

A.4.8 Smalltalk language

A.4.9 Prolog language

A.4.10 Pyton language

A.4.11 Computer architecture

Brooks went on to help develop the IBM System/360 (now called the IBM zSeries) line of computers, in which "architecture" became a noun defining "what the user needs to know".

In [**?**] [5] the concept of computer architecture (low level machine model) is introduced to allow independent evolution for the two different aspects of computer design, which have different rate of evolution: software and hardware; thus, there are now on the market few stable and successful architectures, such as x86, ARM, PowerPC.

A.4.12 John Backus: second involvement

He later did research into the function-level programming paradigm, presenting his findings in his influential 1977 Turing Award lecture "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs"

A.4.13 Parallel computing enter the scene on the back door

While for mono-core computing there are the following stages [**?**]:

- **1936 – mathematical computational models**: four equivalent models are published [**?**] [12] [26] [36] (all reprinted in [**?**]), out of which the *Turing Machine* offered the most expressive and technologically appropriate suggestion for future developments leading eventually to the mono-core, sequential computing

- **1944-45 – abstract machine models**: the MARK 1 computer, built by IBM for Harvard University, consecrated the *Harvard abstract model*, while the von Neumann's report [46] introduced the *von Neumann abstract model*; these two concepts backed the *RAM* (random access machine) abstract model used to evaluate algorithms for sequential machines

- **1952 – manufacturing in quantity**: IBM launched *IBM 701*, the first large-scale electronic computer

- **late 1953 – high-level programming language**: John W. Backus submitted a proposal to his superiors at IBM to develop a more practical alternative to assembly language for programming their IBM 704 mainframe computer; a draft specification for "The IBM Mathematical Formula Translating System" was completed by November 1954; the first manual for FORTRAN appeared in October 1956; with the first FORTRAN compiler delivered in April 1957.

- **1964 – computer architecture**: in [5] the concept of *computer architecture* (low level machine model) is introduced to allow the independent development for the two different aspects of computer design which have different rate of evolution: software and hardware; thus, there are now on the market few stable and successful architectures, such as x86, ARM, ….

for parallel computing we are faced with a completely distorted evolution; let us see its first stages:

- **1962 – manufacturing in quantity**: the first symmetrical MIMD engine is introduced on the computer market by Burroughs
- **1965 – architectural issues**: Edsger W. Dijkstra formulates in [15] the first concerns about specific parallel programming issues
- **1974-76 – abstract machine models**: proposals of the first abstract models (bit vector models in [39] and PRAM models in [18], [20]) start to come in after almost two decades of non-systematic experiments (started in the late 1950) and the too early market production
- **? – mathematical computation model**: no one yet really considered it, regrettably confused with abstract machine models, although it is there waiting for us (see Kleene's mathematical model for computation [26]).

A.4.14 RISC

The term RISC (Reduction Instruction Set Computer) was coined by David Patterson. It means processors with an architecture characterized by:

load-store mechanism : divides instructions into two categories: ALU operations between registers, and memory access as simple load and store between memory and registers instead complex multi-indirected memory access modes

one-word instructions : instructions are coded in on word; even when an immediate value is involved, it is taken into account that in most cases small values are involved that can be encoded with a small number of bits making it unnecessary to add an additional word to specify the value.

one-cycle execution : using a Harvard abstract model, a load-store mechanism and one-word instructions, it is possible to design a processor which execute each instruction in one clock cycle

only most frequent instructions : because the statistics compiled on large program databases showed an uneven distribution of the use of the instructions in the established ISAs, it was decided to keep in the ISA only the frequently used instructions, provided that the omitted ones could be made by a sequence of those maintained

which will lead to:

> *The goal of any instruction format should be: 1. simple decode, 2. simple decode, and 3. simple decode. Any attempts at improved code density at the expense of CPU performance should be ridiculed at every opportunity.* [48]

The resulting reductions in complexity and size have increased the number of registers, increased clock frequency and reduced power consumption.

The first implementations:

IBM 801 : based on a statistical study launched in the mid-1970s that highlighted the need to increase the number of registries and the possibility of removing from current ISAs a significant number of complex instructions that compilers "ignored".

Berkeley RISC : when David Patterson was sent in 1979 on a sabbatical from University of California, Berkeley to help DEC to improve the VAX microcode, he discovered that if the microcode was removed, the programs would run faster. The microcode was responsible for interpreting the complex instructions. Then, removing the complex instructions from ISA becomes a solution for improving processor's performance.

MIPS : stands for Microprocessor without Interlocked Pipeline Stages, a project which came from a graduate course of John L. Hennessy At Stanfort University; it produced a functioning system in 1983.

Since 2010 a new ***open source*** ISA, RISC-V, has been under development at the University of California, Berkeley.

A.4.15 FPGA & Adaptive Computer Acceleration Platform

1985 is considered the year of the birth of FPGA (Field-Programmable Gate Array) technology with the founding of Xilinx, although in 1983 the founding of Altera led to the first forms of this technology.

ACAP (Adaptive Computer Acceleration Platform) is the technology that naturally emerges from the FPGA approach by the fact that users of the last decades have expressed preferences that have outlined specific structures that can be implemented as standardizable IPs.

> *An ACAP is a heterogeneous, hardware adaptable platform that is built from the ground up to be fully software programmable. An ACAP is fundamentally different from any multi-core architecture in that it provides hardware programmability but the developer does not have to understand any of the hardware detail.* [4]

A.5 User-driven evolution: Computation as General-Purpose Technology

A.5.1 Microsoft's Surface

There comes a time when users begin to define and produce computing equipment for their own use. A significant example is Microsoft's Surface series of touchscreen-based personal computers, tablets and interactive whiteboards.

Microsoft first announced Surface at an event on June 18, 2012, as the first major initiative by Microsoft to integrate its Windows operating system ***with its own hardware***, and is the first PC designed and distributed solely by Microsoft.

A.5.2 Google's Tensor Processing Unit

In 2015, a step forward is taken: an end-user begins to define and produce ICs for his own use. Google began producing and using Tensor Processing Unit (TPU) as an AI accelerator ASIC developed specifically for neural network machine learning, particularly using Google's own TensorFlow software.

From 2018, the circuits from the TPU family are also made available to other users. We are witnessing a mechanism by which the new technology is developed by an end-user and then made available to

other users. The development of general purpose machines (processors, computers) that are made available to different users is replaced by a process in a reverse way, users of devices dedicated to a particular field promote products that are disseminated as general purpose products. We can also exemplify by GPUs used as GPGPU. Note the oxymoronic formulation: General-Purpose Graphic Processing Unit.

A.5.3 Apple's M1

M1 = (8-core ARM CPU + GPU + Neural Engine + ... + Cache) + DRAM: a first example of Accelerator-Level Parallelism.

> *"So the physical RAM modules are still separate entities, but they are sitting on the same green substrate as the processor. ... Apple calls its approach a "Unified Memory Architecture" (UMA)."*

A.5.4 Tesla's Artificial Intelligence & Autopilot

FSD Chip Build AI inference chips to run our Full Self-Driving software, considering every small architectural and micro-architectural improvement while squeezing maximum silicon performance-per-watt.

A.5.5 Hadoop & Big-Data

Hadoop is an open source processing system that manages distributed data processing and storage for ***Big Data*** applications for scalable clusters. It manages an ecosystem of Big Data applications that are used to support advanced data mining and machine learning.

A.5.6 The Next Target: Artificial General Intelligence

AGI may be the ability of computers to solve problems in a way that human beings do, using intuition and common sense in addition to formal skills.

Current AI techniques can be considered "narrow AI" or "weak AI" because they refer to well-defined areas of competence, areas in which they currently exceed human performance.

AGI is a goal that is not only difficult to achieve, but, first of all, very difficult to define.

Artificial General Intelligence (AGI)

A.6 Application-driven history

A significant turning point came in 1971, when Intel launched the first successful silicon memory (1103) and the first one-chip microprocessor (4004). In the same year, e-mail (`@Mail`) and the wireless network appeared. It is the moment when the evolution of the field of computing begins to be more and more marked by applications oriented towards the big market. The computer and its applications, until then oriented towards government institutions, universities or corporate space, are beginning to be oriented towards the consumer market. The main consequence will be, from that moment, the evolution under the pressure of the criteria imposed by the market.

Is it a coincidence that one last important theoretical issue – *NP-completeness* – is being addressed this year? An era of theoretical research seems to be coming to an end, and an era of applied developments is beginning.

It is worth mentioning some of the stages completed in the last half century [19]:

1972: HP-35 pocket calculator destroyed the market for the slide rules. HP-35 because it had 35 keys. Used to run at 200 KHz programs no longer then 768 instructions.

1973: first cell phone call in April 3, on Sixth Avenue in New-York City between Fifty-Third and Fifty Four Streets.

1973: Alto the first personal computer developed by Xerox equipped with a graphic-user interface.

1975: *Adventure* the first text-based simulation used as a game.

1983: 3-D printing is an additive manufacturing technology which fabricate objects in the field starting from raw materials.

1983: first laptop comes preloaded with a rudimentary word processor and a basic spreadsheet program. It comes on the market under the specifications of RadioShack® TRS-80 Model 100 equiped with an 8-bit Intel 80C85 microprocessor. The operating system, written almost entirely by Bill Gates, was loaded in a 32 KB of ROM.

1983: MIDI computer music interface helped to put music creation into the hands of more users to generate great music without a professional performer because the computer played the music. (MIDI stands for Musical Instrument Digital Interface.)

1984: text-to-speech technology commoditized by DEC by its standalone appliance DECtalk

1984: virtual reality a term coined by Jaron Lanier as being the outcome of running programs written in Virtual Programming Language on specific hardware.

1984: Verilog is a Hardware Description Language used by designers to describe, simulate, and synthesize digital systems.

1985: desktop publishing allowed anybody to generate high quality documents with a tight control on fonts and graphics.

1988: CD-ROM stands for Compact Disc - Read-Only Memory; it is used to store music, video and software.

1989: www which stands for world wide web, transformed internet connection into a dominating technology connecting virtually every person on the planet.

1990: GPS which stands for Global Positioning System, is a consumer navigation system based on old radio waves technologies.

1992: Boston Dynamics a robotics company maker of biped and quadruped robots capable of traversing rough landscapes.

1992: First mass-market web browser called Mosaic changed the way the www is accessed from a professional procedure to an easy way which does not imply technical expertise.

1993: Apple Newton electronic organizer; handwriting recognizer based on Arcon RISC MAchine (ARM) a low-power computationally powerful controller.

1995: E-Commerce becomes possible as soon as the most important ingredient - security - has been implemented. Netscape *Secure Socket Layer* allowed consumers to securely send credit card numbers over the Internet.

1997: Deep-Blue beat world chess campion Garry Kasparov

1997: E-Ink electronic paper display (EPD) is a reflective display that is visible in direct sunlight.

1998: Google based on an algorithm to rank organizes the pages on WWW. The algorithm takes into account the number of links the quality of pages. A page is important if it is pointed from a big numbers of important pages. In 2006 *Google* becomes a verb.

2001: Wikipedia is the result of a mass-collaborative effort of organizing knowledge as a continuous process which may containing errors, mistakes, biased attitude, but being open to self-correcting mechanisms, till the end it is able to provide a very useful image of the current stage of knowledge.

2004: Facebook is a (too) free communication platform. Provides a solution to the desire to connect with and learn about other people.

2007: iPhone invented by Apple puts together telephony, messaging, internet access, music, color screen, touch-based interface. The main big things associated with iPhone: specialized programs called *apps*.

2008: Blockchain a collection of transactions – *blocks* – managed in the most possible secure mode thus allowing the development of the *criptocoin* environment and many other distributed applications.

2022: chatbot the chat robot *chatGPT* where GPT stands for ***generative pre-trained transformer*** .

A.7 Programming paradigms

Programming languages:

- low level languages
  - machine languages: uses the instructions' numeric values of instructions directly
  - assembly languages: generate executable machine code from assembly code where for each statement there is a machine instruction; uses mnemonic codes to refer to machine code instructions
- high level languages
  - imperative languages: generate explicit statements about how the machine state changes
    - FORTRAN: scientific applications (1953-1957)
    - ALGOL (1958)
    - COBOL: business applications (1959)
    - Basic
    - Pascal
    - C

    * ...
  - declarative languages: describe what a computation should perform, without specifying detailed state changes
    * functional languages: use evaluation of mathematical functions instead of explicit state change (Lisp (1958), Clojure (2007))
    * logic languages: involves explicit mathematical logic for programming (Prolog)
  - multi-paradigm programming languages: programming languages that supports more than one programming paradigm.
    * Python: supports multiple programming paradigms, including procedural, object-oriented, and functional programming
    * ...

- library of functions
  - BLAS
  - Eigen
  - Tensorflow
  - ONNX (Open Neural Network Exchange) is an open format built to represent machine learning models. ONNX defines a common set of operators - the building blocks of machine learning and deep learning models
  - ...

A.8 The Qubit

Quantum computing is a type of computation based on the collective properties of quantum states, such as superposition, interference, and entanglement, to perform computation. The elementary devices that perform quantum computations are qubits organized in what known as quantum computers.

The quantum computation performs the computation using a network of *quantum logic gates*. A quantum gate is a complex linear-algebraic generalization of boolean circuits.

In 2001, a team of IBM scientists factored the number 15 with a quantum computer that had 7 qubits. In 2019, IBM has launched *Q System One*, the first circuit-based commercial quantum computer.

Appendix B

Kleene's Mathematical Model of Computation

B.1 Kleene's Model of Partial Recursive Functions

Definition B.1 *Let be the positive integers $x, y, i \in \mathbb{N}$ and the vector $\vec{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle \in \mathbb{N}^n$. Any partial recursive function $f : \mathbb{N}^n \to \mathbb{N}$ can be computed using three **initial functions**:*

- $ZERO(x) = 0$: *the variable x takes the value* **zero**
- $INC(x) = x + 1$: **increment***s the variable $x \in \mathbb{N}$*
- $SEL(i, \vec{x}) = x_i$: i **select***s the value of x_i from the vector of positive integers $\vec{x}$*

*and the application of the following three **rules**:*

- **Composition:** $f(\vec{x}) = g(h_0(\vec{x}), \ldots, h_{p-1}(\vec{x}))$*, where: $f : \mathbb{N}^n \to \mathbb{N}$ is a total function if $g : \mathbb{N}^p \to \mathbb{N}$ and $h_i : \mathbb{N}^n \to \mathbb{N}$, for $i = 0, 1, \ldots p-1$, are total functions*
- **Primitive recursion:** $f(\vec{x}, y) = g(\vec{x}, f(\vec{x}, (y-1)))$*, with $f(\vec{x}, 0) = h(\vec{x})$ where: $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is a total function if $g : \mathbb{N}^{n+1} \to \mathbb{N}$ and $h : \mathbb{N}^n \to \mathbb{N}$ are total functions.*
- **Minimization:** $f(\vec{x}) = \mu_y[g(\vec{x}, y) = 0]$*, which means: the value of the function $f : \mathbb{N}^n \to \mathbb{N}$ is the smallest y, **if any**, for which the function $g : \mathbb{N}^{n+1} \to \mathbb{N}$ takes the value $g(\vec{x}, y) = 0$.*

$\diamond$

Kleene's model looks like a good candidate for a mathematical model for parallel computing as the Turing's model was for the mono-core computation. In this respect, the composition seems to be a natural embodiment of a many-core abstract model for the parallel computing engine. The following conjecture has a big chance to become a theorem:

Conjecture B.1 *The composition rule, implemented as a two level structure (see Figure B.1):*

- *the* **MAP** *level: a linear array of circuits, one for each $h_i(\vec{x})$ function*

- *the* **REDUCE** *level: a* log*-depth tree-like network of circuits for* $g(h_0(\vec{x}),\ldots,h_{p-1}(\vec{x}))$

where the functions $h_i(\vec{x})$ *and the function* $g(h_0(\vec{x}),\ldots,h_{p-1}(\vec{x}))$ *are initial functions or hierarchic compositions of initial functions, computes any functions* $f:\mathbb{N}^n\to\mathbb{N}$.

◇

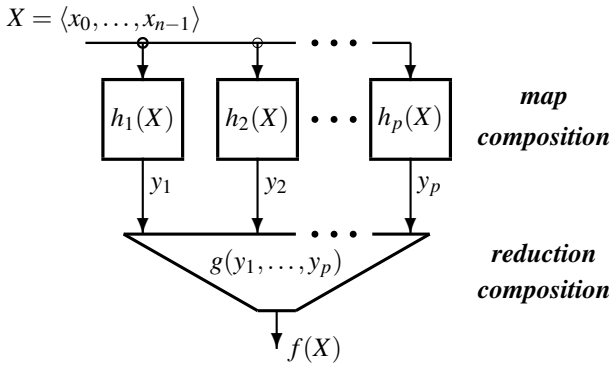

Figure B.1: **The circuit version of composition**. It is a two-layer construct: the parallel expanded *map* layer serially connected with the *reduction* layer.

Two kinds of parallelism are emphasized by the composition rule:

- a n-degree of synchronic parallelism between the computation performed in the circuits of the MAP level
- a 2-degree of diachronic (pipeline) parallelism between the computation on the map level and the computation of the REDUCE level

In the next sections will be proved that, for the other two rules, specific compositions can be used, so as we are in the position to conclude that the computation model proposed by Kleene leads to implementations involving only compositions for which the previous conjecture applies.

B.2 Preliminary Definitions

Definition B.2 *The reduction-less composition or* map composition*, MC, is the particular composition* $f:\mathbb{N}^n\to\mathbb{N}^p$ *where:*

$$f(\vec{x})=f(x_0,\ldots x_{n-1})=\langle h_0(\vec{x}),\ldots h_{p-1}(\vec{x})\rangle=\langle y_0,\ldots y_{p-1}\rangle=\vec{y}$$

$h_i:\mathbb{N}^n\to\mathbb{N}$*, and* $g(\vec{y})=\vec{y}$ *is the identity function, for* $i=0,\ldots,p-1$.

◇

Definition B.3 *The map-less composition or* reduction composition*, RC, is the particular composition* $f:\mathbb{N}^n\to\mathbb{N}$ *where:*

$$f(\vec{x})=g(\vec{x})$$

with $y_i = h_i(\vec{x}) = SEL(i,\vec{x}) = x_i$, *for* $i = 0,\ldots,p-1$ *and* $n = p$.

◇

According to the previous two definitions, the composition rule can be considered as having a ***map-reduce*** structure (Figure B.1), where a MC is serially connected with a RC. The two functional level can have associated the physical implementation 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 a computer.

Definition B.4 *The RC function* $redOR : \mathbb{N}^n \to \mathbb{N}$ *is*

$$redOR(\vec{x}) = x_0|x_1|\ldots|x_{n-1}$$

where: $|$ *denote the bitwise OR logical function.*

◇

Definition B.5 *For* $\vec{x} = \langle x_0, x_1, \ldots, x_i \ldots\rangle$, *let us define the* ***right-chained MC*** *as:*

$$rightMC(x,\vec{x}) = H'(x,\vec{z}) \circ H(\vec{x})$$

composed by the following two MCs:

$$H(\vec{x}) = \langle h_0(\vec{x}), h_1(\vec{x}), \ldots, h_i(\vec{x}) \ldots\rangle = \langle z_0, z_1, \ldots, z_i \ldots\rangle$$

$$H'(x,\vec{z}) = \langle h'_0(x, SEL(0,\vec{z})), h'_1(SEL(0,\vec{z}), SEL(1,\vec{z})), \ldots, h'_i(SEL(i-1,\vec{z}), SEL(i,\vec{z})), \ldots\rangle$$

◇

Thus, *rightMC*s is a MC which contains a serial connection between the cells performing the functions $h_i \circ h'_i$, for $i = 0,1,\ldots,i,\ldots$. Theoretically, MAP section is right unlimited. Results the circuit from Figure B.2. Similarly, a left serial connection can be defined.

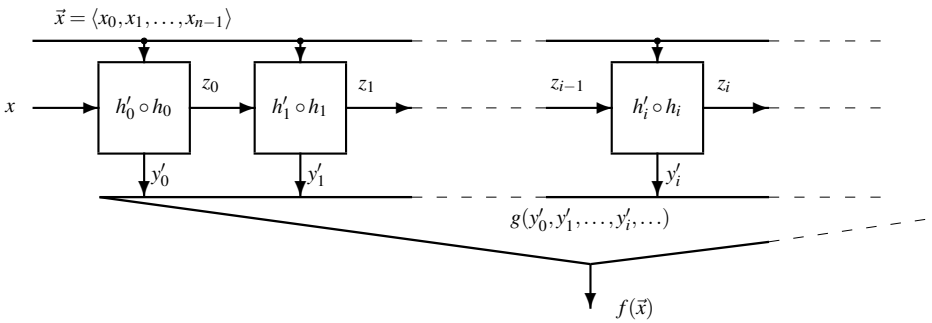

Figure B.2: **The circuit associated to the composition rule expanded with a rightSHIFT serial connections.**

Definition B.6 *By definition,* $rightSHIFT(x,\vec{x})$ *is a* $rightMC(x,\vec{x})$ *for* $\vec{x} = \langle x_0, x_1, \ldots, x_i \ldots\rangle$, $h_i(\vec{x}) = SEL(i,\vec{x})$, $h'_0 = x$, *and* $h'_i = SEL(i-1,\vec{z})$, *for* $i = 1,2,\ldots,i,\ldots$, *such that:*

$$rightSHIFT(x,\vec{x}) = \langle x, x_0, x_1, \ldots, x_i \ldots\rangle$$

◇

Definition B.7 *We define $prefixOR(\vec{b})$ as a scan circuit for the OR prefix function [**?**], so it receiving a binary input vector*

$$\vec{b} = \langle b_0, b_1, \ldots, b_i, \ldots \rangle$$

returns

$$prefixOR(\vec{b}) = \langle b_0, b_0|b_1, b_0|b_1|b_2, \ldots, OR_0^i b_j, \ldots \rangle$$

◇

Definition B.8 *The MC function $scanFIRST : \{0,1\}^n \to \{0,1\}^n$ is:*

$$scanFIRST(\vec{b}) = prefixOR(\vec{b}) \& \sim (rigthSHIFT(prefixOR(\vec{b})))$$

where: $\vec{b} = \langle b_0, b_1, \ldots, b_{n-1} \rangle$ is a Boolean sequence.

◇

The $scanFIRST(\vec{b})$ function identifies, ***if any***, the first occurrence of 1 in a the Boolean sequence $\vec{b}$.

B.3 Primitive Recursion Computed as a Sequence of Compositions

Theorem B.1 *The primitive recursive rule is reducible to repeated applications of specific compositions.*

◇

Proof B.1 *The primitive recursion rule could be applied using its iteratively expanded form:*

$$f(\vec{x}, y) = g(\vec{x}, f(\vec{x}, y-1)) = \ldots = \underbrace{g(\vec{x}, g(\vec{x}, g(\vec{x}, \ldots g}_{(y-1)\ times}(\vec{x}, f(\vec{x}, 1)) \ldots))) =$$

$$= \underbrace{g(\vec{x}, g(x, g(\vec{x}, \ldots g}_{y\ times}(\vec{x}, f(x, 0)) \ldots))) = \underbrace{g(\vec{x}, g(\vec{x}, g(\vec{x}, \ldots g}_{y\ times}(\vec{x}, h(x)) \ldots)))$$

Let be, in Figure B.3, the specific instantiation of the righMC function (see Definition B.5). It computes iteratively, starting in the first stage, P_0, with the function $f(\vec{x}, 0) = h(\vec{x})$, in each P_i the values $f(\vec{x}, i)$ for $i = 1, 2, \ldots$. In each cell the predicate $(y = i)$ is computed. The functions P_i, for $i = 1, 2, \ldots$, takes form P_{i-1} the value of $f(\vec{x}, i-1)$ and computes $f(\vec{x}, i)$. The redOR function takes from P_i its arguments as $(y = i)\ ?f(\vec{x}, i) : 0$ for $i = 0, 1, 2, \ldots$. Because for only one i the predicate $y = i$ takes the value 1, the function redOR returns the value of $f(\vec{x}, y)$.

Thus, for primitive recursion we need to compose two compositions, $rightSHIFT$ and redOR.

◇

Figure B.3 presents the circuit version of the function obtained by composing a specific *rightMC* function with the *redOR* 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 Machine. But, it is very important that the algorithmic complexity of the description is in $O(1)$, because the functions P_i, defining *rightMC*, and *redOR* have constant size descriptions.

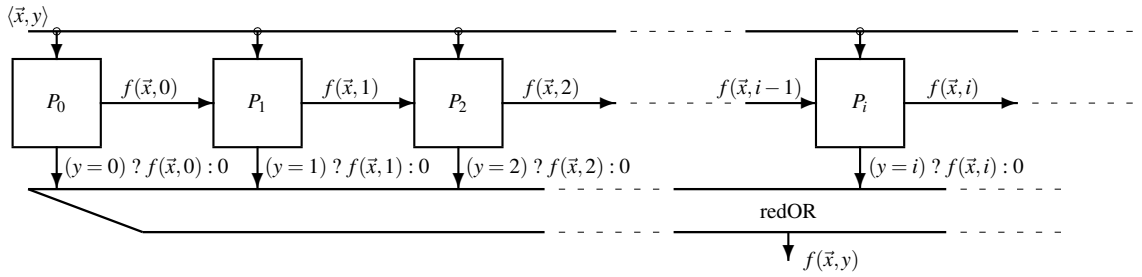

Figure B.3: **The** *rightMC* & *redOR* **circuit version for the partial recursive rule**.

B.4 Minimization Computed as a Sequence of Compositions

Theorem B.2 *The minimization (least-search) rule is reducible to repeated applications of specific compositions.*

◇

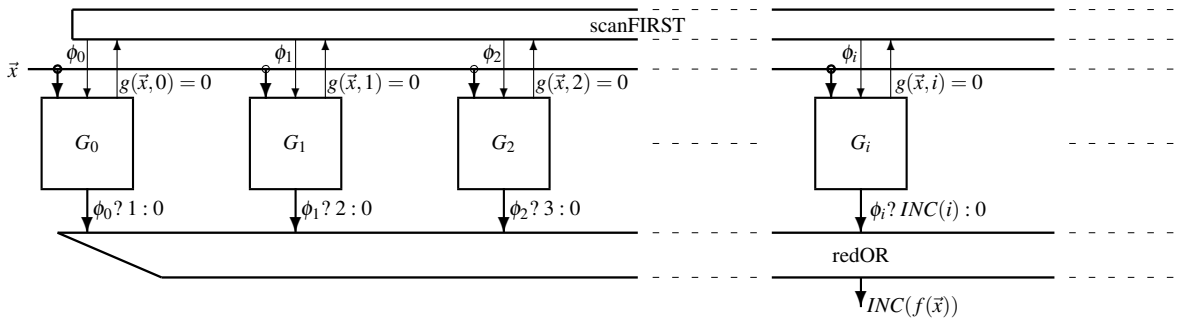

Figure B.4: **The circuit structure for the minimization rule**.

Proof B.2 *The minimization (least-search) rule computes the value of $f(\vec{x})$ as the smallest y,* **if any**, *for which $g(\vec{x},y) = 0$. The functional structure to which we will refer is represented in Figure B.4.*

The scanFIRST *MC receives from the MAP section the vector of predicates*

$$\langle (g(\vec{x},0) = 0), (g(\vec{x},1) = 0), \ldots, (g(\vec{x},i) = 0), \ldots \rangle$$

and returns the Boolean vector

$$\vec{\phi} = \langle \phi_0, \phi_1, \ldots, \phi_i, \ldots \rangle$$

Thus, scanFIRST *MC points, with $\phi_k = 1$, to the first cell,* ***if any****, which provided the predicate $(g(\vec{x},i) = 0) = 1$. The MAP section uses the vector $\vec{\phi}$ to generate the vector:*

$$\langle (\phi_0 \ ? \ 1 : 0), (\phi_1 \ ? \ 2 : 0), \ldots, (\phi_i \ ? \ INC(i) : 0), \ldots \rangle$$

as input for the reduction section redOR. *Thus, the output of the reduction section takes the value* $INC(f(\vec{x}))$*, because the value 0 is reserved to indicate that the function is not defined for the value* $\vec{x}$ *applied on the input.*

◇

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 $O(1)$.

B.5 Partial Recursion Means Composition Only

Kleene's approach defines, besides the composition rule, other two rules, ordinary (primitive) recursion and minimization (least-search), only for providing the means for classifying the recursive functions (to emphasize in the class of recursive functions partial recursive functions and primitive recursive functions). Therefore, to define the computation the next corollary makes the necessary and sufficient delimitation.

Corollary B.1 *Any computation defined in Definition B.1 can be done, according to Theorem B.1 and Theorem B.2, using the initial functions and the repeated application of the composition rule.*

◇

The primitive recursion is differentiated from the partial recursion by the main fact that it does not require the *scanFIRST* section. It is only a straight forward sequence of compositions. For the partial recursive rule, the additional scan section is required to test whether an imposed condition is met or not.

The composition rule is defined for a finite p, while various forms of the composition rule used to define primitive recursion and minimization are designed for an "infinite" p. Thus, Corollary B.1 can be used to define an ***abstract model for parallel computations***. To get rid of the "infinity" assumed by the pure mathematical definition of parallelism, a recurrent process is necessary. It will be based on the endowment of cells with internal memory, and on the introduction of a sequencer necessary to control the recurrent process, control that takes into account the feedback loops introduced by the two *log*-depth networks: the reduction network and the scan network. The sequencer will allow the theoretically "infinite" size assumed by the mathematical model to be emulated using a constant-size structure defined as the abstract model for parallel computing, just as the Harvard and von Neumann abstract model did for Turing's model.

Appendix C

HETEROGENEOUS SYSTEM SIMULATOR

C.1 Heterogenous Computing System Structure: `1_hetSys.sv`

```
/*****************************************************************
File: 0_hetSys.sv
Name: Heterogenous System
Description:
*****************************************************************/
`include "0_DEFINES.vh"
module hetSys( input  logic reset ,
               input  logic clock );
    logic           h2iPwrite   ; // host to interface program write
    logic           i2hPfull    ; // programFIFO is full
    logic [63:0]    h2i         ; // host to data program & data
    logic           h2iDwrite   ; // host to interface data write
    logic           i2hDfull    ; // inputDataFIFO is full
    logic [63:0]    i2h         ; // interface to host data
    logic           h2iDread    ; // host to interface data read
    logic           i2hDempty   ; // outputDataFIFO is empty
    logic           a2hInt      ; // accelerator to host interrupt
    logic           h2aInta     ; // interrupt acknowledge

    host HOST( h2iPwrite    ,
               i2hPfull     ,
               h2i          ,
               h2iDwrite    ,
               i2hDfull     ,
               i2h          ,
               h2iDread     ,
               i2hDempty    ,
               a2hInt       ,
               h2aInta      ,
               reset        ,
```

```
                clock       );
    accelerator ACCELERATOR(h2iPwrite    ,
                            i2hPfull     ,
                            h2i          ,
                            h2iDwrite    ,
                            i2hDfull     ,
                            i2h          ,
                            h2iDread     ,
                            i2hDempty    ,
                            a2hInt       ,
                            h2aInta      ,
                            reset        ,
                            clock        );
endmodule
```

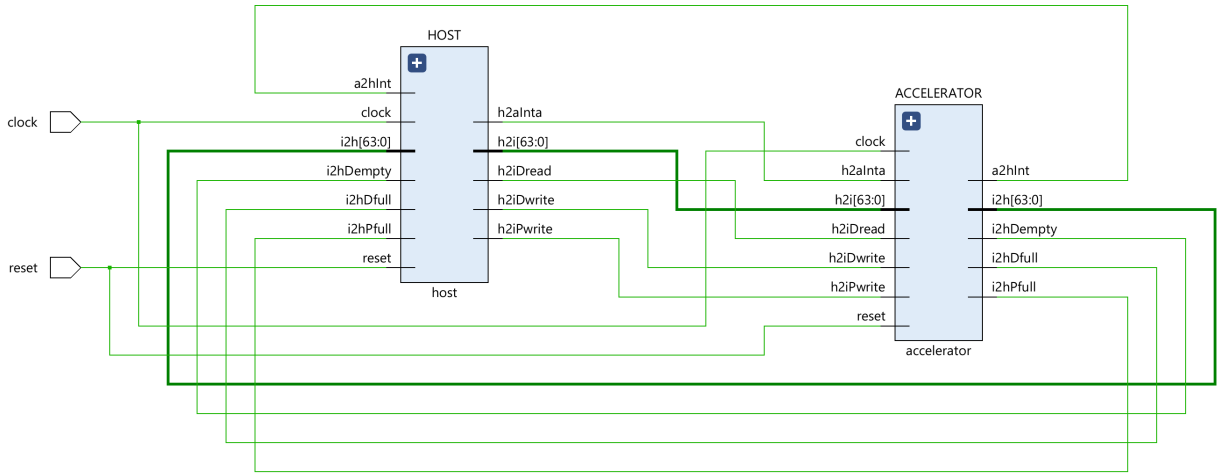

Figure C.1: Heterogeneous System.

C.1.1 Host Computer: `2_host.sv`

It is a conventional RISC computing engine.

```
/*****************************************************************
File: 1_host.sv
Name: Host Computer
Description:
*****************************************************************/
module host(output  logic          h2iPwrite    ,
            input   logic          i2hPfull     ,
            output  logic [63:0]   h2i          ,
            output  logic          h2iDwrite    ,
```

```
        input   logic           i2hDfull    ,
        input   logic [63:0]    i2h         ,
        output  logic           h2iDread    ,
        input   logic           i2hDempty   ,
        input   logic           a2hInt      ,
        output  logic           h2aInta     ,
        input   logic           reset       ,
        input   logic           clock       );
logic [$clog2(`m)-1:0]  hostProgAddr;
logic [$clog2(`m)-1:0]  hostDataAddr;
logic [31:0]            instruction ;
logic [$clog2(`m)-1:0]  transAddr   ;
logic [`n-1:0]          procDataIn  ;
logic [`n-1:0]          procDataOut ;
logic                   procWrite   ;
logic [$clog2(`m):0]    procAddr    ;
logic [63:0]            mem2int     ;
logic                   selProgOut  ;
logic [63:0]            prog2int    ;

assign h2i = selProgOut ? prog2int : mem2int;
hostProcessor hostProcessor(h2iPwrite       ,
                            i2hPfull        ,
                            mem2int[31:19]  ,
                            h2iDwrite       ,
                            i2hDfull        ,
                            h2iDread        ,
                            i2hDempty       ,
                            selProgOut      ,
                            prog2int        ,
                            hostProgAddr    ,
                            instruction     ,
                            transAddr       ,
                            procDataIn      ,
                            procDataOut     ,
                            procWrite       ,
                            procAddr        ,
                            a2hInt          ,
                            h2aInta         ,
                            reset, clock    );
hostProgramMemory
    hostProgramMemory(  .hostProgAddr(hostProgAddr  ),
                        .hostProgram (0             ),
                        .we          (1'b0          ),
                        .instruction (instruction   ),
                        .clock       (clock         ));
hostDataMemory hostDataMemory(  mem2int     ,
                                i2h         ,
                                h2iDread    ,
                                transAddr   ,
```

```
                                        procDataIn   ,
                                        procDataOut  ,
                                        procWrite    ,
                                        procAddr     ,
                                        clock        );
endmodule
```

Processor: `3_hostProcessor.sv`

It is a simple RISC processor called ***toyRISC*** processor.

```
/*****************************************************************
File: 2_hostProcessor.sv
Name: Host Processor
Description:
*****************************************************************/
module hostProcessor(
        output  logic                       h2iPwrite       ,
        input   logic                       i2hPfull        ,
        input   logic [12:0]                cInstr          ,
        output  logic                       h2iDwrite       ,
        input   logic                       i2hDfull        ,
        output  logic                       h2iDread        ,
        input   logic                       i2hDempty       ,
        output  logic                       selProgOut      ,
        output  logic [63:0]                prog2int        ,
        output  logic [$clog2(`m)-1:0]      hostProgAddr    ,
        input   logic [31:0]                instruction     ,
        output  logic [$clog2(`m)-1:0]      transAddr       ,
        input   logic [`n-1:0]              procDataIn      ,
        output  logic [`n-1:0]              procDataOut     ,
        output  logic                       procWrite       ,
        output  logic [$clog2(`m):0]        procAddr        ,
        input   logic                       a2hInt          ,
        output  logic                       h2aInta         ,
        input   logic                       reset, clock    );
    logic [1:0]     loadCom         ;
    logic [31:0]    leftOut         ;
    logic [31:0]    hostCycleCounter;

    assign procAddr     = leftOut[$clog2(`m):0]     ;
    assign transAddr    = leftOut[$clog2(`m)-1:0]   ;
    hostDCD hDCD
        (   .instr      (instruction    ),
            .writeEnable(writeEnable    ),
            .loadCom    (loadCom        ),
            .h2iPwrite  (h2iPwrite      ),
```

```
            .h2iDwrite   (h2iDwrite       ),
            .h2iDread    (h2iDread        ),
            .procWrite   (procWrite       ),
            .selProgOut  (selProgOut      ),
            .prog2int    (prog2int        ));
    hostPC hPC(
        .host2accInta       (h2aInta                    ),
        .acc2hostInt        (a2hInt                     ),
        .cInstr             (cInstr                     ),
        .leftEqRigth        (leftOut == procDataOut     ),
        .leftOut            (leftOut                    ),
        .hostProgAddr       (hostProgAddr               ),
        .hOpCode            (instruction[31:26]         ),
        .value              (instruction[$clog2(`m)-1:0]),
        .hostCycleCounter   (hostCycleCounter           ),
        .reset              (reset                      ),
        .clock              (clock                      ));
    hostRALU hRALU
        (   .writeEnable(writeEnable        ),
            .destAddr   (instruction[25:21] ),
            .leftAddr   (instruction[20:16] ),
            .rightAddr  (instruction[15:11] ),
            .value      (instruction[15:0]  ),
            .procDataIn (procDataIn         ),
            .hFunction  (instruction[31:26] ),
            .i2hPfull   (i2hPfull           ),
            .i2hDfull   (i2hDfull           ),
            .i2hDempty  (i2hDempty          ),
            .loadCom    (loadCom            ),
            .leftOut    (leftOut            ),
            .rightOut   (procDataOut        ),
            .clock      (clock              ));
endmodule
```

Decoder: `4_hostDCD.sv` is the ***toyRISC***'s decoder.

```
/*****************************************************************
File: 3_hostDCD.sv
Name: Host Decoder
Description:
*****************************************************************/
module hostDCD (input   logic [31:0]    instr       ,
                output  logic           writeEnable ,
                output  logic [1:0]     loadCom     ,
                output  logic           h2iPwrite   ,
                output  logic           h2iDwrite   ,
                output  logic           h2iDread    ,
```

```
                output  logic             procWrite    ,
                output  logic             selProgOut   ,
                output  logic [63:0]      prog2int     );

    assign writeEnable = (instr[31:30] == 2'b01) ||
                         (instr[31:30] == 2'b10)     ;
    always_comb case(instr[31:26])
                    `hvalue : loadCom = 2'b00;
                    `hinsval: loadCom = 2'b01;
                    `hload  : loadCom = 2'b10;
                    default : loadCom = 2'b11;
                endcase
    assign h2iPwrite    =   (instr[31:26] == `hpsend) ||
                            (instr[31:26] == `hfsend) ||
                            (instr[31:26] == `hssend)   ;
    assign selProgOut   =   (instr[31:26] == `hfsend) ||
                            (instr[31:26] == `hssend)   ;
    assign h2iDwrite    =    instr[31:26] == `hdsend    ;
    assign h2iDread     =    instr[31:26] == `hdget     ;
    assign procWrite    =    instr[31:26] == `hstore    ;

    assign prog2int = {32'b0, ((instr[31:26] == `hfsend) ?
        {`contr, `imm, `prun,  instr[18:0]} :
        {{6{instr[25]}}, instr[25:0]})}                ;
endmodule
```

Program Counter: `4_hostPC.sv` is the program counter of ***toyRISC***.

```
/*****************************************************************
File: 3_hostPC.sv
Name: Host Program Counter
Description:
*****************************************************************/
module hostPC(
        output  logic                   host2accInta    ,
        input   logic                   acc2hostInt     ,
        input   logic [12:0]            cInstr          ,
        input   logic                   leftEqRigth     ,
        input   logic [31:0]            leftOut         ,
        output  logic [$clog2(`m)-1:0]  hostProgAddr    ,
        input   logic [5:0]             hOpCode         ,
        input   logic [$clog2(`m)-1:0]  value           ,
        output  logic [31:0]            hostCycleCounter,
        input   logic                   reset           ,
        input   logic                   clock           );
    logic [$clog2(`m)-1:0]  programCounter      ;
    logic                   hCycleCounterEnable ;
```

```
    always_ff @(posedge clock)
     if (reset) hostCycleCounter    <= 0    ;
      else begin
           if (hCycleCounterEnable)
            hostCycleCounter    <= hostCycleCounter + 1 ;
           if (hOpCode == `hsttcc)
            hCycleCounterEnable <= 1                    ;
           if (hOpCode == `hstpcc)
            hCycleCounterEnable <= 0                    ;
          end
    assign host2accInta =
           acc2hostInt && (hOpCode == `hintwait)   ;
    always_ff @(posedge clock)
        if (reset)  programCounter <= -1            ;
            else    programCounter <= hostProgAddr  ;
    hNextPC hnPC(   hostProgAddr    ,
                    programCounter  ,
                    hOpCode         ,
                    value           ,
                    acc2hostInt     ,
                    cInstr          ,
                    leftEqRigth     ,
                    leftOut         );

endmodule
```

RALU: 4_hostRALU.sv is the Verilog file describing the RALU unit of ***toyRISC***.

```
/*****************************************************************
File: 3_hostRALU.sv
Name: Host RALU
Description:
*****************************************************************/
module hostRALU (   input   logic           writeEnable ,
                    input   logic [4:0]     destAddr    ,
                    input   logic [4:0]     leftAddr    ,
                    input   logic [4:0]     rightAddr   ,
                    input   logic [15:0]    value       ,
                    input   logic [31:0]    procDataIn  ,
                    input   logic [5:0]     hFunction   ,
                    input   logic           i2hPfull    ,
                    input   logic           i2hDfull    ,
                    input   logic           i2hDempty   ,
                    input   logic [1:0]     loadCom     ,
                    output  logic [31:0]    leftOut     ,
                    output  logic [31:0]    rightOut    ,
```

```
                    input   logic           clock       );
    logic [31:0]    result  ;
    logic [31:0]    muxOut  ;

    hostRegisterFile hRF(   muxOut      ,
                            writeEnable ,
                            destAddr    ,
                            leftAddr    ,
                            rightAddr   ,
                            leftOut     ,
                            rightOut    ,
                            clock       );
    hostALU hALU(   result      ,
                    leftOut     ,
                    rightOut    ,
                    value       ,
                    hFunction   ,
                    i2hPfull    ,
                    i2hDfull    ,
                    i2hDempty   ,
                    clock       );
    hostInMUX hInMux(   .loadCom    (loadCom        ),
                        .result     (result         ),
                        .value      (value          ),
                        .procDataIn (procDataIn     ),
                        .leftOutLow (leftOut[15:0]  ),
                        .muxOut     (muxOut         ));
endmodule
```

Host Register File : `4_hostRegisterFile.sv` is the Verilog file describing the register file unit of ***hostRALU***.

```
/*****************************************************************
File: 4_hostRegisterFile.sv
Name:
Description:
*****************************************************************/
module hostRegisterFile
        (   input   logic [31:0]    muxOut      ,
            input   logic           writeEnable ,
            input   logic [4:0]     destAddr    ,
            input   logic [4:0]     leftAddr    ,
            input   logic [4:0]     rightAddr   ,
            output  logic [31:0]    leftOut     ,
            output  logic [31:0]    rightOut    ,
            input                   clock       );
    logic [31:0]    hRegFile[0:31]  ;
```

```
    always_ff @(posedge clock)
        if (writeEnable) hRegFile[destAddr] <= muxOut    ;
    assign leftOut  = hRegFile[leftAddr ]    ;
    assign rightOut = hRegFile[rightAddr]    ;
endmodule
```

Host ALU : `4_hostALU.sv` is the Verilog file describing the register file unit of ***hostALU***.

```
/*********************************************************************
File: 4_hostALU.sv
Name:
Description:
*********************************************************************/
module hostALU (     output  logic [31:0]     result      ,
                     input   logic [31:0]     leftOut     ,
                     input   logic [31:0]     rightOut    ,
                     input   logic [15:0]     value       ,
                     input   logic [5:0]      hFunction   ,
                     input   logic            i2hPfull    ,
                     input   logic            i2hDfull    ,
                     input   logic            i2hDempty   ,
                     input   logic            clock       );
    logic            crFF    ;
    logic            cr      ;
    logic [31:0]     rightVal;

    assign rightVal = {{16{value[15]}}, value}  ;
    always_comb
     case(hFunction[5:0])
        `hadd    : {cr, result} = leftOut + rightOut           ;
        `haddcr  : {cr, result} = leftOut + rightOut + crFF    ;
        `hsub    : {cr, result} = leftOut - rightOut           ;
        `hsubcr  : {cr, result} = leftOut - rightOut - crFF    ;
        `hmult   : {cr, result} = {crFF, leftOut * rightOut}   ;
        `hbwand  : {cr, result} = {crFF, leftOut & rightOut}   ;
        `hbwor   : {cr, result} = {crFF, leftOut | rightOut}   ;
        `hbwxor  : {cr, result} = {crFF, leftOut ^ rightOut}   ;
        `haddv   : {cr, result} = leftOut + rightVal           ;
        `haddcrv : {cr, result} = leftOut + rightVal + crFF    ;
        `hsubv   : {cr, result} = leftOut - rightVal           ;
        `hsubcrv : {cr, result} = leftOut - rightVal - crFF    ;
        `hmultv  : {cr, result} = {crFF, leftOut * rightVal}   ;
        `hbwandv : {cr, result} = {crFF, leftOut & rightVal}   ;
        `hbworv  : {cr, result} = {crFF, leftOut | rightVal}   ;
        `hbwxorv : {cr, result} = {crFF, leftOut ^ rightVal}   ;
        `hpsend  : {cr, result} = {crFF, leftOut + !i2hPfull}  ;
```

```
        `hdsend   : {cr, result} = {crFF, leftOut + !i2hDfull}   ;
        `hdget    : {cr, result} = {crFF, leftOut + !i2hDempty}  ;
        default   : {cr, result} = {crFF, 32'bx}                 ;
      endcase
    always_ff @(posedge clock)  crFF <= cr  ;
endmodule
```

Input MUX : `4_hostInMUX.sv` is the Verilog file describing the .

```
/*****************************************************************
File: 4_hostInMUX.sv
Name:
Description:
*****************************************************************/
module hostInMUX (  input   logic [1:0]     loadCom     ,
                    input   logic [31:0]    result      ,
                    input   logic [15:0]    value       ,
                    input   logic [31:0]    procDataIn  ,
                    input   logic [15:0]    leftOutLow  ,
                    output  logic [31:0]    muxOut      );

    always_comb case(loadCom)
                    2'b00: muxOut = {{16{value[15]}}, value};
                    2'b01: muxOut = {leftOutLow, value}     ;
                    2'b10: muxOut = procDataIn              ;
                    2'b11: muxOut = result                  ;
                endcase
endmodule
```

Data Memory: 3_hostDataMemory.sv

```
/*****************************************************************
File: 2_hostDataMemory.sv
Name: Host Data Memory
Description:
*****************************************************************/
module hostDataMemory
        (   output  logic [63:0]            mem2int     ,
            input   logic [63:0]            i2h         ,
            input   logic                   h2iDread    ,
            input   logic [$clog2(`m)-1:0]  transAddr   ,
            output  logic [`n-1:0]          procDataIn  ,
            input   logic [`n-1:0]          procDataOut ,
            input   logic                   procWrite   ,
            input   logic [$clog2(`m):0]    procAddr    ,
            input   logic                   clock       );
    logic [63:0] hDmem[0:`m-1]  ;
    logic [63:0] procData       ;

    always_ff @(posedge clock)
     if (h2iDread) hDmem[transAddr] <= i2h  ;
      else if (procWrite) hDmem[procAddr[$clog2(`m):1]]
             <= procAddr[0] ?
                    {procDataOut, procData[`n-1:0]} :
                    {procData[63:`n], procDataOut}  ;
    assign mem2int = hDmem[transAddr]   ;
    assign procData = hDmem[procAddr[$clog2(`m):1]] ;
    assign procDataIn = procAddr[0] ? procData[63:`n] :
                                      procData[`n-1:0]  ;
endmodule
```

Program Memory: 3_hostProgramMemory.sv

```
/*****************************************************************
File: 2_hostProgramMemory.sv
Name: Host Program memory
Description:
*****************************************************************/
module hostProgramMemory
        (   input   logic [$clog2(`m)-1:0]  hostProgAddr,
            input   logic [31:0]            hostProgram ,
            input   logic                   we          ,
            output  logic [31:0]            instruction ,
            input   logic                   clock       );
    logic [31:0]    hPmem[0:`m-1];
```

```
    always_ff @(posedge clock) begin
      if (we) hPmem[hostProgAddr]   <= hostProgram          ;
              instruction           <= hPmem[hostProgAddr]  ;
    end
endmodule
```

C.1.2 Accelerator: `1_accelerator.sv`

```
/*****************************************************************
File: 1_accelerator.sv
Name: Interfaces
Description:
*****************************************************************/
module accelerator( input   logic           h2iPwrite    ,
                    output  logic           i2hPfull     ,
                    input   logic [63:0]    h2i          ,
                    input   logic           h2iDwrite    ,
                    output  logic           i2hDfull     ,
                    output  logic [63:0]    i2h          ,
                    input   logic           h2iDread     ,
                    output  logic           i2hDempty    ,
                    output  logic           a2hInt       ,
                    input   logic           h2aInta      ,
                    input   logic           reset        ,
                    input   logic           clock        );
    logic [63:0]    i2aData ;
    logic [63:0]    a2i     ;
    logic [63:0]    i2aProg ;

    interfaces INTERFACES(  h2iPwrite    ,
                            i2hPfull     ,
                            h2i          ,
                            h2iDwrite    ,
                            i2hDfull     ,
                            i2h          ,
                            h2iDread     ,
                            i2hDempty    ,
                            i2aProg      ,
                            a2iPread     ,
                            i2aPempty    ,
                            i2aData      ,
                            a2iDread     ,
                            i2aDempty    ,
                            a2i          ,
                            a2iDwrite    ,
                            i2aDfull     ,
                            reset        ,
```

```
                        clock       );
    pRISC pRISC(i2aProg     ,
                a2iPread    ,
                i2aPempty   ,
                i2aData     ,
                a2iDread    ,
                i2aDempty   ,
                a2i         ,
                a2iDwrite   ,
                i2aDfull    ,
                a2hInt      ,
                h2aInta     ,
                reset       ,
                clock       );
endmodule
```

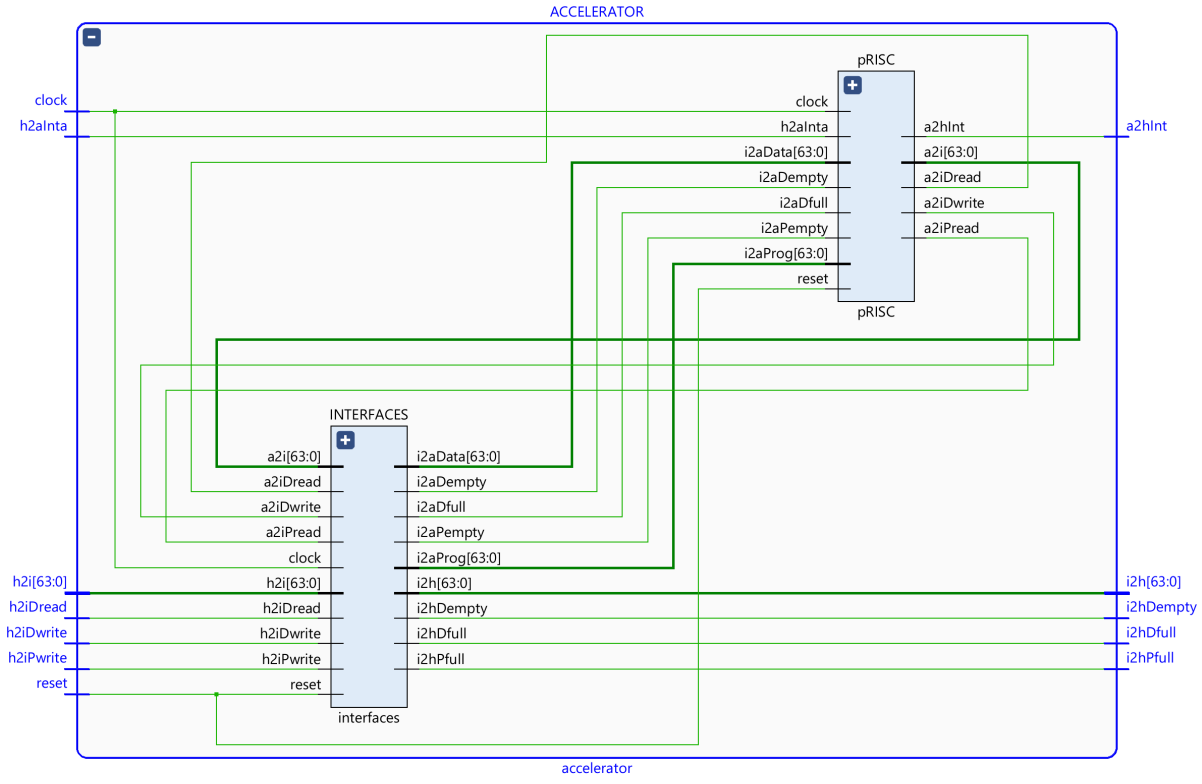

Figure C.2: The accelerator.

Interfaces: 2_interfaces.sv

```
/*****************************************************************************
```

```
File: 2_interfaces.sv
Name: Interfaces
Description:
*********************************************************************/
module interfaces(  input   logic           h2iPwrite   ,
                    output  logic           i2hPfull    ,
                    input   logic [63:0]    h2i         ,
                    input   logic           h2iDwrite   ,
                    output  logic           i2hDfull    ,
                    output  logic [63:0]    i2h         ,
                    input   logic           h2iDread    ,
                    output  logic           i2hDempty   ,
                    output  logic [63:0]    i2aProg     ,
                    input   logic           a2iPread    ,
                    output  logic           i2aPempty   ,
                    output  logic [63:0]    i2aData     ,
                    input   logic           a2iDread    ,
                    output  logic           i2aDempty   ,
                    input   logic [63:0]    a2i         ,
                    input   logic           a2iDwrite   ,
                    output  logic           i2aDfull    ,
                    input   logic           reset, clock);

    fifo programFIFO(   .dataIn (h2i        ),
                        .write  (h2iPwrite  ),
                        .full   (i2hPfull   ),
                        .dataOut(i2aProg    ),
                        .read   (a2iPread   ),
                        .empty  (i2aPempty  ),
                        .reset  (reset      ),
                        .clock  (clock      ));
    fifo inputDataFIFO( .dataIn (h2i        ),
                        .write  (h2iDwrite  ),
                        .full   (i2hDfull   ),
                        .dataOut(i2aData    ),
                        .read   (a2iDread   ),
                        .empty  (i2aDempty  ),
                        .reset  (reset      ),
                        .clock  (clock      ));
    fifo outputDataFIFO(.dataIn (a2i        ),
                        .write  (a2iDwrite  ),
                        .full   (i2aDfull   ),
                        .dataOut(i2h        ),
                        .read   (h2iDread   ),
                        .empty  (i2hDempty  ),
                        .reset  (reset      ),
                        .clock  (clock      ));
endmodule
```

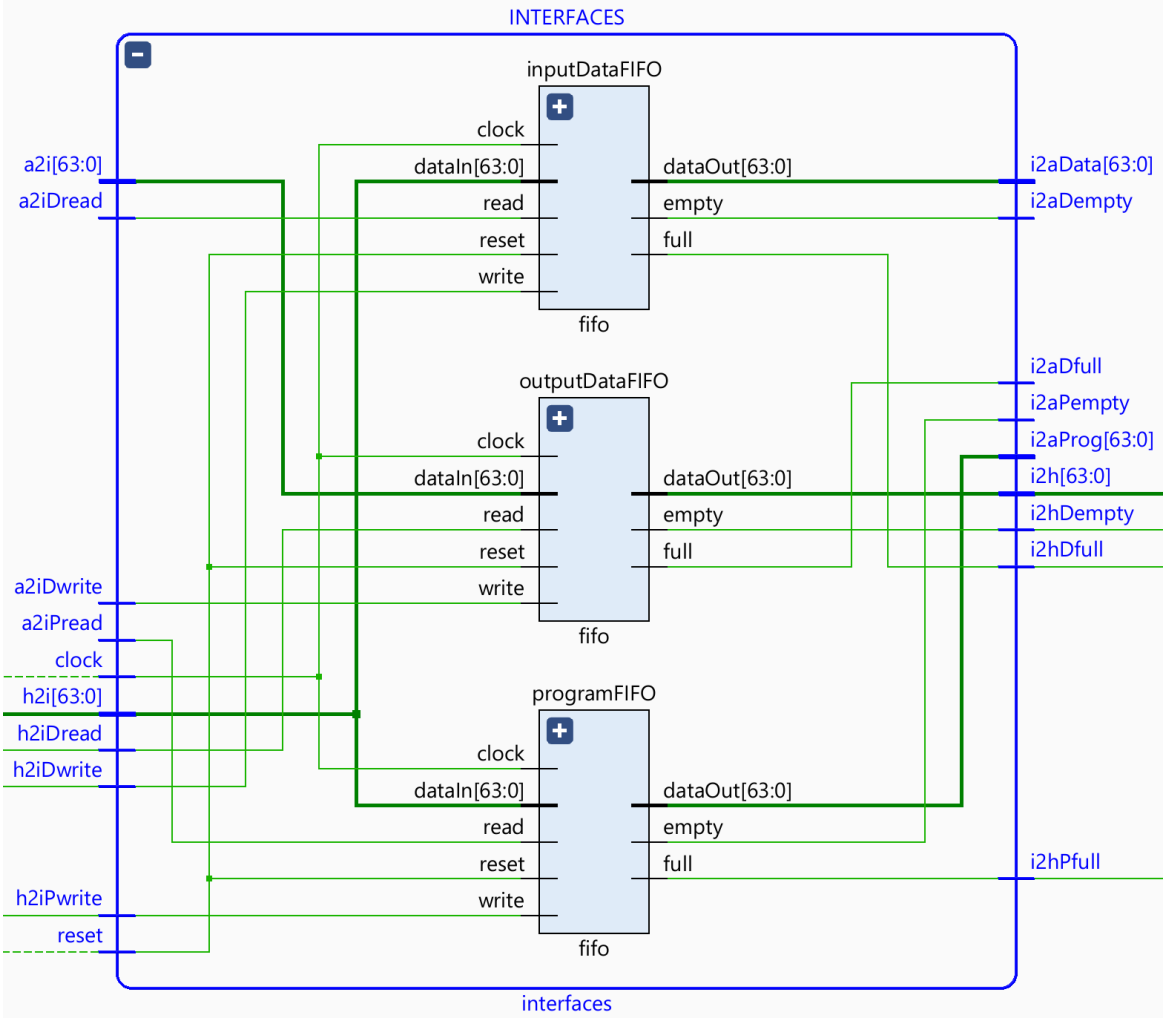

Figure C.3: Interfaces.

FIFO: `3_fifo.sv` the synchronous version

```
/*****************************************************************************
File: 3_fifo.sv
Name: FIFO
Description:
*****************************************************************************/
module fifo(input   logic [63:0]    dataIn  ,
            input   logic           write   ,
            output  logic           full    ,
            output  logic [63:0]    dataOut ,
            input   logic           read    ,
            output  logic           empty   ,
```

```
            input   logic           reset   ,
            input   logic           clock   );
    logic [63:0]    ram[0:255]  ;
    logic [8:0] rA, wA      ;

    assign empty = (rA[7:0] == wA[7:0]) & ~(rA[8] ^ wA[8]);
    assign full =  (rA[7:0] == wA[7:0]) &  (rA[8] ^ wA[8]);
    always_ff @(posedge clock)
        if (write && ~full) ram[wA[7:0]] <= dataIn  ;
    assign dataOut = ram[rA[7:0]]   ;
    always_ff @(posedge clock)
        if (reset)  begin wA <= 0   ;
                          rA <= 0   ;
                    end
         else begin if (write && ~full) wA <= wA + 1;
                    if (read && ~empty) rA <= rA + 1;
              end
endmodule
```

pRISC: `2_pRISC.sv`

```
/*****************************************************************
File: 2_pRISC.sv
Name: Accelerator
Description:
*****************************************************************/
module pRISC(   input   logic [63:0]    i2aProg     ,
                output  logic           a2iPread    ,
                input   logic           i2aPempty   ,
                input   logic [63:0]    i2aData     ,
                output  logic           a2iDread    ,
                input   logic           i2aDempty   ,
                output  logic [63:0]    a2i         ,
                output  logic           a2iDwrite   ,
                input   logic           i2aDfull    ,
                output  logic           a2hInt      ,
                input   logic           h2aInta     ,
                input   logic           reset, clock);
    logic [2:0]     dataTransCom;
    logic [`n+12:0] contr2array ;
    logic [`n-1:0]  red         ;

    controller controller(  i2aProg     ,
                            a2iPread    ,
                            i2aPempty   ,
                            dataTransCom,
                            a2iDread    ,
```

```
                                i2aDempty    ,
                                a2iDwrite    ,
                                i2aDfull     ,
                                contr2array  ,
                                red          ,
                                a2hInt       ,
                                h2aInta      ,
                                reset, clock);
    array array(i2aData      ,
                a2i          ,
                dataTransCom ,
                contr2array  ,
                red          ,
                clock        );
endmodule
```

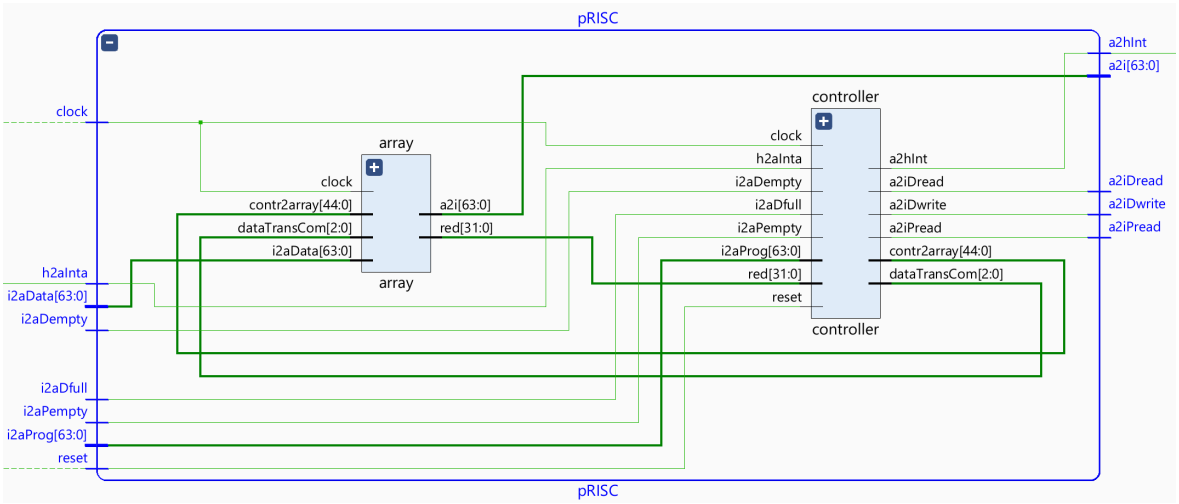

Figure C.4: pRISC.

```
/*****************************************************************************
File: 3_controller.sv
Name: Controller
Description:
*****************************************************************************/
module controller( input  logic [63:0]  i2aProg      ,
                  output logic          a2iPread     ,
                  input  logic          i2aPempty    ,
                  output logic [2:0]    dataTransCom ,
                  output logic          a2iDread     ,
                  input  logic          i2aDempty    ,
```

```
                output  logic             a2iDwrite    ,
                input   logic             i2aDfull     ,
                output  logic [`n+12:0]   contr2array  ,
                input   logic [`n-1:0]    red          ,
                output  logic             a2hInt       ,
                input   logic             h2aInta      ,
                input   logic             reset        ,
                input   logic             clock        );
logic [$clog2(`m)-1:0]  progAddr    ;
logic [63:0]            dualInstr   ;
logic [63:0]            delayedInstr;
logic [31:0]            contrMemIn  ;
logic [31:0]            contrMemOut ;
logic [1:0]             accState    ;
logic                   initFunc    ;
logic                   getParam    ;

controller2distribute contr2distr(
    .contr2array    (contr2array    ),
    .contrMemIn     (contrMemIn     ),
    .delayedInstr   (delayedInstr   ),
    .clock          (clock          ));
controllerProgramMemory contrProgMem(   i2aProg     ,
                                        a2iPread    ,
                                        i2aPempty   ,
                                        progAddr    ,
                                        dualInstr   ,
                                        delayedInstr,
                                        accState    ,
                                        initFunc    ,
                                        getParam    ,
                                        reset, clock);
controllerDataMemory contrDataMem(
    .contrDataAddr   (dualInstr[$clog2(`m)-1:0]     ),
    .contrDelDataAddr(delayedInstr[$clog2(`m)-1:0]  ),
    .contrMemOut     (contrMemOut                   ),
    .contrMemIn      (contrMemIn                    ),
    .contrMemCom     (dualInstr[31:24]              ),
    .contrDelMemCom  (delayedInstr[31:19]           ),
    .clock           (clock                         ));
controllerProcessor contrProc(
    .progAddr       (progAddr           ),
    .instr          (dualInstr[31:0]    ),
    .delayedInstr   (delayedInstr[31:0] ),
    .contrMemOut    (contrMemOut        ),
    .contrMemIn     (contrMemIn         ),
    .dataTransCom   (dataTransCom       ),
    .a2iDread       (a2iDread           ),
    .i2aDempty      (i2aDempty          ),
    .a2iDwrite      (a2iDwrite          ),
```

```
        .i2aDfull       (i2aDfull           ),
        .red2contr      (red                ),
        .a2hInt         (a2hInt             ),
        .h2aInta        (h2aInta            ),
        .accState       (accState           ),
        .initFunc       (initFunc           ),
        .getParam       (getParam           ),
        .int2accProg    (i2aProg[31:0]      ),
        .reset          (reset              ),
        .clock          (clock              ));
endmodule
```

Controller: 3_controller.sv

Controller Processor: `4_controllerProcessor.sv` is the Verilog file describing CONTROLLER's processor.

```
/*****************************************************************
File: 4_controllerProcessor.sv
Name: Controller Processor
Description:
*****************************************************************/
module controllerProcessor (
        output  logic [$clog2(`m)-1:0]  progAddr     ,
        input   logic [31:0]            instr        ,
        input   logic [31:0]            delayedInstr ,
        input   logic [31:0]            contrMemOut  ,
        output  logic [31:0]            contrMemIn   ,
        output  logic [2:0]             dataTransCom ,
        output  logic                   a2iDread     ,
        input   logic                   i2aDempty    ,
        output  logic                   a2iDwrite    ,
        input   logic                   i2aDfull     ,
        input   logic [`n-1:0]          red2contr    ,
        output  logic                   a2hInt       ,
        input   logic                   h2aInta      ,
        input   logic [1:0]             accState     ,
        input   logic                   initFunc     ,
        input   logic                   getParam     ,
        input   logic [31:0]            int2accProg  ,
        input   logic                   reset, clock);
    logic endTransfer   ;
    logic zero          ;

    controllerCom contCom(
        .a2hInt         (a2hInt         ),
        .h2aInta        (h2aInta        ),
        .instr          (instr          ),
        .dataTransCom   (dataTransCom   ),
        .endTransfer    (endTransfer    ),
        .a2iDread       (a2iDread       ),
        .i2aDempty      (i2aDempty      ),
        .a2iDwrite      (a2iDwrite      ),
        .i2aDfull       (i2aDfull       ),
        .reset          (reset          ),
        .clock          (clock          ));
    controllerPC contrPC(
        .progAddr   (progAddr                   ),
        .instr      (instr                      ),
        .zero       (zero                       ),
        .endTransfer(endTransfer                ),
        .accState   (accState                   ),
        .initFunc   (initFunc                   ),
        .initAddr   (int2accProg[$clog2(`m)-1:0]),
```

```
        .reset       (reset                          ),
        .clock       (clock                          ));
    controllerAALU contAALU(
        .instruction(delayedInstr    ),
        .contrMemOut(contrMemOut     ),
        .red2contr  (red2contr       ),
        .int2accProg(int2accProg     ),
        .cAcc       (contrMemIn      ),
        .zero       (zero            ),
        .clock      (clock           ));
endmodule
```

Controller Data Memory: `4_controllerDataMemory.sv` is the System Verilog file describing CONTROLLER's data memory.

```
/*************************************************************************
File:4_controllerDataMemory.sv
Name: Controller Data Memory
Description:
*************************************************************************/
module controllerDataMemory (
        input   logic [$clog2(`m)-1:0]  contrDataAddr    ,
        input   logic [$clog2(`m)-1:0]  contrDelDataAddr ,
        output  logic [`n-1:0]          contrMemOut      ,
        input   logic [`n-1:0]          contrMemIn       ,
        input   logic [7:0]             contrMemCom      ,
        input   logic [12:0]            contrDelMemCom   ,
        input   logic                   clock            );
// memCom: 00:nop; 01:absolute; 10:rel; 11:rel+update
    logic [31:0]            contrDM[0:`m-1] ;
    logic [$clog2(`m)-1:0]  contrAddrReg    ;
    logic [1:0]             wCom            ;
    logic [1:0]             rCom            ;
    logic [$clog2(`m)-1:0]  wAddr           ;
    logic [$clog2(`m)-1:0]  rAddr           ;
    logic                   addrLoad        ;

// Transcode:
//memCom: 00:nop; 01:abs; 10:rel; 11:rel+update
    always_comb case(contrDelMemCom[12:5])
                    {`cstore,`dir}: wCom = 2'b01;
                    {`cstore,`cdr}: wCom = 2'b01;
                    {`cstore,`rel}: wCom = 2'b10;
                    {`cstore,`crl}: wCom = 2'b10;
                    {`cstore,`rei}: wCom = 2'b11;
                    {`cstore,`cri}: wCom = 2'b11;
                    default:        wCom = 2'b00;
                endcase
    always_comb if (contrMemCom[7:3] == `cstore)
                    rCom = 2'b00;
                  else  case(contrMemCom[2:0])
                            `dir:   rCom = 2'b01;
                            `cdr:   rCom = 2'b01;
                            `rel:   rCom = 2'b10;
                            `crl:   rCom = 2'b10;
                            `rei:   rCom = 2'b11;
                            `cri:   rCom = 2'b11;
                            default:rCom = 2'b00;
                        endcase
    assign addrLoad =
            {`contr, `imm, `crela} == contrDelMemCom;
// Address compute
```

```
    always_comb
     case(wCom[1:0])
        2'b01: wAddr = contrDelDataAddr                  ;
        2'b10: wAddr = contrAddrReg + contrDelDataAddr   ;
        2'b11: wAddr = contrAddrReg + contrDelDataAddr   ;
      default: wAddr = contrDelDataAddr                  ;
     endcase
    always_comb
     case(rCom[1:0])
        2'b01: rAddr = contrDataAddr                  ;
        2'b10: rAddr = contrAddrReg + contrDataAddr   ;
        2'b11: rAddr = contrAddrReg + contrDataAddr   ;
      default: rAddr = contrDataAddr                  ;
     endcase
// Operation
    always_ff @(posedge clock)  begin
     if (|rCom)      contrMemOut     <= contrDM[rAddr]    ;
     if (|wCom)      contrDM[wAddr]  <= contrMemIn        ;
     if (&wCom)      contrAddrReg
                     <= contrAddrReg + contrDelDataAddr   ;
     if (&rCom)      contrAddrReg
                     <= contrAddrReg + contrDataAddr      ;
     if (addrLoad)   contrAddrReg
                     <= contrMemIn[$clog2(`m)-1:0]        ;
    end
endmodule
```

Controller Program Memory: `4_controllerProgramMemory.sv` is the Verilog file describing CONTROLLER's program memory.

```
/*****************************************************************
File: 4_controllerProgramMemory.sv
Name: Controller Program Memory
Description:
*****************************************************************/
module controllerProgramMemory(
        input   logic [63:0]            i2aProg     ,
        output  logic                   a2iPread    ,
        input   logic                   i2aPempty   ,
        input   logic [$clog2(`m)-1:0]  progAddr    ,
        output  logic [63:0]            dualInstr   ,
        output  logic [63:0]            delayedInstr,
        output  logic [1:0]             accState    ,
        output  logic                   initFunc    ,
        output  logic                   getParam    ,
        input   logic                   reset, clock);
    logic [63:0]            contrPM[0:`m-1] ;
```

```
    logic [$clog2(`m)-1:0]  loadReg          ;

    always_ff @(posedge clock)
        if (reset)  accState <= `idle    ;
         else begin
                if ((accState == `idle) && !i2aPempty &&
                    (i2aProg[31:24] == {`contr, `imm}) &&
                    (i2aProg[23:19] == `pload))
                    accState <= `loading;
                if ((accState == `idle) && !i2aPempty &&
                    (i2aProg[31:24] == {`contr, `imm}) &&
                    (i2aProg[23:19] == `prun))
                    accState <= `running;
                if ((accState == `loading) && !i2aPempty &&
                    (i2aProg[31:24] == {`contr, `imm}) &&
                    (i2aProg[23:19] == `prun))
                    accState <= `running;
                if ((accState == `running) &&
                    (dualInstr[31:24] == {`contr, `imm}) &&
                    (dualInstr[23:19] == `halt))
                    accState <= `idle    ;
            end
    assign initFunc =
                (accState == `idle) && !i2aPempty &&
                (i2aProg[31:24] == {`contr, `imm}) &&
                (i2aProg[23:19] == `prun)                ;
    assign getParam =
                (accState == `running) && !i2aPempty &&
                (delayedInstr[31:24] == {`contr, `imm}) &&
                (delayedInstr[23:19] == `param)              ;
    always_ff @(posedge clock)  begin
        if (accState == `idle)
                loadReg <= i2aProg[$clog2(`m)-1:0]  ;
        if ((accState == `loading) && !i2aPempty)
                loadReg <= loadReg + 1                   ;
        if (accState == `running)
                loadReg <= loadReg                       ;
        end
    always_ff @(posedge clock)
        if ((accState == `loading) && !i2aPempty)
                contrPM[loadReg] <= i2aProg ;

    always_ff @(posedge clock)  begin
        dualInstr    <= (accState == `running) ?
                        contrPM[progAddr] : 0   ;
        delayedInstr <= dualInstr               ;
    end
    assign a2iPread  = ((accState == `loading) && !i2aPempty)
                        || initFunc || getParam ;
endmodule
```

Controller to Array: 4_controller2distribute.sv is the Verilog file describing the connection between CONTROLLER and ARRAY.

```
/*****************************************************************
File: 4_controller2distribute.sv
Name: Controller to Array Distribute
Description:
*****************************************************************/
module controller2distribute(
        output  logic [`n+12:0] contr2array ,
        input   logic [31:0]    contrMemIn  ,
        input   logic [63:0]    delayedInstr,
        input   logic           clock       );
    logic [1:0] dsh[0:$clog2(`p)]   ;
    logic [1:0] redIns              ;
    logic [2:0] shift               ;
    integer i   ;

    assign redIns =
           ((delayedInstr[31:19] == {`contr, `imm, `gshift}) &&
             delayedInstr[2]) ?
           {1'b1, delayedInstr[0]} : 2'b00 ;
    assign shift =
           ((delayedInstr[31:19] == {`contr, `imm, `gshift}) &&
             !delayedInstr[2]) ?
           {1'b1, delayedInstr[1:0]} : 3'b000  ;
    always_ff @(posedge clock) for(i=0; i<$clog2(`p)+1; i=i+1)
                               dsh[i] <= (i==0) ? redIns : dsh[i-1];
    assign contr2array =
        {dsh[$clog2(`p)],
         shift,
         delayedInstr[63:56],
            (((delayedInstr[58:56] == `cim) |
              (delayedInstr[58:56] == `cdr) |
              (delayedInstr[58:56] == `crl) |
              (delayedInstr[58:56] == `cri)) ?
                contrMemIn :
                {{8{delayedInstr[55]}}, delayedInstr[55:32]})};
endmodule
```

Array of Cells: 3_array.sv are

```
/*****************************************************************
```

```
File: 3_array.sv
Name: Array
Description:
*********************************************************************/
module array(   input   logic [63:0]    i2aData      ,
                output  logic [63:0]    a2i          ,
                input   logic [2:0]     dataTransCom ,
                input   logic [`n+12:0] contr2array  ,
                output  logic [`n-1:0]  red          ,
                input   logic           clock        );
// dataTransCom : 000 : nop
// dataTransCom : 100 : even insert
// dataTransCom : 101 : odd insert
// dataTransCom : 110 : even extract
// dataTransCom : 111 : odd extract
    logic [`n+1:0]  map2reduce[0:`p-1]  ; // func[1:0], acc
    logic [0:`p-1]  boolean             ;
    logic [0:`p-1]  first               ;
    logic [0:`p-1]  next                ;
    logic [`n+1:0]  red2contr           ;

    map map(.int2accData    (i2aData        ),
            .a2i            (a2i            ),
            .dataTransCom   (dataTransCom   ),
            .contr2array    (contr2array    ),
            .map2reduce     (map2reduce     ),
            .red            (red            ),
            .boolean        (boolean        ),
            .first          (first          ),
            .next           (next           ),
            .clock          (clock          ));
    scan SCAN(  boolean ,
                first   ,
                next    ,
                clock   );
    reduce REDUCE(  red2contr   ,
                    map2reduce  ,
                    clock       );

    assign red = red2contr[`n-1:0]  ;
endmodule
```

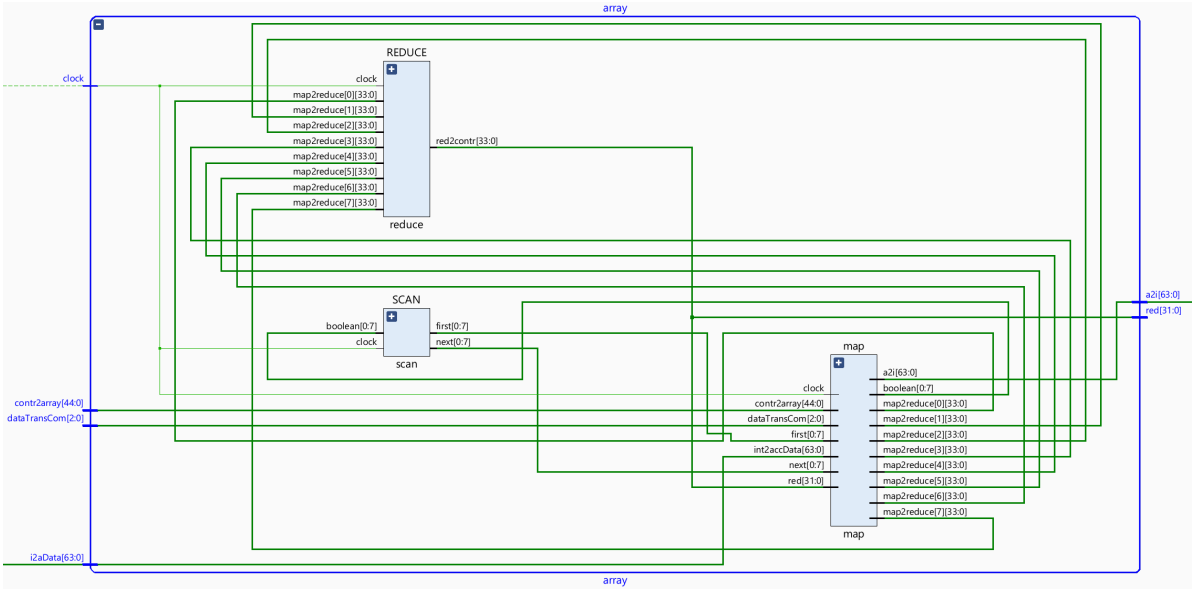

Figure C.5: Array for $p = 8$.

Map: `4_map.sv` is the Verilog file describing the MAP section in ARRAY.

```
/*****************************************************************************
File: 4_map.sv
Name: Map section
Description:
*****************************************************************************/
module map( input   logic [63:0]     int2accData          ,
            output  logic [63:0]     a2i                  ,
            input   logic [2:0]      dataTransCom         ,
            input   logic [`n+12:0]  contr2array          ,
            output  logic [`n+1:0]   map2reduce[0:`p-1]    ,
            input   logic [`n-1:0]   red                  ,
            output  logic [0:`p-1]   boolean              ,
            input   logic [0:`p-1]   first                ,
            input   logic [0:`p-1]   next                 ,
            input   logic            clock                );
// dataTransCom : 000 : nop
// dataTransCom : 100 : even insert
// dataTransCom : 101 : odd insert
// dataTransCom : 110 : even extract
// dataTransCom : 111 : odd extract
    logic [`n:0]        leftEndIn           ;
    logic [`n-1:0]      rightEndIn          ;
    logic [0:`p/2-1]    boolean0            ;
    logic [0:`p/2-1]    boolean1            ;
```

```
logic [0:`p/2-1]    first0              ;
logic [0:`p/2-1]    first1              ;
logic [0:`p/2-1]    next0               ;
logic [0:`p/2-1]    next1               ;
logic [`n:0]        rightOut0[0:`p/4-1] ;
logic [`n-1:0]      leftOut0[0:`p/4-1]  ;
logic [`n:0]        rightOut1[0:`p/4-1] ;
logic [`n-1:0]      leftOut1[0:`p/4-1]  ;
logic               insSerial           ;
logic [1:0]         rotateSelLeft       ;
logic [1:0]         rotateSelRight      ;
logic [63:0]        acc2intData0        ;
logic [63:0]        acc2intData1        ;
integer i, j, k ;

always_comb for(i=0; i<`p; i= i+1)
    case(i[1:0])
        2'b00: boolean[i] = boolean0[2*(i/4)]   ;
        2'b01: boolean[i] = boolean0[2*(i/4)+1] ;
        2'b10: boolean[i] = boolean1[2*(i/4)]   ;
        2'b11: boolean[i] = boolean1[2*(i/4)+1] ;
    endcase
always_comb for(j=0; j<`p/4; j= j+1) begin
    {first0[2*j], first0[2*j+1]} =
    {first[4*j],  first[4*j+1]}     ;
    {first1[2*j], first1[2*j+1]} =
    {first[4*j+2],first[4*j+3]}     ;
end
always_comb for(k=0; k<`p/4; k= k+1) begin
    {next0[2*k], next0[2*k+1]} =
    {next[4*k],  next[4*k+1]}   ;
    {next1[2*k], next1[2*k+1]} =
    {next[4*k+2],next[4*k+3]}   ;
end
always_comb
    case(rotateSelLeft)
        2'b00: leftEndIn = {`n+1{1'b0}}                         ;
        2'b01: leftEndIn = {1'b0, rightOut1[`p/4-1][`n-1:0]};
        2'b10: leftEndIn = {1'b0, red}                          ;
        2'b11: leftEndIn = {`n+1{1'b0}}                         ;
    endcase
always_comb
    case(rotateSelRight)
        2'b00: rightEndIn = {`n{1'b0}}  ;
        2'b01: rightEndIn = leftOut0[0] ;
        2'b10: rightEndIn = red         ;
        2'b11: rightEndIn = {`n{1'b0}}  ;
    endcase
assign a2i = dataTransCom[0] ? acc2intData0 : acc2intData1  ;
column col0(
```

```
      .int2accData      (int2accData                              ),
      .acc2intData      (acc2intData1                             ),
      .dataTransCom     (dataTransCom                             ),
      .map2reduce       (map2reduce[0:`p/2-1]                     ),
      .boolean          (boolean0                                 ),
      .first            (first0                                   ),
      .next             (next0                                    ),
      .rightOut         (rightOut0[0:`p/4-1]                      ),
      .leftIn           ({leftEndIn, rightOut1[0:`p/4-2]}         ),
      .leftOut          (leftOut0[0:`p/4-1]                       ),
      .rightIn          (leftOut1[0:`p/4-1]                       ),
      .contr2array      (contr2array                              ),
      .rotateSelLeft    (rotateSelLeft                            ),
      .rotateSelRight   (                                         ),
      .index            (1'b0                                     ),
      .clock            (clock                                    ));
    column col1(
      .int2accData      (int2accData                              ),
      .acc2intData      (acc2intData0                             ),
      .dataTransCom     ({dataTransCom[2:1],!dataTransCom[0]}),
      .map2reduce       (map2reduce[`p/2:`p-1]                    ),
      .boolean          (boolean1                                 ),
      .first            (first1                                   ),
      .next             (next1                                    ),
      .rightOut         (rightOut1[0:`p/4-1]                      ),
      .leftIn           (rightOut0[0:`p/4-1]                      ),
      .leftOut          (leftOut1[0:`p/4-1]                       ),
      .rightIn          ({leftOut0[1:`p/4-1], rightEndIn}         ),
      .contr2array      (contr2array                              ),
      .rotateSelLeft    (                                         ),
      .rotateSelRight   (rotateSelRight                           ),
      .index            (1'b1                                     ),
      .clock            (clock                                    ));
endmodule
```

Scan: `4_scan.sv` is the Verilog file describing the SCAN section in ARRAY.

```
/*****************************************************************
File: 4_scan.sv
Name: Scan Network
Description:
*****************************************************************/
module scan (   input   logic [0:`p-1]  boolean ,
                output  logic [0:`p-1]  first   ,
                output  logic [0:`p-1]  next    ,
                input   logic           clock   );
    logic [0:`p-1] out;
```

```
    orPrefix     orPrefix(    .in (boolean),
                              .out(out     ));
    always_ff @(posedge clock) begin
        first    <= out & ~(out >> 1);
        next     <= (out >> 1)       ;
    end
endmodule
endmodule
```

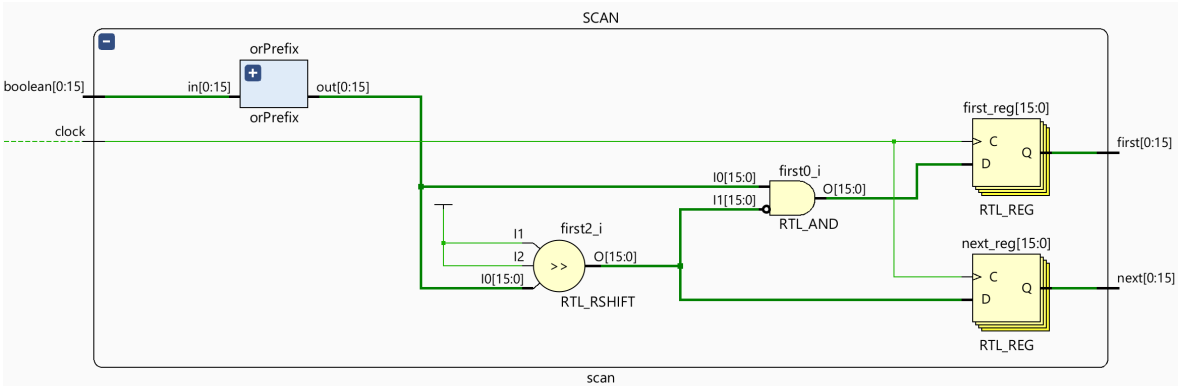

Figure C.6: Scan network.

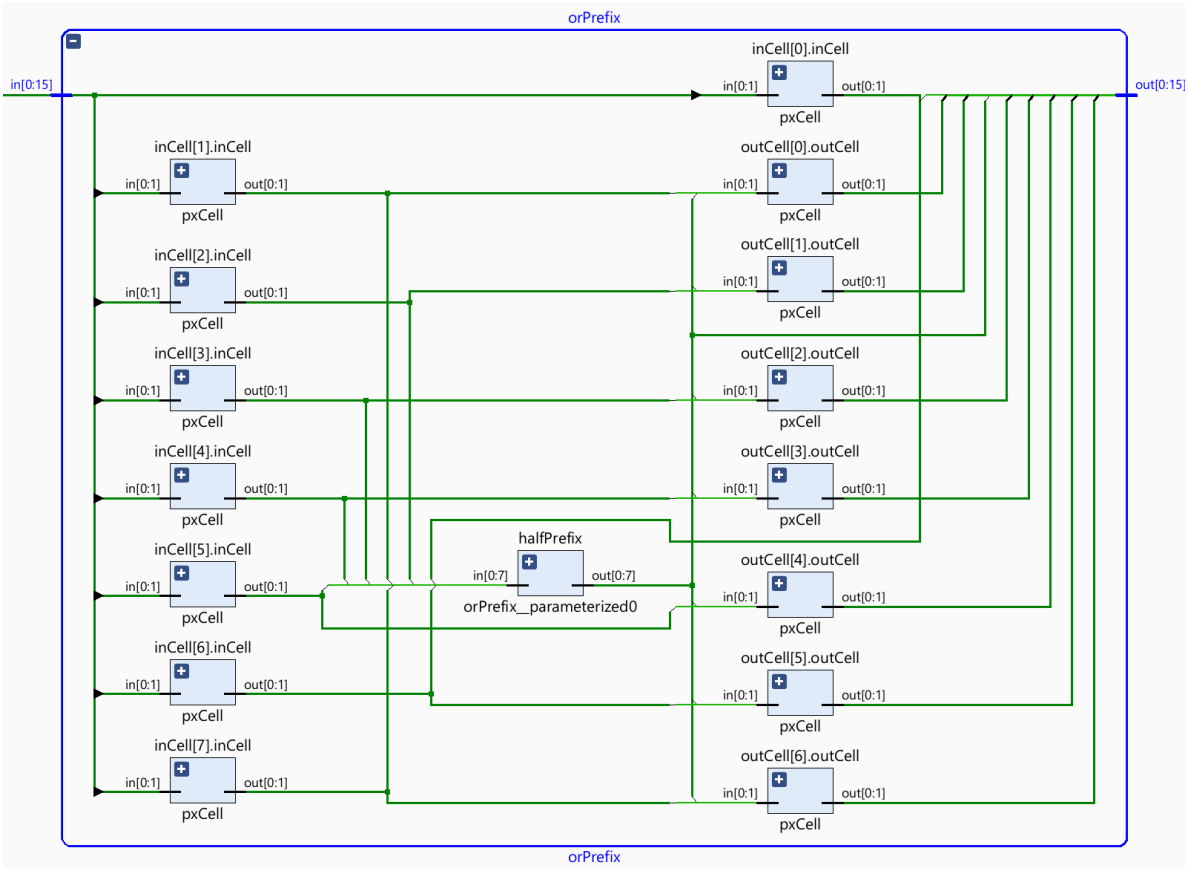

Figure C.7: OR prefix network.

Reduce: `4_reduce.sv` is the Verilog file describing the REDUCE section in ARRAY.

```
/*****************************************************************************
File: 4_reduce.sv
Name: Reduce Network
Description:
*****************************************************************************/
module reduce #(parameter P=`p)(
        output  logic [`n+1:0]  red2contr           ,
        input   logic [`n+1:0]  map2reduce[0:P-1]   ,
        input   logic           clock               );
   logic [`n+1:0]   out0, out1  ;

   genvar   i   ;
   generate
    if (P == 2) eRed    ered    (   map2reduce[0:1] ,
                                    red2contr       ,
                                    clock           );
     else begin
        reduce #(.P(P/2))
```

```
            reduce0(.red2contr  (out0                ),
                    .map2reduce (map2reduce[P/2:P-1]),
                    .clock      (clock               )),
            reduce1(.red2contr  (out1                ),
                    .map2reduce (map2reduce[0:P/2-1]),
                    .clock      (clock               ));
        eRed ereduce(    .in    ({out0, out1}        ),
                         .out   (red2contr           ),
                         .clock (clock               ));
    end
  endgenerate
endmodule
```

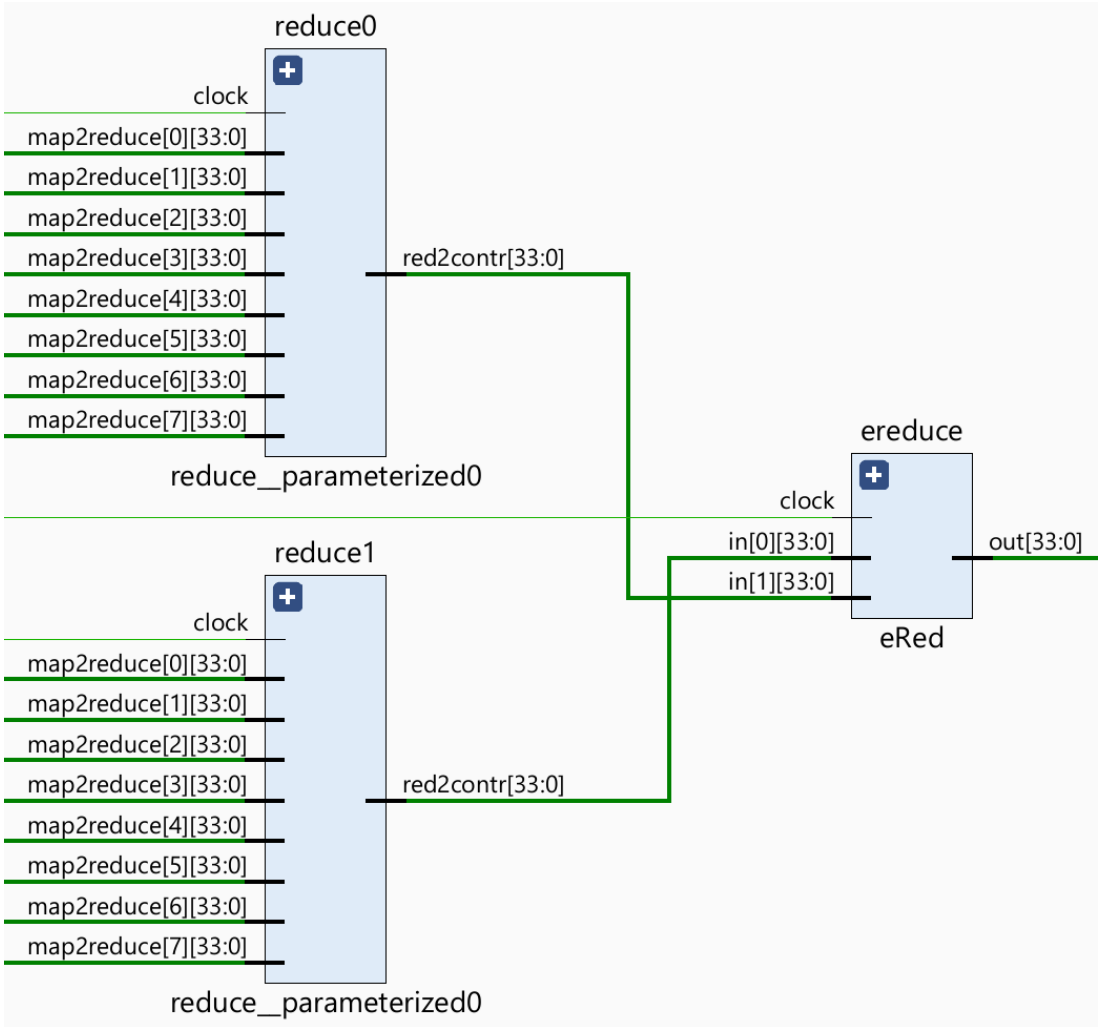

Figure C.8: Reduce network.

C.2 Heterogeneous Architecture

The lowest level of the architecture is the micro-architecture used to define in the file `0_DEFINE`:

- the micro-operations performed by HOST
- the micro-operations performed by CONTROLLER
- the micro-operations performed by each cell in MAP array
- the main parameters of the ACCELERATOR:
  - n: the size of the word used in cells
  - p: the number of cells in MAP array
  - m: the size of local data memory (register file) in each cell

The next level in defining the architecture is Instruction Set Architecture by specifying the way the instructions are generated using the micro-architecture. There are two code generators defined in the following System Verilog files::

0_hostCodeGenerator.sv : defines the instruction code used to generate the binary form of the program executed by HOST

0_accCodeGenerator.sv : defines the instruction code used to generate the binary form of the program executed by CONTROLLER and ARRAY of cells

C.2.1 Micro-architecture: `0_DEFINES.vh`

```
/*************************************************************************
File name: 0_DEFINES.vh
*************************************************************************/
`define n (32)       // internal word
`define p (16)       // number of cells
`define m (1024)     // memory size

                     ////////
                     // HOST //
                     ////////

/*************************************************************************
Name: Host's instruction set architecture

instr={opCode[5:0],dest[4:0],left[4:0],right[4:0],nu[10:0]} |
      {opCode[5:0],dest[4:0],left[4:0],immue[15:0]}
reg [31:0]                rf[0:31]
reg [$clog2(`m)-1:0]      pc
reg                       cr
*************************************************************************/
```

```
// CONTROL INSTRUCTIONS
`define hnop      6'b000000 // pc <= pc+1
`define hjmp      6'b000001 // pc <= pc+immue
`define hjmpz     6'b000010 // pc <= (rf[left]==0) ? pc+immue : pc+1
`define hjmpnz    6'b000011 // pc <= !(rf[left]==0) ? pc+immue : pc+1
`define hpsend    6'b100000 // pc <= (host2int[31:24]==
                            // {`prun,`ctl}) ? pc + 1 : pc
                            // h2iPwrite = 1
                            // rf[left] <= rf[left]+!i2hPfull
`define hdget     6'b100010 // pc <= (leftOut==rightOut) ?
                            // pc + 1 : pc
                            // i2hDread = 1
                            // rf[dest] <= rf[left]+!i2hDfull
`define hdsend    6'b100001 // pc <= (leftOut==rightOut) ?
                            // pc + 1 : pc
                            // h2iDwrite = 1
                            // rf[left] <= rf[left]+!i2hDfull
`define hintwait  6'b000111 // pc <= (int) ? pc+1 : pc
`define hajmp     6'b001000 // pc <= immue
`define hhalt     6'b001001 // pc <= pc
`define hsttcc    6'b001010 // start host cycle counter
`define hstpcc    6'b001011 // stop host cycle counter

// FUNCTIONAL INSTRUCTIONS
`define hadd      6'b010000 // {cr,rf[dest]} <= rf[left]+rf[right]
`define haddcr    6'b010001 // {cr,rf[dest]} <= rf[left]+rf[right]+cr
`define hsub      6'b010010 // {cr,rf[dest]} <= rf[left]-rf[right]
`define hsubcr    6'b010011 // {cr,rf[dest]} <= rf[left]-rf[right]-cr
`define hmult     6'b010100 // {cr,rf[dest]} <= rf[left]*rf[right]
`define hbwand    6'b010101 // {cr,rf[dest]} <= {cr, rf[left] & rf[right]}
`define hbwor     6'b010110 // {cr,rf[dest]} <= {cr, rf[left] | rf[right]}
`define hbwxor    6'b010111 // {cr,rf[dest]} <= {cr, rf[left] ^ rf[right]}
`define haddv     6'b011000
    // {cr,rf[dest]} <= rf[left]+{{16{immue[15]}},immue}
`define haddcrv   6'b011001
    // {cr,rf[dest]} <= rf[left]+{{16{immue[15]}},immue}+cr
`define hsubv     6'b011010
    // {cr,rf[dest]} <= rf[left]-{{16{immue[15]}},immue}
`define hsubcrv   6'b011011
    // {cr,rf[dest]} <= rf[left]-{{16{immue[15]}},immue}-cr
`define hmultv    6'b011100
    // {cr,rf[dest]} <= rf[left]*{{16{immue[15]}},immue}
`define hbwandv   6'b011101
    // {cr,rf[dest]} <= {cr,rf[left]&{{16{immue[15]}},immue}}
`define hbworv    6'b011110
    // {cr,rf[dest]} <= {cr,rf[left]|{{16{immue[15]}},immue}}
`define hbwxorv   6'b011111
    // {cr,rf[dest]} <= {cr,rf[left]^{{16{immue[15]}},immue}}

// DATA TRANSFER INSTRUCTIONS
```

```
`define hfsend   6'b100011 // function send:
    // {code[5:0], `prun, `ctl, initAddress}
`define hssend   6'b100100 // scalar send: {code[5:0], scalar}
`define hvalue   6'b101000 // {cr,rf[dest]} <= {cr,{{16{immue[15]}},immue}}
`define hinsval  6'b101001 // {cr,rf[dest]} <= {cr,{rf[left][15:0],immue}
`define hload    6'b101010 // {cr,rf[dest]} <= {cr,hDmem[rf[left]}
`define hstore   6'b111011 // hDmem[rf[left]] <= rf[right]


                //////////////////////////
                // ACCELERATOR'S CONTROLLER //
                //////////////////////////

/*********************************************************************
 Accelerator's state:
*********************************************************************/
`define idle    2'b00
`define loading 2'b01
`define running 2'b10

/*********************************************************************
 Addressing modes defining the right operand:
*********************************************************************/
`define imm 3'b000 // op: immue
`define dir 3'b001 // op: mem[immue]
`define rel 3'b010 // op: mem[addrReg + immue]
`define rei 3'b011 // op: mem[addrReg + immue]; addrReg = addrReg + immue
`define cim 3'b100 // op: coOperand
`define cdr 3'b101 // op: mem[coOperand]
`define crl 3'b110 // op: mem[addrReg + coOpereand]
`define ctl 3'b111 // control

/*********************************************************************
Name: Controller's instruction set architecture

instr = {contrOpcode[4:0], contrMode[2:0], contrimmue[23:0]}

reg      [$clog2(`m)-1:0] cPC
reg [31:0]            cAcc      // controller's accumuulator
reg                   cCR       // controller's carry
reg[$clog2(`m)-1:0]       cAddrReg // controller's address reg.
reg[31:0]             cMem      // controller's data memory
*********************************************************************/
// FUNCTIONAL INSTRUCTIONS contrMode != 111
`define cadd    5'b00000 // {cCR,cAcc} <= cAcc + op
`define caddcr  5'b00001 // {cCR,cAcc} <= cAcc + op + cCR
`define csub    5'b00010 // {cCR,cAcc} <= cAcc - op
`define csubcr  5'b00011 // {cCR,cAcc} <= cAcc - op - cCR
`define cmult   5'b00100 // {cCR,cAcc} <= {cCR, cAcc * op}
`define cbwand  5'b00101 // {cCR,cAcc} <= {cCR, cAcc & op}
```

```
`define cbwor   5'b00110 // {cCR,cAcc} <= {cCR, cAcc | op}
`define cbwxor  5'b00111 // {cCR,cAcc} <= {cCR, cAcc ^ op}
// DATA TRANSFER INSTRUCTIONS
`define cload   5'b01000 // {cCR,cAcc} <= {cCR, op}
`define cinsval 5'b01001 // {cCR,cAcc} <= {cCR, (cAcc[23:0] ^ op[7:0]}
`define redload 5'b01010 // {cCR,cAcc} = reductionOut
`define cstore  5'b01011 // cMem <= cAcc
`define cshift  5'b01100 // shift one position immue coded
// CONTROL INSTRUCTIONS  contrMode == 111
`define nop     5'b00000 // pc <= pc+1
`define jmp     5'b00001 // pc <= pc+immue
`define brz     5'b00010 // pc <= cAcc=0 ? pc+immue : pc+1
`define brnz    5'b00011 // pc <= cAcc=0? pc+1 : pc+immue
`define brzdec  5'b00100 // pc <= cAcc=0 ? pc+immue : pc+1
`define brnzdec 5'b00101 // pc <= cAcc=0 ? pc+1 : pc+immue
`define ajmp    5'b00110 // pc <= immue
`define halt    5'b00111 // pc <= pc
`define setint  5'b01000 // set interrupt
`define datains 5'b01001 // insert in array
`define dataext 5'b01010 // extract from arrray
`define crela   5'b01011 // cAddrReg <= cAcc; load address
`define start   5'b11011 // start cycle counter
`define stop    5'b11100 // stop cycle coounter
`define param   5'b11101 // load parameter
`define pload   5'b11110 // load program starting to immue
`define prun    5'b11111 // run the program at immue


                    //////////////////////
                    // ACCELERATOR'S ARRAY//
                    //////////////////////

/*****************************************************************
Name: Array's instruction set architecture

instr = {arrayOpcode[4:0], arrayMode[2:0], immue[23:0]}

reg [31:0]           acc[0:p-1]      // array's accumuulator
reg                  cr[0:p-1]       // array's carry
reg[$clog2(m)-1:0]   addrReg[0:p-1]  // array's address reg.
reg[n-1:0]           mem[0:m]        // array's data memory
reg[$clog2(p)-1]     act[0:p-1]      // activate counter
reg[n-1:0]           sr[0:p-1]       // serial register
*****************************************************************/
// FUNCTIONAL INSTRUCTIONS contrMode != 111
`define add     5'b00000 // {aCR,aAcc} <=  aAcc + op
`define addcr   5'b00001 // {aCR,aAcc} <=  aAcc + op + cr
`define sub     5'b00010 // {aCR,aAcc} <=  aAcc - op
`define subcr   5'b00011 // {aCR,aAcc} <=  aAcc - op - cr
`define mult    5'b00100 // {aCR,aAcc} <= {aCR, aAcc * op}
```

```
`define bwand    5'b00101 // {aCR,aAcc} <= {aCR, aAcc & op}
`define bwor     5'b00110 // {aCR,aAcc} <= {aCR, aAcc | op}
`define bwxor    5'b00111 // {aCR,aAcc} <= {aCR, aAcc ^ op}
// DATA TRANSFER INSTRUCTIONS
`define load     5'b01000 // {aCR,aAcc} <= {aCR, op}
`define store    5'b01001 // aMem[immue] <= aAcc
`define sendio   5'b01010 // ioReg[i] <= aMem[immue]; ioSet
`define getio    5'b01011 // aMem[immue] <= ioReg[i]; ioRst
`define insval   5'b01100 // {aCR,aAcc} <= {aCR, (aAcc[23:0],immue[7:0]}
`define search   5'b01101 // where !(b[i] & acc[i]=op); act <= act+1
`define csearch  5'b01110 // where !(b[i-1] & acc[i]=op); act <= act+1
`define loadi    5'b01111 // load & left insert redOut in sr

// CONTROL INSTRUCTIONS contrMode == 111
`define allact   5'b10000 // all cells active act[i] <= 0
`define where    5'b10001 // where (b[i] & cond) act <= act+1
`define elsew    5'b10010 // if(act=0)act<=1; if(act=1)act<=0
`define back     5'b10011 // act <= act-1 as positive integer
// GLOBAL INSTRUCTIONS contrMode == 111
`define shift    5'b10100 // shifts according to immue[2:0]
`define selsh    5'b10101 // selection shift
`define insert   5'b10110 // insert in first position in sr
`define delete   5'b10111 // delete in first position in sr
`define ixload   5'b11000 // acc[i] <= i
`define arela    5'b11001 // aAddrReg <= aAcc
`define setred   5'b11010 // set reduce's function
`define getsr    5'b11011 // acc[i] <= sr[i]
`define sendsr   5'b11100 // sr[i] <= acc[i]
`define redins   5'b11101 // left insert in serial register
`define gshift   5'b11110 // global_shifts(immue[2:0])
```

The Instruction Set Architecture (ISA), based on the previously defined micro-architectures, has two components:

- HOST ISA defined in `0_hostCodeGenerator.sv` (see below) is a typical RISC ISA
- ACCELERATOR ISA with its two components:
  - CONTROLLER ISA defined in `0_accCodeGenerator.sv` (see below) is a typical RISC ISA
  - ARRAY ISA defined in `0_accCodeGenerator.sv` (see below) is a an atypical ISA with instructions specific for an array of execution units designed as an accumulator-based engine working with a large two-port register file.

C.2.2 HOST's Instruction Set Architecture

Code Generator for Host: `0_hostCodeGenerator.sv`

This file is included in `0_simulator.sv` Verilog file to generate the binary form of the program `0_hProgram.sv` executed by HOST.

```
/*********************************************************************
File name: 0_hostCodeGenerator.sv

                    CODE GENERATOR FOR HOST

*********************************************************************/
    reg [5:0]   opCode          ;
    reg [4:0]   dest            ;
    reg [4:0]   left            ;
    reg [4:0]   right           ;
    reg [15:0]  value           ;
    reg [9:0]   addrCounter     ;
    reg [9:0]   labelTab[0:1023];

    `include "0_DEFINES.vh"

    task endLine;
     begin
        dut.host.hostProgramMemory.hPmem[addrCounter][31:0] =
            {   opCode  ,
                dest    ,
                left    ,
                value   }                   ;
            addrCounter = addrCounter + 1   ;
     end
    endtask

    // sets labelTab in the first pass
    // associating 'counter' with 'labelIndex'
    task hLB    ;
        input [5:0] labelIndex;
        labelTab[labelIndex] = addrCounter;
    endtask
    // uses the content of labelTab in the second pass
    task hULB;
        input [5:0] labelIndex;
        value = labelTab[labelIndex] - addrCounter;
    endtask

    `include "cgHOST_LIBRARY.sv"

// CONTROL INSTRUCTIONS
    task hNOP; // increment host program counter: pc <= pc + 1
        begin   opCode  = `hnop ;
                dest    = 5'b0  ;
                left    = 5'b0  ;
                value   = 16'b0 ;
                endLine         ;
        end
    endtask
```

```
task hJMP; // unconditional jump to address labeled with "label"
    input   [15:0]  label   ;
    begin   opCode  = `hjmp ;
            dest    = 5'b0  ;
            left    = 5'b0  ;
            hULB(label)     ;
            endLine         ;
    end
endtask

task hJMPZ; // jump if (rf[regFile] == 0) to label
    input   [4:0]   regfile     ;
    input   [9:0]   label       ;
    begin   opCode  = `hjmpz    ;
            dest    = 5'b0      ;
            left    = regfile   ;
            hULB(label)         ;
            endLine             ;
    end
endtask

task hJMPNZ; // jump if !(rf[regFile] == 0) to label
    input   [4:0]   regfile     ;
    input   [9:0]   label       ;
    begin   opCode  = `hjmpnz   ;
            dest    = 5'b0      ;
            left    = regfile   ;
            hULB(label)         ;
            endLine             ;
    end
endtask

task hPSEND; // program send:
             // pc <= (host2int[31:24] == \{prun,ctl\}) ? pc+1 : pc
             // h2iPwrite = 1
             // rf[dest] <= rf[left]+!i2hPfull
    input   [4:0] d ;
    input   [4:0] l ;
    begin   opCode  = `hpsend;
            dest    = d     ;
            left    = l     ;
            value   = 16'b0 ;
            endLine         ;
    end
endtask

task hDGET; // data get:
            // pc <= (leftOut == rightOut) ? pc+1 : pc
            // i2hDread = 1
```

```
            // rf[dest] <= rf[left] + !i2hDempty
    input   [4:0] d ;
    input   [4:0] l ; // initial address
    input   [4:0] r ; // final address
    begin   opCode  = `hdget    ;
            dest    = d         ;
            left    = l         ;
            value   = {r, 11'b0};
            endLine             ;
    end
endtask

task hDSEND; // data send:
             // pc <= (leftOut == rightOut) ? pc+1 : pc
             // i2hDwrite = 1
             // rf[dest] <= rf[left] + !i2hDfull
    input   [4:0] d ;
    input   [4:0] l ; // initial address
    input   [4:0] r ; // final address
    begin   opCode  = `hdsend   ;
            dest    = d         ;
            left    = l         ;
            value   = {r, 11'b0};
            endLine             ;
    end
endtask

task hINTWAIT; // interrupt wait: pc <= (int) ? pc+1 : pc
    begin   opCode  = `hintwait ;
            dest    = 5'b0      ;
            left    = 5'b0      ;
            value   = 16'b0     ;
            endLine             ;
    end
endtask

task hAJMP; // absolute jump: pc <= scalar
    input   [15:0]  scalar  ;
    begin   opCode  = `hajmp;
            dest    = 5'b0  ;
            left    = 5'b0  ;
            value   = scalar;
            endLine         ;
    end
endtask

task hHALT; // pc <= pc
    begin   opCode  = `hhalt;
            dest    = 5'b0  ;
            left    = 5'b0  ;
```

```
                value   = 16'b0 ;
                endLine         ;
        end
    endtask

    task hSTARTCC; // start host cycle counter
        begin   opCode  = `hsttcc   ;
                dest    = 5'b0      ;
                left    = 5'b0      ;
                value   = 16'b0     ;
                endLine             ;
        end
    endtask

    task hSTOPCC; // stop host cycle counter
        begin   opCode  = `hstpcc   ;
                dest    = 5'b0      ;
                left    = 5'b0      ;
                value   = 16'b0     ;
                endLine             ;
        end
    endtask

// FUNCTIONAL INSTRUCTIONS
    task hADD; // {cr, rf[d]} <= rf[l] + rf[r]
        input   [4:0]   d   ;
        input   [4:0]   l   ;
        input   [4:0]   r   ;
        begin   opCode  = `hadd     ;
                dest    = d         ;
                left    = l         ;
                value   = {r, 11'b0};
                endLine             ;
        end
    endtask

    task hADDCR; // {cr, rf[d]} <= rf[l] + rf[r] + cr
        input   [4:0]   d   ;
        input   [4:0]   l   ;
        input   [4:0]   r   ;
        begin   opCode  = `haddcr   ;
                dest    = d         ;
                left    = l         ;
                value   = {r, 11'b0};
                endLine             ;
        end
    endtask

    task hSUB; // {cr, rf[d]} <= rf[l] - rf[r]
        input   [4:0]   d   ;
```

```
        input   [4:0]   l   ;
        input   [4:0]   r   ;
        begin   opCode  = `hsub     ;
                dest    = d         ;
                left    = l         ;
                value   = {r, 11'b0};
                endLine             ;
        end
    endtask

    task hSUBCR; // {cr, rf[d]} <= rf[l] - rf[r] - cr
        input   [4:0]   d   ;
        input   [4:0]   l   ;
        input   [4:0]   r   ;
        begin   opCode  = `hsubcr   ;
                dest    = d         ;
                left    = l         ;
                value   = {r, 11'b0};
                endLine             ;
        end
    endtask

    task hMULT; // rf[d] <= rf[l] * rf[r]
        input   [4:0]   d   ;
        input   [4:0]   l   ;
        input   [4:0]   r   ;
        begin   opCode  = `hmult    ;
                dest    = d         ;
                left    = l         ;
                value   = {r, 11'b0};
                endLine             ;
        end
    endtask

    task hAND; // rf[d] <= rf[l] & rf[r]
        input   [4:0]   d   ;
        input   [4:0]   l   ;
        input   [4:0]   r   ;
        begin   opCode  = `hbwand   ;
                dest    = d         ;
                left    = l         ;
                value   = {r, 11'b0};
                endLine             ;
        end
    endtask

    task hOR; // rf[d] <= rf[l] | rf[r]
        input   [4:0]   d   ;
        input   [4:0]   l   ;
        input   [4:0]   r   ;
```

```
        begin   opCode  = `hbwor    ;
                dest    = d         ;
                left    = l         ;
                value   = {r, 11'b0};
                endLine             ;
        end
    endtask

    task hXOR; // rf[d] <= rf[l] ^ rf[r]
        input   [4:0]   d   ;
        input   [4:0]   l   ;
        input   [4:0]   r   ;
        begin   opCode  = `hbwxor   ;
                dest    = d         ;
                left    = l         ;
                value   = {r, 11'b0};
                endLine             ;
        end
    endtask

    task hADDV; // {cr, rf[d]} <= rf[l] + hVal
        input   [4:0]   d       ;
        input   [4:0]   l       ;
        input   [15:0]  scalar  ;
        begin   opCode  = `haddv;
                dest    = d     ;
                left    = l     ;
                value   = scalar;
                endLine         ;
        end
    endtask

    task hADDCRV; {cr, rf[d]} <= rf[l] + hVal + cr
        input   [4:0]   d       ;
        input   [4:0]   l       ;
        input   [15:0]  scalar  ;
        begin   opCode  = `haddcrv  ;
                dest    = d         ;
                left    = l         ;
                value   = scalar    ;
                endLine             ;
        end
    endtask

    task hSUBV; {cr, rf[d]} <= rf[l] - hVal
        input   [4:0]   d       ;
        input   [4:0]   l       ;
        input   [15:0]  scalar  ;
        begin   opCode  = `hsubv;
                dest    = d     ;
```

```
            left    = l     ;
            value   = scalar;
            endLine         ;
    end
endtask

task hSUBCRV; {cr, rf[d]} <= rf[l] - hVal - cr
    input   [4:0]   d       ;
    input   [4:0]   l       ;
    input   [15:0]  scalar  ;
    begin   opCode  = `hsubcrv  ;
            dest    = d         ;
            left    = l         ;
            value   = scalar    ;
            endLine             ;
    end
endtask

task hMULTV; rf[d] <= rf[l] * hVal
    input   [4:0]   d       ;
    input   [4:0]   l       ;
    input   [15:0]  scalar  ;
    begin   opCode  = `hmultv   ;
            dest    = d         ;
            left    = l         ;
            value   = scalar    ;
            endLine             ;
    end
endtask

task hANDV; rf[d] <= rf[l] & {{16{hVal[15]}},hVal}
    input   [4:0]   d       ;
    input   [4:0]   l       ;
    input   [15:0]  scalar  ;
    begin   opCode  = `hbwandv  ;
            dest    = d         ;
            left    = l         ;
            value   = scalar    ;
            endLine             ;
    end
endtask

task hORV; rf[d] <= rf[l] | {{16{hVal[15]}},hVal}
    input   [4:0]   d       ;
    input   [4:0]   l       ;
    input   [15:0]  scalar  ;
    begin   opCode  = `hbworv   ;
            dest    = d         ;
            left    = l         ;
            value   = scalar    ;
```

```
                endLine             ;
        end
    endtask

    task hXORV; rf[d] <= rf[l] ^ {{16{hVal[15]}},hVal}
        input   [4:0]   d       ;
        input   [4:0]   l       ;
        input   [15:0]  scalar  ;
        begin   opCode  = `hbwxorv  ;
                dest    = d         ;
                left    = l         ;
                value   = scalar    ;
                endLine             ;
        end
    endtask

// DATA TRANSFER INSTRUCTIONS
    task hVALUE; rf[d] <= {{16{hVal[15]}},hVal}
        input   [4:0]   d       ;
        input   [15:0]  scalar  ;
        begin   opCode  = `hvalue   ;
                dest    = d         ;
                left    = 5'b0      ;
                value   = scalar    ;
                endLine             ;
        end
    endtask

    task hINSVAL; rf[d] <= {rf[left][15], hVal}
        input   [4:0]   d       ;
        input
        input   [15:0]  scalar  ;
        begin   opCode  = `hinsval  ;
                dest    = d         ;
                left    = 5'b0      ;
                value   = scalar    ;
                endLine             ;
        end
    endtask

    task hLOAD; rf[d] <= hDataMem[rf[l]]
        input   [4:0]   d       ;
        input   [4:0]   l       ;
        begin   opCode  = `hload;
                dest    = d     ;
                left    = l     ;
                value   = 16'b0 ;
                endLine         ;
        end
    endtask
```

```
task hSTORE; // hDataMem[rf[l]] <= rf[r]
    input   [4:0]   l       ;
    input   [4:0]   r       ;
    begin   opCode  = `hstore   ;
            dest    = 5'b0      ;
            left    = l         ;
            value   = {r, 11'b0};
            endLine             ;
    end
endtask

// RUNNING
initial begin   addrCounter = 0;
                `include "0_hProgram.sv" // first pass
                addrCounter = 0;
                `include "0_hProgram.sv" // second pass
        end
```

C.2.3 ACCELERATOR's Instruction Set Architecture

This file is included in `0_simulator.sv` Verilog file to generate the binary form of the two-column assembly program `0_aProgram.sv` executed by ACCELERATOR. A synthetic form is presented in the next 4 tables, and the explicit form for each instructions follows after.

The first column and the first line in Table 6.1 contains components of the micro-architecture.

TRANSFER :

program run starting from `cVal` :

- `cPRUN: instruction to self-delimit the program loaded by HOST in CONTOLLER's program memory`

program load starting from `cVal` :

- `cPLOAD: instruction to start in CONTROLLER the loading of the program sent by HOST`

parameter load :

- `cPARAM: acc <= progFIFOout`

pop from inDataFIFO and insert in ARRAY :

- `cDATAINS`

extract from ARRAY and push in outDataFIFO :

- `cDATAEXT`

interrupt set to HOST :

- `cSETINT`

input-output register store :

- `GETIO: mem[i][aVal] <= ioReg[i]`

input-output register relative store :

- `RGETIO: mem[i][addrr[i] + aVal] <= ioReg[i]`

input-output register relative store & address increment :

- `RIGETIO: mem[i][addr[i] + aVal] <= ioReg[i]`
  `addr[i] <= addr[i] + aVal`

input-output register absolute load :

- `SENDIO: ioReg[i] <= mem[i][aVal]`

input-output register relative load & address update :

- `SENDIO: ioReg[i] <= mem[i][addr[i] + aVal] addr[i] <= addr[i] + aVal`

CONTROL

flow control :

- `cHALT: pc <= pc`
- `cNOP: pc <= pc + 1`
- `cJMP: pc <= pc + cVal`
- `cAJMP: pc <= cVal`
- `cBRNZ: pc <= (acc == 0) ?  pc + 1 :  pc + cVal`
- `cBRZ: pc <= (acc == 0) pc + cVal :  pc + 1`
- `cBRNZDECpc <= (acc == 0) ?  pc + 1 :  pc + cVal; acc <= acc - 1`
- `cBRZDEC: pc <= (acc == 0) pc + cVal :  pc + 1; acc <= acc - 1`

spatial control in array :

- `ACTIVATE: b[i] <= 1`
- `WHEREZERO: b[i] <= (b[i] && acc[i] == 0) ?  1 :  0`
- `WHERECARRY: b[i] <= (b[i] && cy[i]) ?  1 :  0`
- `WHERENEGATIVE: b[i] <= (b[i] && acc[i][n-1]) ?  1 :  0`
- `WHEREPREVACT: b[i] <= b[i] ?  b[i-1] :  0`
- `WHEREFIRST: b[i] <= (first) ?  1 :  0`
- `WHERENEXT: b[i] <= (next) ?  1 :  0`

- `WHEREPREV b[i] <= (next || first) ? 1 : 0`
- `WHERENZERO: b[i] <= !(b[i] && acc[i] == 0) ? 1 : 0`
- `WHERENCARRY: b[i] <= !(b[i] && cy[i]) ? 1 : 0`
- `WHERENNEGATIVE: b[i] <= !(b[i] && acc[i][n-1]) ? 1 : 0`
- `WHERENPREVACT: b[i] <= !(i == 0) ? 0 : b[i-1]`
- `WHERENFIRST: b[i] <= !(first) ? 1 : 0`
- `WHERENNEXT: b[i] <= !(next) ? 1 : 0`
- `WHERENPREV b[i] <= !(next || first) ? 1 : 0`
- `ELSEWHERE: activates the most recently deactivated cell`
- `ENDWHERE: cancels the effect of the last "where" selection`
- `SELSHIFT: b[i] <= b[i-1]`

LOAD

immediate load :

- `cVLOAD: acc <= cVal`
- `VLOAD: acc[i] <= aVal`

insert value :

- `cINSVAL: acc <= {acc[23:0], cVal[7:0]}`
- `INSVAL: acc[i] <= {acc[i][23:0], aVal[7:0]}`

absolute load :

- `cLOAD: acc <= mem[cVal]`
- `LOAD: acc[i] <= mem[i][aVal]`

relative load :

- `cRLOAD: acc <= mem[addr + cVal]`
- `RLOAD: acc[i] <= mem[i][addr[i] + aVal]`

relative load & increment :

- `cRILOAD: acc <= mem[addr + cVal]`
  `addr <= addr + cVal`
- `RILOAD: acc[i] <= mem[i][addr[i] + aVal]`
  `addr[i] <= addr[i] + aVal`

co-operand immediate load :

- `cCLOAD: acc <= redOut`
- `CLOAD: acc[i] <= acc`

co-operand absolute load :

- `CDLOAD: acc[i] <= mem[i][acc]`

co-operand relative load :

- `CDLOAD: acc[i] <= mem[i][addr[i] + acc]`

index load :

- `IXLOAD: acc[i] <= i`

serial register load :

- `GETSR: acc[i] <= serialReg[i]`

input-output register load :

- `GETIO: mem[i][aVal] <= ioReg[i]`

STORE

relative address store :

- `cADDRLD: addr <= acc`
- `ADDRLD: addr[i] <= acc[i]`

absolute store :

- `cSTORE: mem[cVal] <= acc`
- `STORE: mem[i][aVal] <= acc[i]`

relative store :

- `cRSTORE: mem[addr + cVal] <= acc`
- `RSTORE: mem[i][addr[i] + aVal] <= acc[i]`

relative store & address update :

- `cRISTORE: mem[addr + cVal] <= acc`
  `addr <= addr + cVal`
- `RISTORE: mem[i][addr[i] + aVal] <= acc[i]`
  `addr[i] <= addr[i] + aVal`

co-operand absolute store :

- `CSTORE: mem[i][acc] <= acc[i]`

co-operand relative store :

- `CSTORE: mem[i][addr[i] + acc] <= acc[i]`

ADDITION :

immediate addition :

- `cVADD: {cy, acc} <= acc + cVal`
- `VADD: {cy[i], acc[i]} <= acc[i] + aVal`

absolute addition :

- `cADD: {cy, acc} <= acc + mem[cVal]`
- `ADD: {cy[i], acc[i]} <= acc[i] + mem[i][aVal]`

relative addition :

- `cRADD: {cy, acc} <= acc + mem[addr + cVal]`
- `RADD: {cy[i], acc[i]} <= acc[i] + mem[i][addr[i] + aVal]`

relative addition & increment :

- `cRIADD: {cy, acc} <= acc + mem[addr + cVal]`
  `addr <= addr + cVal`
- `RIADD: {cy[i], acc[i]} <= acc[i] + mem[i][addr[i] + aVal]`
  `addr[i] <= addr[i] + aVal`

immediate addition with co-operand :

- `cCADD: acc{cy, acc} <= acc + redOut`
- `CADD: {cy[i], acc[i]} <= acc[i] + acc`

absolute addition with co-operand :

- `CAADD: {cy[i], acc[i]} <= acc[i] + mem[i][acc]`

relative addition with co-operand :

- `CRADD: {cy[i], acc[i]} <= acc[i] + mem[i][addr[i] + acc]`

ADDITION WITH CARRY :

immediate addition with carry :

- `cVADDC: {cy, acc} <= acc + cVal + cy`
- `VADDC: {cy[i], acc[i]} <= acc[i] + aVal + cy[i]`

absolute addition with carry :

- `cADDC: {cy, acc} <= acc + mem[cVal] + cy`
- `ADDC: {cy[i], acc[i]} <= acc[i] + mem[i][aVal] + cy[i]`

relative addition with carry :

- `cRADDC {cy, acc} <= acc + mem[addr + cVal] + cy`
- `RADDC: {cy[i], acc[i]} <= acc[i] + mem[i][addr[i] + aVal] + cy[i]`

relative addition with carry & increment :

- `cRIADDC: {cy, acc} <= acc + mem[addr + cVal + cy]`
  `addr <= addr + cVal`
- `RIADDC: {cy[i], acc[i]} <= acc[i] + mem[i][addr[i] + aVal + cy[i]]`
  `addr[i] <= addr[i] + aVal`

immediate addition with carry with co-operand :

- `cCADDC: {cy, acc} <= acc + redOut + cy`
- `CADDC: {cy[i], acc[i]} <= acc[i] + acc + cy[i]`

absolute addition with carry with co-operand :

- `CAADDC: {cy[i], acc[i]} <= acc[i] + mem[i][acc] + cy[i]`

relative addition with carry with co-operand :

- `CRADDC: {cy[i], acc[i]} <= acc[i] + mem[i][addr[i] + acc + cy[i]`

SUBTRACT :

immediate subtract :

- `cVSUB: {cy, acc} <= acc - cVal`
- `VSUB: {cy[i], acc[i]} <= acc[i] - aVal`

absolute subtract :

- `cSUB: {cy, acc} <= acc - mem[cVal]`
- `SUB: {cy[i], acc[i]} <= acc[i] - mem[i][aVal]`

relative subtract :

- `cRSUB: {cy, acc} <= acc + mem[addr - cVal]`
- `RSUB: {cy[i], acc[i]} <= acc[i] - mem[i][addr[i] + aVal]`

relative subtract & increment :

- `cRISUB: {cy, acc} <= acc - mem[addr + cVal]`
  `addr <= addr + cVal`
- `RISUB: {cy[i], acc[i]} <= acc[i] - mem[i][addr[i] + aVal]`
  `addr[i] <= addr[i] + aVal`

immediate subtract with co-operand :

- `cCSUB: acc{cy, acc} <= acc - redOut`

- CSUB: {cy[i], acc[i]} <= acc[i] - acc

absolute subtract with co-operand :

- CASUB: {cy[i], acc[i]} <= acc[i] - mem[i][acc]

relative subtract with co-operand :

- CRSUB: {cy[i], acc[i]} <= acc[i] - mem[i][addr[i] + acc]

SUBTRACT WITH CARRY :

immediate subtract with carry :

- cVSUBC: {cy, acc} <= acc - cVal - cy
- VSUBC: {cy[i], acc[i]} <= acc[i] - aVal - cy[i]

absolute subtract with carry :

- cSUBC: {cy, acc} <= acc - mem[cVal] - cy
- SUBC: {cy[i], acc[i]} <= acc[i] - mem[i][aVal] - cy[i]

relative subtract with carry :

- cRSUBC: {cy, acc} <= acc - mem[addr + cVal] - cy
- RSUBC: {cy[i], acc[i]} <= acc[i] - mem[i][addr[i] + aVal] - cy[i]

relative subtract with carry & increment :

- cRISUBC: {cy, acc} <= acc + mem[addr - cVal] - cy
  addr <= addr + cVal
- RISUBC: {cy[i], acc[i]} <= acc[i] - mem[i][addr[i] + aVal] - cy[i]
  addr[i] <= addr[i] + aVal

immediate subtract with carry with co-operand :

- cCSUBC: acc{cy, acc} <= acc - redOut - cy
- CSUBC: {cy[i], acc[i]} <= acc[i] - acc - cy[i]

absolute subtract with carry with co-operand :

- CASUBC: {cy[i], acc[i]} <= acc[i] - mem[i][acc] - cy[i]

relative subtract with carry with co-operand :

- CRSUBC: {cy[i], acc[i]} <= acc[i] - mem[i][addr[i] + acc] - cy[i]

MULTIPLY :

immediate multiply :

- `cVMULT: acc <= acc * cVal`
- `VMULT: acc[i] <= acc[i] * aVal`

absolute multiply :

- `cMULT: acc <= acc * mem[cVal]`
- `MULT: acc[i] <= acc[i] * mem[i][aVal]`

relative multiply :

- `cRMULT: acc <= acc * mem[addr + cVal]`
- `RMULT: acc[i] <= acc[i] * mem[i][addr[i] + aVal]`

relative multiply & increment :

- `cRIMULT: acc <= acc * mem[addr + cVal]`
  `addr <= addr + cVal`
- `RIMULT: acc[i] <= acc[i] * mem[i][addr[i] + aVal]`
  `addr[i] <= addr[i] + aVal`

immediate multiply with co-operand :

- `cCMULT: acc <= acc * redOut`
- `CMULT: acc[i] <= acc[i] * acc`

absolute multiply with co-operand :

- `CAMULT: acc[i] <= acc[i] * mem[i][acc]`

relative multiply with co-operand :

- `CRMULT: acc[i] <= acc[i] * mem[i][addr[i] + acc]`

SHIFT/ROTATE

right shift one bit position :

- `cSHR: acc <= {1'b0, acc[n-1:1]}`
  `cy <= acc[0]`
- `SHR: acc[i] <= {1'b0, acc[i][n-1:1]}`
  `cy[i] <= acc[i][0]`

arithmetic right shift one bit position :

- `cASHR: acc <= {acc[n-1], acc[n-1:1]}`
  `cy <= acc[0]`

- ASHR: acc[i] <= {acc[i][n-1], acc[i][n-1:1]}
  cy[i] <= acc[i][0]

right shift one bit position with carry :

- cSHRC: acc <= {cy, acc[n-1:1]}
  cy <= acc[0]
- SHRC: acc[i] <= {cy[i], acc[i][n-1:1]}
  cy[i] <= acc[i][0]

left shift one bit position :

- cSHL: acc <= {acc[n-2:0], 1'b0} cy <= acc[n-1]
- SHL: acc[i] <= {acc[i][n-2:0], 1'b0} cy[i] <= acc[i][n-1]

left shift one bit position with carry :

- cSHLC: acc <= {acc[n-2:0], cy} cy <= acc[n-1]
- SHLC: acc[i] <= {acc[i][n-2:0], cy[i]} cy[i] <= acc[i][n-1]

right rotate one bit position :

- cROTR: acc <= {acc[0], acc[n-1:1]}
- ROTR: acc[i] <= {acc[i][0], acc[i][n-1:1]}

left rotate one bit position :

- cROTL: acc <= {acc[n-2:0], acc[n-1]}
- ROTL: acc[i] <= {acc[i][n-2:0], acc[i][n-1]}

AND :

immediate bitwise AND :

- cVAND: acc <= acc & cVal
- VAND: acc[i] <= acc[i] & aVal

absolute bitwise AND :

- cAND: acc <= acc & mem[cVal]
- AND: acc[i] <= acc[i] & mem[i][aVal]

relative bitwise AND :

- cRAND: acc <= acc & mem[addr + cVal]
- RAND: acc[i] <= acc[i] & mem[i][addr[i] + aVal]

relative bitwise AND & increment :

- `cRIADND: acc <= acc & mem[addr + cVal]`
  `addr <= addr + cVal`
- `RIAND: acc[i] <= acc[i] & mem[i][addr[i] + aVal]`
  `addr[i] <= addr[i] + aVal`

immediate bitwise AND with co-operand :

- `cCAND: acc <= acc & redOut`
- `CAND: acc[i] <= acc[i] & acc`

absolute bitwise AND with co-operand :

- `CAAND: acc[i] <= acc[i] & mem[i][acc]`

relative bitwise AND with co-operand :

- `CRAND: acc[i] <= acc[i] & mem[i][addr[i] + acc]`

OR :

immediate bitwise OR :

- `cVOR: acc <= acc | cVal`
- `VOR: acc[i] <= acc[i] | aVal`

absolute bitwise OR :

- `cOR: acc <= acc | mem[cVal]`
- `OR: acc[i] <= acc[i] | mem[i][aVal]`

relative bitwise OR :

- `cROR: acc <= acc | mem[addr + cVal]`
- `ROR: acc[i] <= acc[i]  mem[i][addr[i] + aVal]`

relative bitwise OR & increment :

- `cRIOR: acc <= acc | mem[addr + cVal]`
  `addr <= addr + cVal`
- `RIADDOR: acc[i] <= acc[i] | mem[i][addr[i] + aVal]`
  `addr[i] <= addr[i] + aVal`

immediate bitwise OR with co-operand :

- `cCOR: acc <= acc | redOut`
- `COR: acc[i] <= acc[i] | acc`

absolute bitwise OR with co-operand :

- `CAOR: acc[i] <= acc[i] | mem[i][acc]`

relative bitwise OR with co-operand :

- `CROR: acc[i] <= acc[i] | mem[i][addr[i] + acc]`

XOR :

immediate bitwise XOR :

- `cVXOR: acc <= acc ⊕ cVal`
- `VXOR: acc[i] <= acc[i] ⊕ aVal`

absolute bitwise XOR :

- `cXOR: acc <= acc ⊕ mem[cVal]`
- `XOR: acc[i] <= acc[i] ⊕ mem[i][aVal]`

relative bitwise XOR :

- `cRXOR: acc <= acc ⊕ mem[addr + cVal]`
- `RXOR: acc[i] <= acc[i] ⊕ mem[i][addr[i] + aVal]`

relative bitwise XOR & increment :

- `cRIXOR: acc <= acc ⊕ mem[addr + cVal]`
  `addr <= addr + cVal`
- `RIXOR: acc[i] <= acc[i] ⊕ mem[i][addr[i] + aVal]`
  `addr[i] <= addr[i] + aVal`

immediate bitwise XOR with co-operand :

- `cCXOR: acc <= acc ⊕ redOut`
- `CXOR: acc[i] <= acc[i] ⊕ acc`

absolute bitwise XOR with co-operand :

- `CAXOR: acc[i] <= acc[i] ⊕ mem[i][acc]`

relative bitwise XOR with co-operand :

- `CRXOR: acc[i] <= acc[i] ⊕ mem[i][addr[i] + acc]`

REDUCE :

reduction add :

- `REDADD: acc <= acc[0] + acc[1] + ... + acc[p-1]`

reduction OP :

- `REDOR: acc <= acc[0] | acc[1] | ... | acc[p-1]`

reduction maximum :

- `REDMAX: acc <= max(acc[0], acc[1], ..., acc[p-1])`

reduction minimum :

- `REDMIN: acc <= min(acc[0], acc[1], ..., acc[p-1])`

SEARCH :

search co-operand :

- `SEARCH: b[i] <= (b[i] && (acc[i] == acc)) ? 1 : 0`

search value :

- `VSEARCH: b[i] <= (b[i] && (acc[i] == aVal)) ? 1 : 0`

conditioned search co-operand :

- `CSEARCH: b[i] <= (b[i-1] && (acc[i] == acc)) ? 1 : 0`

conditioned search value :

- `VCSEARCH: b[i] <= (b[i-1] && (acc[i] == aVal)) ? 1 : 0`

g

GLOBAL :

global right shift with one position :

- `GRSHIFT: serialReg[i] <= (i==0) ? 0 : serialReg[i-1]`

global left shift with one position :

- `GLSHIFT: serialReg[i] <= (i==p-1) ? 0 : aerialReg[i+1]`

insert left redOut in shift register :

- `LREDINS: serialReg[i] <= (i==0) ? redOut : serialReg[i-1]`

insert right redOut in shift register :

- `RREDINS: serialReg[i] <= (i==p-1) ? redOut : aerialReg[i+1]`

global right rotate with one position :

- `GRROTATE: serialReg[i] <= (i==0) ? serialReg[p-1] : serialReg[i-1]`

global left rotate with one position :

- `GLROTATE: serialReg[i] <= (i==p-1) ? aerialReg[0] : aerialReg[i+1]`

send to serial register :

- `SENDSR: serialReg[i] <= acc[i]`

insert value in the first active position :

- `INSERT: serialReg[i] <= (first) ? aVal : (next) ? serialReg[i-1] : serialReg[i]`

insert co-operand in the first active position :

- `CINSERT: serialReg[i] <= (first) ? acc : ((next) ? serialReg[i-1] : serialReg[i])`

delete the first active accumulator :

- `DELETE: (first || next) ? serialReg[i+1] : ((i==p-1) ? 0 : serialReg)`

C.2.4 ACCELERATOR's Code Generator: `0_accCodeGenerator.sv`

```
/*************************************************************
File name: 0_accCodeGenerator.sv

                CODE GENERATOR FOR ACCELERATOR

*************************************************************/

    reg [4:0]    aOpCode                    ;
    reg [2:0]    aOperand                   ;
    reg [23:0]   aScalar                    ;
    reg [4:0]    cOpCode                    ;
    reg [2:0]    cOperand                   ;
    reg [23:0]   cScalar                    ;
    reg [$clog2(`m)-1:0] addrCount          ;
    reg [$clog2(`m)-1:0] labTab[0:`m-1]     ;

    `include "0_DEFINES.vh"

    task endLineA;
        begin
            dut.host.hostDataMemory.hDmem[addrCount] =
            //dut.accelerator.controller.contrProgMem.
            //contrPM[addrCount] =          // without host
            {   aOpCode ,
                aOperand,
                aScalar ,
                cOpCode ,
                cOperand,
                cScalar }                   ;
            addrCount   = addrCount + 1 ;
        end
    endtask

    // sets labelTab in the first pass
    task LB ;
        input [5:0] labIndex;
        labTab[labIndex] = addrCount;
    endtask
```

```
// uses the content of labelTab in the second pass
task cULB;
    input [5:0] labIndex;
    cScalar = labTab[labIndex] - addrCount;
endtask

task NOP;
    begin   aOpCode     = `add  ;
            aOperand    = `val  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask

task cNOP;
    begin   cOpCode     = `cadd ;
            cOperand    = `val  ;
            cScalar     = 0     ;
    end
endtask

`include "cgCONTROL.sv"     // control instructions
`include "cgLOAD.sv"        // load accumulator
`include "cgSTORE.sv"       // store accumulator
`include "cgADD.sv"         // addition
`include "cgADDC.sv"        // addition with carry
`include "cgSUB.sv"         // subtract
`include "cgSUBC.sv"        // subtract with carry
`include "cgMULT.sv"        // multiplication
`include "cgSHIFT.sv"       // shift
`include "cgAND.sv"         // bit-wise AND
`include "cgOR.sv"          // bit-wise OR
`include "cgXOR.sv"         // bit-wise XOR
`include "cgREDUCE.sv"      // reduce network functions
`include "cgGLOBAL.sv"      // global operations
`include "cgTRANSFER.sv"    // io transfer operations
`include "cgSEARCH.sv"      // search functions
//`include "lowLevelLibrary.sv"

task cSTART;
    begin   cOpCode     = `start;
            cOperand    = `ctl  ;
            cScalar     = 0     ;
    end
endtask

task cSTOP;
    begin   cOpCode     = `stop ;
            cOperand    = `ctl  ;
            cScalar     = 0     ;
```

```
        end
    endtask

    // RUNNING
    initial begin   addrCount   = 0;
                    `include "0_aProgram.sv" // first pass
                    addrCount   = 0;
                    `include "0_aProgram.sv" // second pass
            end
```

Control Instructions: `cgCONTROL.sv` is a non Verilog file.

```
/*************************************************************
File name: cgCONTROL.sv

                    CONTROL INSTRUCTIONS

*************************************************************/
// in CONTROLLER
    task cHALT;
        begin   cOpCode     = `halt ;
                cOperand    = `ctl  ;
                cScalar     = 0     ;
        end
    endtask

    task cJMP;
        input [9:0] lab ;
        begin   cOpCode     = `jmp  ;
                cOperand    = `ctl  ;
                cULB(lab)           ;
        end
    endtask

    task cAJMP;
        input [9:0] value   ;
        begin   cOpCode     = `ajmp ;
                cOperand    = `ctl  ;
                cScalar     = value ;
        end
    endtask

    task cBRNZ;
        input [9:0] lab ;
        begin   cOpCode     = `brnz ;
                cOperand    = `ctl  ;
                cULB(lab)           ;
```

```
        end
    endtask

    task cBRZ;
        input [9:0] lab ;
        begin   cOpCode     = `brz   ;
                cOperand    = `ctl   ;
                cULB(lab)            ;
        end
    endtask

    task cBRNZDEC;
        input [9:0] lab ;
        begin   cOpCode     = `brnzdec  ;
                cOperand    = `ctl      ;
                cULB(lab)               ;
        end
    endtask

    task cBRZDEC;
        input [9:0] lab ;
        begin   cOpCode     = `brzdec   ;
                cOperand    = `ctl      ;
                cULB(lab)               ;
        end
    endtask

//in ARRAY
    task ACTIVATE;
        begin   aOpCode     = `allact   ;
                aOperand    = `ctl      ;
                aScalar     = 0         ;
                endLineA                ;
        end
    endtask

    task WHEREZERO; // where acc[i] = 0
        begin   aOpCode     = `where    ;
                aOperand    = `ctl      ;
                aScalar     = 24'b0000  ;
                endLineA                ;
        end
    endtask

    task WHERECARRY; // where carry
        begin   aOpCode     = `where    ;
                aOperand    = `ctl      ;
                aScalar     = 24'b0001  ;
                endLineA                ;
        end
```

```
endtask

task WHERENEGATIVE; // where negative
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b0010  ;
            endLineA                ;
    end
endtask

task WHEREPREVACT; // where previous is active
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b0011  ;
            endLineA                ;
    end
endtask

task WHEREFIRST; // where first
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b0100  ;
            endLineA                ;
    end
endtask

task WHERENEXT; // where next to first
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b0101  ;
            endLineA                ;
    end
endtask

task WHEREPREV; // where previous to first
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b0110  ;
            endLineA                ;
    end
endtask

task WHERENZERO; // where acc[i] != 0
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b1000  ;
            endLineA                ;
    end
endtask
```

```
task WHERENCARRY; // where not carry
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b1001  ;
            endLineA                ;
    end
endtask

task WHEREPOSITIVE; // where positive
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b1010  ;
            endLineA                ;
    end
endtask

task WHERENPREVACT; // where previous is active
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b1011  ;
            endLineA                ;
    end
endtask

task WHERENFIRST; // where not first
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b1100  ;
            endLineA                ;
    end
endtask

task WHERENNEXT; // where not next to first
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b1101  ;
            endLineA                ;
    end
endtask

task WHERENPREV; // where not previous to first
    begin   aOpCode     = `where    ;
            aOperand    = `ctl      ;
            aScalar     = 24'b1110  ;
            endLineA                ;
    end
endtask

task ELSEWHERE; // else where
    begin   aOpCode     = `elsew;
```

```
                aOperand    = `ctl   ;
                aScalar     = 0      ;
                endLineA             ;
        end
    endtask

    task ENDWHERE;  // end where
        begin   aOpCode     = `back  ;
                aOperand    = `ctl   ;
                aScalar     = 0      ;
                endLineA             ;
        end
    endtask
```

Load Instructions: `cgLOAD.sv` is a non Verilog file.

```
/*************************************************************
File name: cgLOAD.sv

                    LOAD INSTRUCTIONS

*************************************************************/
// in CONTROLLER
    task cVLOAD;
        input [23:0] value;

        begin   cOpCode     = `cload;
                cOperand    = `imm   ;
                cScalar     = value  ;
        end
    endtask

    task cINSVAL;
        input [23:0] value;

        begin   cOpCode     = `cinsval  ;
                cOperand    = `ctl      ;
                cScalar     = value     ;
        end
    endtask

    task cLOAD;
        input [23:0] value;

        begin   cOpCode     = `cload;
                cOperand    = `dir   ;
                cScalar     = value  ;
```

```
        end
    endtask

    task cRLOAD;
        input [23:0] value;

        begin   cOpCode     = `cload;
                cOperand    = `rel  ;
                cScalar     = value ;
        end
    endtask

    task cRILOAD;
        input [23:0] value;

        begin   cOpCode     = `cload;
                cOperand    = `rei  ;
                cScalar     = value ;
        end
    endtask

    task cCLOAD;
        begin   cOpCode     = `cload;
                cOperand    = `cim  ;
                cScalar     = 0     ;
        end
    endtask

//in ARRAY

    task IXLOAD;     // // acc[i] <= i
        begin   aOpCode     = `ixload   ;
                aOperand    = `ctl      ;
                aScalar     = 0         ;
                endLineA                ;
        end
    endtask

    task GETSR; // acc[i] <= serialReg[i]
        begin   aOpCode     = `getsr;
                aOperand    = `ctl  ;
                aScalar     = 0     ;
                endLineA            ;
        end
    endtask

    task VLOAD;
        input [23:0] value;

        begin   aOpCode     = `load;
```

```
            aOperand    = `imm  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task INSVAL;
    input [23:0] value;

    begin   aOpCode     = `insval   ;
            aOperand    = `ctl      ;
            aScalar     = value     ;
            endLineA                ;
    end
endtask

task LOAD;
    input [23:0] value;

    begin   aOpCode     = `load;
            aOperand    = `dir  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task RLOAD;
    input [23:0] value;

    begin   aOpCode     = `load;
            aOperand    = `rel  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task RILOAD;
    input [23:0] value;

    begin   aOpCode     = `load;
            aOperand    = `rei  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task RILOADI;
    input [23:0] value;

    begin   aOpCode     = `loadi;
```

```
                aOperand    = `rei   ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task CLOAD;
        begin   aOpCode     = `load;
                aOperand    = `cim   ;
                aScalar     = 0     ;
                endLineA            ;
        end
    endtask

    task CDLOAD;
        begin   aOpCode     = `load;
                aOperand    = `cdr   ;
                aScalar     = 0     ;
                endLineA            ;
        end
    endtask

    task CRLOAD;

        begin   aOpCode     = `load;
                aOperand    = `crl   ;
                aScalar     = 0     ;
                endLineA            ;
        end
    endtask
```

Store Instructions: `cgSTORE.sv` is a non Verilog file.

```
/*********************************************************
File name: cgSTORE.sv

                    STORE INSTRUCTIONS

*********************************************************/
// in CONTROLLER
    task cADDRLD; // addr <= acc
        begin   cOpCode     = `crela     ;
                cOperand    = `val       ;
                cScalar     = 0          ;
        end
    endtask
```

```
    task cSTORE; // store acc at contrScalar
        input [9:0] value;
        begin   cOpCode     = `cstore   ;
                cOperand    = `imm      ;
                cScalar     = value     ;
        end
    endtask

    task cRSTORE; // store acc at addr + contrScalar
        input [9:0] value;
        begin   cOpCode     = `cstore   ;
                cOperand    = `rel      ;
                cScalar     = value     ;
        end
    endtask

    task cRISTORE; // store acc at addr + contrScalar
        input [9:0] value;
        begin   cOpCode     = `cstore   ;
                cOperand    = `rei      ;
                cScalar     = value     ;
        end
    endtask

// in ARRAY
    task STORE;     // store acc[i] at arrayScalar
        input [23:0] value;
        begin   aOpCode     = `store    ;
                aOperand    = `imm      ;
                aScalar     = value     ;
                endLineA                ;
        end
    endtask

    task RSTORE;    // store acc[i] at addr[i] + arrayScalar
        input [23:0] value;
        begin   aOpCode     = `store    ;
                aOperand    = `rel      ;
                aScalar     = value     ;
                endLineA                ;
        end
    endtask

    task RISTORE;   // store acc[i] at addr[i] + arrayScalar
                    // addr[i] <= addr[i] + contrScalar
        input [23:0] value;
        begin   aOpCode     = `store    ;
                aOperand    = `rei      ;
                aScalar     = value     ;
                endLineA                ;
```

```
    end
endtask

task CSTORE;     // store acc[i] at  acc
    begin   aOpCode     = `store    ;
            aOperand    = `cim      ;
            aScalar     = 0         ;
            endLineA                ;
    end
endtask

task CRSTORE;    // store acc[i] at addr[i] + acc
    begin   aOpCode     = `store    ;
            aOperand    = `crl      ;
            aScalar     = 0         ;
            endLineA                ;
    end
endtask

task ADDRLD;     // addr[i] <= acc[i]
    begin   aOpCode     = `arela;
            aOperand    = `ctl  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask

task SENDIO;
    input [23:0] value;
    begin   aOpCode     = `sendio   ;
            aOperand    = `imm      ;
            aScalar     = value     ;
            endLineA                ;
    end
endtask

task RISENDIO;
    input [23:0] value;
    begin   aOpCode     = `sendio   ;
            aOperand    = `rei      ;
            aScalar     = value     ;
            endLineA                ;
    end
endtask
```

Add Instructions: `cgADD.sv` is a non Verilog file.

```
/*************************************************************
File name: cgADD.sv

                    ADDITION INSTRUCTIONS

*************************************************************/
// in CONTROLLER
    task cVADD; // immediate addition:
                // acc <= acc + cScalar
        input [23:0] value;
        begin   cOpCode     = `cadd ;
                cOperand    = `val  ;
                cScalar     = value ;
        end
    endtask

    task cADD; // immediate addressing addition:
               // acc <= acc + mem[cScalar[s-1:0]]
        input [9:0] value;
        begin   cOpCode     = `cadd ;
                cOperand    = `imm  ;
                cScalar     = value ;
        end
    endtask

    task cRADD; // relative addressing addition:
                // acc <= acc + mem[addr + cScalar[s-1:0]]
        input [9:0] value;
        begin   cOpCode     = `cadd ;
                cOperand    = `rel  ;
                cScalar     = value ;
        end
    endtask

    task cRIADD; // relative addressing add; address update
                // acc <= acc + mem[addr + cScalar[s-1:0]]
                // addr = addr + cScalar[s-1:0]
        input [9:0] value;
        begin   cOpCode     = `cadd ;
                cOperand    = `rei  ;
                cScalar     = value ;
        end
    endtask

    task cCADD; // cooperand addressing addition:
                // acc <= acc + coOperand
        input [9:0] value;
        begin   cOpCode     = `cadd ;
                cOperand    = `cop  ;
                cScalar     = value ;
```

```
        end
    endtask

// in ARRAY
    task VADD; // value add:
               // acc[i] <= acc[i] + aScalar
        input [23:0] value;
        begin   aOpCode     = `add  ;
                aOperand    = `val  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task ADD; // absolute add
              // acc[i] <= acc[i] + vectMem[i][aScalar]
        input [23:0] value;
        begin   aOpCode     = `add  ;
                aOperand    = `imm  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task RADD;  // relative
                // add: acc[i] <= acc[i] +
                // vectMem[i][addrVect[i] + aScalar[v-1:0]]
        input [23:0] value;
        begin   aOpCode     = `add  ;
                aOperand    = `rel  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task RIADD; // relative add and increment:
                // acc[i] <= acc[i] + vectMem[i][addrVect[i]
                // + aScalar[v-1:0]]
                // addrVect[i] <= addrVect[i]+aScalar[v-1:0]
        input [23:0] value;
        begin   aOpCode     = `add  ;
                aOperand    = `rei  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task CADD; // co-operand add:
               // acc[i] <= acc[i] + acc
        begin   aOpCode     = `add  ;
```

```
                aOperand    = `cop   ;
                aScalar     = 0      ;
                endLineA             ;
        end
    endtask

    task CAADD; // co-operand absolute add
                // acc[i] <= acc[i] + vectMem[i][acc[v-1:0]]
        begin   aOpCode     = `add   ;
                aOperand    = `cim   ;
                aScalar     = 0      ;
                endLineA             ;
        end
    endtask

    task CRADD; // co-operand relative add:
                // acc[i] <= acc[i] + vectMem[i][addrVect[i]
                // + acc[v-1:0]]
        begin   aOpCode     = `add   ;
                aOperand    = `crl   ;
                aScalar     = 0      ;
                endLineA             ;
        end
    endtask
```

Add with Carry Instructions: `cgADDC.sv` is a non Verilog file.

```
/*************************************************************
File name: cgADDC.sv

                ADDITION WITH CARRY INSTRUCTIONS

*************************************************************/
// in CONTROLLER
    task cVADDC; // immediate addition with carry:
                 // acc <= acc + cScalar
        input [23:0] value;
        begin   cOpCode     = `caddcr    ;
                cOperand    = `val       ;
                cScalar     = value      ;
        end
    endtask

    task cADDC; // immediate addressing addition with carry:
               // acc <= acc + mem[cScalar[s-1:0]]
        input [9:0] value;
        begin   cOpCode     = `caddcr    ;
```

```
                cOperand    = `imm      ;
                cScalar     = value     ;
        end
    endtask

    task cRADDC; // relative addressing addition with carry:
                // acc <= acc + mem[addr + cScalar[s-1:0]]
        input [9:0] value;
        begin   cOpCode     = `caddcr   ;
                cOperand    = `rel      ;
                cScalar     = value     ;
        end
    endtask

    task cRIADDC; // relative addressing addition with carry
                  //with address update:
                  // acc <= acc + mem[addr + cScalar[s-1:0]]
                  // addr = addr + cScalar[s-1:0]
        input [9:0] value;
        begin   cOpCode     = `caddcr   ;
                cOperand    = `rei      ;
                cScalar     = value     ;
        end
    endtask

    task cCADDC; // cooperand addressing addition with carry:
                // acc <= acc + coOperand
        input [9:0] value;
        begin   cOpCode     = `caddcr   ;
                cOperand    = `cop      ;
                cScalar     = value     ;
        end
    endtask

// in ARRAY
    task VADDC; // value add with carry:
               // acc[i] <= acc[i] + aScalar + carry
        input [23:0] value;
        begin   aOpCode     = `addcr;
                aOperand    = `val  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task ADDC; // absolute add with carry
              // acc[i] <= acc[i]+vectMem[i][aScalar]+ carry
        input [23:0] value;
        begin   aOpCode     = `addcr;
                aOperand    = `imm  ;
```

```
            aScalar     = value ;
            endLineA            ;
    end
endtask

task RADDC; // relative add with carry:
           // acc[i] <= acc[i] + vectMem[i][addrVect[i]
           // + aScalar[v-1:0]] + carry
    input [23:0] value;
    begin   aOpCode     = `addcr;
            aOperand    = `rel  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task RIADDC; // relative add with carry and increment:
            // acc[i] <= acc[i] + vectMem[i][addrVect[i]
            // + aScalar[v-1:0]] + carry
            // addrVect[i] <= addrVect[i]+aScalar[v-1:0]
    input [23:0] value;
    begin   aOpCode     = `addcr;
            aOperand    = `rei  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task CADDC; // co-operand add with carry:
           // acc[i] <= acc[i] + acc + carry
    begin   aOpCode     = `addcr;
            aOperand    = `cop  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask

task CAADDC; // co-operand absolute add with carry
             // acc[i] <= acc[i] + vectMem[i][acc[v-1:0]]
             // + carry
    begin   aOpCode     = `addcr;
            aOperand    = `cim  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask

task CRADDC; // co-operand relative add with carry:
             // acc[i] <= acc[i] + vectMem[i][addrVect[i]
             // + acc[v-1:0]] + carry
```

```
        begin   aOpCode     = `addcr;
                aOperand    = `crl  ;
                aScalar     = 0     ;
                endLineA            ;
        end
    endtask
```

Subtract Instructions: `cgSUB.sv` is a non Verilog file.

```
/*************************************************************
File name: cgSUB.sv

                    SUBTRACT INSTRUCTIONS

*************************************************************/
// in CONTROLLER
    task cVSUB; // immediate subtract:
                // acc <= acc - cScalar
        input [23:0] value;
        begin   cOpCode     = `csub ;
                cOperand    = `val  ;
                cScalar     = value ;
        end
    endtask

    task cSUB; // immediate addressing subtract:
               // acc <= acc - mem[cScalar[9:0]]
        input [9:0] value;
        begin   cOpCode     = `csub ;
                cOperand    = `imm  ;
                cScalar     = value ;
        end
    endtask

    task cRSUB; // relative addressing subtract:
                // acc <= acc - mem[addr + cScalar[9:0]]
        input [9:0] value;
        begin   cOpCode     = `csub ;
                cOperand    = `rel  ;
                cScalar     = value ;
        end
    endtask

    task cRISUB; // relative addressing sub; address update:
                 // acc <= acc - mem[addr + cScalar[9:0]]
                 // addr = addr + cScalar[s-1:0]
        input [9:0] value;
```

```
        begin   cOpCode     = `csub ;
                cOperand    = `rei  ;
                cScalar     = value ;
        end
    endtask

    task cCSUB; // cooperand addressing subtract:
                // acc <= acc - coOperand
        input [9:0] value;
        begin   cOpCode     = `csub ;
                cOperand    = `cop  ;
                cScalar     = value ;
        end
    endtask


// in ARRAY
    task VSUB; // value subtract:
               // acc[i] <= acc[i] - aScalar
        input [23:0] value;
        begin   aOpCode     = `sub  ;
                aOperand    = `val  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task SUB; // absolute subtract
              // acc[i] <= acc[i] - vectMem[i][aScalar]
        input [23:0] value;
        begin   aOpCode     = `sub  ;
                aOperand    = `imm  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task RSUB; // relative subtract:
               // acc[i] <= acc[i] - vectMem[i][addrVect[i]
               / + aScalar[v-1:0]]
        input [23:0] value;
        begin   aOpCode     = `sub  ;
                aOperand    = `rel  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task RISUB; // relative subtract and increment:
                // acc[i] <= acc[i] - vectMem[i][addrVect[i]
```

```
                // + aScalar[v-1:0]]
                // addrVect[i] <= addrVect[i]+aScalar[v-1:0]
        input [23:0] value;
        begin   aOpCode     = `sub  ;
                aOperand    = `rei  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task CSUB; // co-operand subtract:
               // acc[i] <= acc[i] - acc
        begin   aOpCode     = `sub  ;
                aOperand    = `cop  ;
                aScalar     = 0     ;
                endLineA            ;
        end
    endtask

    task CASUB; // co-operand absolute subtract
              // acc[i] <= acc[i] - vectMem[i][acc[v-1:0]]
        begin   aOpCode     = `sub  ;
                aOperand    = `cim  ;
                aScalar     = 0     ;
                endLineA            ;
        end
    endtask

    task CRSUB; // co-operand relative subtract:
                // acc[i] <= acc[i] - vectMem[i][addrVect[i]
                // + acc[v-1:0]]
        begin   aOpCode     = `sub  ;
                aOperand    = `crl  ;
                aScalar     = 0     ;
                endLineA            ;
        end
    endtask
```

Subtract with Carry Instructions: `cgSUBC.sv` is a non Verilog file.

```
/*************************************************************
File name: cgSUBC.sv

                SUBTRACT WITH CARRY INSTRUCTIONS

*************************************************************/
// in CONTROLLER
```

```
    task cVSUBC; // immediate subtract with carry:
                 // acc <= acc - cScalar - carry
        input [23:0] value;
        begin   cOpCode     = `csubcr   ;
                cOperand    = `val      ;
                cScalar     = value     ;
        end
    endtask

    task cSUBC; // immediate addressing subtract with carry:
                // acc <= acc - mem[cScalar[9:0]] - carry
        input [9:0] value;
        begin   cOpCode     = `csubcr   ;
                cOperand    = `imm      ;
                cScalar     = value     ;
        end
    endtask

    task cRSUBC; // relative addressing subtract with carry:
                 // acc <= acc-mem[addr+cScalar[9:0]]-carry
        input [9:0] value;
        begin   cOpCode     = `csubcr   ;
                cOperand    = `rel      ;
                cScalar     = value     ;
        end
    endtask

    task cRISUBC; // relative addressing subtract with carry
                  //with address update:
                 // acc <= acc-mem[addr+cScalar[9:0]]-carry
                 // addr = addr + cScalar[s-1:0]
        input [9:0] value;
        begin   cOpCode     = `csubcr   ;
                cOperand    = `rei      ;
                cScalar     = value     ;
        end
    endtask

    task cCSUBC; // cooperand addressing subtract with carry:
                 // acc <= acc - coOperand - carry
        input [9:0] value;
        begin   cOpCode     = `csubcr   ;
                cOperand    = `cop      ;
                cScalar     = value     ;
        end
    endtask

// in ARRAY
    task VSUBC; // value subtract with carry:
                // acc[i] <= acc[i] - aScalar - carry
```

```
    input [23:0] value;
    begin   aOpCode     = `subcr;
            aOperand    = `val  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task SUBC; // absolute subtract with carry
          // acc[i] <= acc[i]-vectMem[i][aScalar]-carry
    input [23:0] value;
    begin   aOpCode     = `subcr;
            aOperand    = `imm  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task RSUBC; // relative subtract with carry:
           // acc[i] <= acc[i] - vectMem[i][addrVect[i]
           // + aScalar[v-1:0]] - carry
    input [23:0] value;
    begin   aOpCode     = `subcr;
            aOperand    = `rel  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task RISUBC; // relative subtract with carry & increment:
            // acc[i] <= acc[i] - vectMem[i][addrVect[i]
            // + aScalar[v-1:0]] - carry
            // addrVect[i] <= addrVect[i]+aScalar[v-1:0]
    input [23:0] value;
    begin   aOpCode     = `subcr;
            aOperand    = `rei  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task CSUBC; // co-operand subtract with carry:
           // acc[i] <= acc[i] - acc - carry
    begin   aOpCode     = `subcr;
            aOperand    = `cop  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask
```

```
task CASUBC; // co-operand absolute subtract with carry
             // acc[i] <= acc[i]-vectMem[i][acc[v-1:0]]
             // - carry
    begin    aOpCode     = `subcr;
             aOperand    = `cim  ;
             aScalar     = 0     ;
             endLineA            ;
    end
endtask

task CRSUBC; // co-operand relative subtract with carry:
            // acc[i] <= acc[i] - vectMem[i][addrVect[i]
            // + acc[v-1:0]] - carry
    begin    aOpCode     = `subcr;
             aOperand    = `crl  ;
             aScalar     = 0     ;
             endLineA            ;
    end
endtask
```

Multiply Instructions: `cgMULT.sv` is a non Verilog file.

```
/*************************************************************
File name: cgMULT.sv

                    MULTIPLY INSTRUCTIONS

*************************************************************/
    // in CONTROLLER

    task cVMULT; // immediate multiplication:
                 // acc <= acc * cScalar
        input [23:0] value;
        begin    cOpCode     = `cmult;
                 cOperand    = `val  ;
                 cScalar     = value ;
        end
    endtask

    task cMULT; // immediate addressing multiplication:
                // acc <= acc * mem[cScalar[s-1:0]]
        input [9:0] value;
        begin    cOpCode     = `cmult;
                 cOperand    = `imm  ;
                 cScalar     = value ;
        end
    endtask
```

```
    task cRMULT; // relative addressing multiplication:
                 // acc <= acc * mem[addr + cScalar[s-1:0]]
        input [9:0] value;
        begin   cOpCode     = `cmult;
                cOperand    = `rel  ;
                cScalar     = value ;
        end
    endtask

    task cRIMULT; // relative addressing mult; address update
                  // acc <= acc * mem[addr + cScalar[s-1:0]]
                  // addr = addr + cScalar[s-1:0]
        input [9:0] value;
        begin   cOpCode     = `cmult;
                cOperand    = `rei  ;
                cScalar     = value ;
        end
    endtask

    task cCMULT; // cooperand addressing multiplication:
                 // acc <= acc * coOperand
        input [9:0] value;
        begin   cOpCode     = `cmult;
                cOperand    = `cop  ;
                cScalar     = value ;
        end
    endtask

// in ARRAY
    task VMULT; // value multiplication:
                // acc[i] <= acc[i] * aScalar
        input [23:0] value;
        begin   aOpCode     = `mult ;
                aOperand    = `val  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task MULT; // absolute multiplication
               // acc[i] <= acc[i] * vectMem[i][aScalar]
        input [23:0] value;
        begin   aOpCode     = `mult ;
                aOperand    = `imm  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask
```

```
task RMULT; // relative multiplication:
            // acc[i] <= acc[i] * vectMem[i][addrVect[i]
            //+ aScalar[v-1:0]]
    input [23:0] value;
    begin   aOpCode     = `mult ;
            aOperand    = `rel  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task RIMULT; // relative multiplication and increment:
             // acc[i] <= acc[i] * vectMem[i][addrVect[i]
             // + aScalar[v-1:0]]
             // addrVect[i] <= addrVect[i]+aScalar[v-1:0]
    input [23:0] value;
    begin   aOpCode     = `mult ;
            aOperand    = `rei  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task CMULT; // co-operand multiplication:
            // acc[i] <= acc[i] * acc
    begin   aOpCode     = `mult ;
            aOperand    = `cop  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask

task CAMULT; // co-operand absolute multiplication
           // acc[i] <= acc[i] * vectMem[i][acc[v-1:0]]
    begin   aOpCode     = `mult ;
            aOperand    = `cim  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask

task CRMULT; // co-operand relative multiplication:
            // acc[i] <= acc[i] * vectMem[i][addrVect[i]
            // + acc[v-1:0]]
    begin   aOpCode     = `mult ;
            aOperand    = `crl  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask
```

Shift Instructions: `cgSHIFT.sv` is a non Verilog file.

```
/*************************************************************
File name: cgSHIFT.sv

                    SHIFT INSTRUCTIONS

*************************************************************/
    // in CONTROLLER
    task cSHR; // shift right one bit position
        begin   cOpCode     = `cshift   ;
                cOperand    = `imm      ;
                cScalar     = 24'b100   ;
        end
    endtask

    task cASHR; // shift right arithmetic one bit position
        begin   cOpCode     = `cshift   ;
                cOperand    = `imm      ;
                cScalar     = 24'b101   ;
        end
    endtask

    task cSHRC; // shift right one bit position with carry
        begin   cOpCode     = `cshift   ;
                cOperand    = `imm      ;
                cScalar     = 24'b110   ;
        end
    endtask

    task cSHL; // shift left one bit position
        begin   cOpCode     = `cshift   ;
                cOperand    = `imm      ;
                cScalar     = 24'b001   ;
        end
    endtask

    task cSHLC; // shift left one bit position with carry
        begin   cOpCode     = `cshift   ;
                cOperand    = `imm      ;
                cScalar     = 24'b010   ;
        end
    endtask

    task cROTR; // rotate right one bit position
        begin   cOpCode     = `cshift   ;
                cOperand    = `imm      ;
```

```
            cScalar     = 24'b111   ;
    end
endtask

task cROTL; // rotate right one bit position
    begin   cOpCode     = `cshift   ;
            cOperand    = `imm      ;
            cScalar     = 24'b011   ;
    end
endtask

// in ARRAY
task SHR; // shift right one bit position
    begin   aOpCode     = `shift    ;
            aOperand    = `imm      ;
            aScalar     = 24'b100   ;
            endLineA                ;
    end
endtask

task ASHR; // shift right arithmetic one bit position
    begin   aOpCode     = `shift    ;
            aOperand    = `imm      ;
            aScalar     = 24'b101   ;
            endLineA                ;
    end
endtask

task SHRC; // shift right one bit position with carry
    begin   cOpCode     = `shift    ;
            cOperand    = `imm      ;
            cScalar     = 24'b110   ;
            endLineA                ;
    end
endtask

task SHL; // shift left one bit position
    begin   aOpCode     = `shift    ;
            aOperand    = `imm      ;
            aScalar     = 24'b001   ;
            endLineA                ;
    end
endtask

task SHLC; // shift left one bit position with carry
    begin   aOpCode     = `shift    ;
            aOperand    = `imm      ;
            aScalar     = 24'b010   ;
            endLineA                ;
    end
```

```
    endtask

    task ROTR; // rotate right one bit position
        begin   aOpCode     = `shift    ;
                aOperand    = `imm      ;
                aScalar     = 24'b111   ;
                endLineA                ;
        end
    endtask

    task ROTL; // rotate right one bit position
        begin   cOpCode     = `shift    ;
                cOperand    = `imm      ;
                cScalar     = 24'b011   ;
                endLineA                ;
        end
    endtask
```

AND Instructions: `cgAND.sv` is a non Verilog file.

```
/*************************************************************
File name: cgAND.sv

                    BIT-WISE AND INSTRUCTIONS

*************************************************************/
// in CONTROLLER
    task cVAND; // immediate AND-ing:
                // acc <= acc & cScalar
        input [23:0] value;
        begin   cOpCode     = `cbwand   ;
                cOperand    = `val      ;
                cScalar     = value     ;
        end
    endtask

    task cAND; // immediate addressing AND-ing:
               // acc <= acc & mem[cScalar[s-1:0]]
        input [9:0] value;
        begin   cOpCode     = `cbwand   ;
                cOperand    = `imm      ;
                cScalar     = value     ;
        end
    endtask

    task cRAND; // relative addressing AND-ing:
                // acc <= acc & mem[addr + cScalar[s-1:0]]
```

```
        input [9:0] value;
        begin   cOpCode     = `cbwand   ;
                cOperand    = `rel      ;
                cScalar     = value     ;
        end
    endtask

    task cRIAND; // relative addressing AND; address update:
                 // acc <= acc & mem[addr + cScalar[s-1:0]]
                 // addr = addr + cScalar[s-1:0]
        input [9:0] value;
        begin   cOpCode     = `cbwand   ;
                cOperand    = `rei      ;
                cScalar     = value     ;
        end
    endtask

    task cCAND; // cooperand addressing AND-ing:
                // acc <= acc & coOperand
        input [9:0] value;
        begin   cOpCode     = `cbwand   ;
                cOperand    = `cop      ;
                cScalar     = value     ;
        end
    endtask

// in ARRAY
    task VAND; // value AND-ing:
               // acc[i] <= acc[i] & aScalar
        input [23:0] value;
        begin   aOpCode     = `bwand;
                aOperand    = `val  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task AND; // absolute AND-ing
              // acc[i] <= acc[i] & vectMem[i][aScalar]
        input [23:0] value;
        begin   aOpCode     = `bwand;
                aOperand    = `imm  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task RAND; // relative AND-ing:
               // acc[i] <= acc[i] & vectMem[i][addrVect[i]
               // + aScalar[v-1:0]]
```

```
    input [23:0] value;
    begin   aOpCode     = `bwand;
            aOperand    = `rel  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task RIAND; // relative AND-ing and increment:
            // acc[i] <= acc[i] & vectMem[i][addrVect[i]
            // + aScalar[v-1:0]]
            // addrVect[i] <= addrVect[i] + aScalar[v-1:0]
    input [23:0] value;
    begin   aOpCode     = `bwand;
            aOperand    = `rei  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task CAND; // co-operand AND-ing:
           // acc[i] <= acc[i] & acc
    begin   aOpCode     = `bwand;
            aOperand    = `cop  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask

task CAAND; // co-operand absolute AND-ing
          // acc[i] <= acc[i] & vectMem[i][acc[v-1:0]]
    begin   aOpCode     = `bwand;
            aOperand    = `cim  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask

task CRAND; // co-operand relative AND-ing
           // acc[i] <= acc[i] & vectMem[i][addrVect[i]
           // + acc[v-1:0]]
    begin   aOpCode     = `bwand;
            aOperand    = `crl  ;
            aScalar     = 0     ;
            endLineA            ;
    end
endtask
```

OR Instructions: `cgOR.sv` is a non Verilog file.

```
/*************************************************************
File name: cgOR.sv

                    BIT-WISE OR INSTRUCTIONS

*************************************************************/
// in CONTROLLER

    task cVOR; // immediate OR-ing:
               // acc <= acc | cScalar
        input [23:0] value;
        begin   cOpCode     = `cbwor;
                cOperand    = `val  ;
                cScalar     = value ;
        end
    endtask

    task cOR; // immediate addressing OR-ing:
              // acc <= acc | mem[cScalar[s-1:0]]
        input [9:0] value;
        begin   cOpCode     = `cbwor;
                cOperand    = `imm  ;
                cScalar     = value ;
        end
    endtask

    task cROR; // relative addressing OR-ing:
               // acc <= acc | mem[addr + cScalar[s-1:0]]
        input [9:0] value;
        begin   cOpCode     = `cbwor;
                cOperand    = `rel  ;
                cScalar     = value ;
        end
    endtask

    task cRIOR; // relative addressing OR; address update:
                // acc <= acc | mem[addr + cScalar[s-1:0]]
                // addr = addr + cScalar[s-1:0]
        input [9:0] value;
        begin   cOpCode     = `cbwor;
                cOperand    = `rei  ;
                cScalar     = value ;
        end
    endtask

    task cCOR; // cooperand addressing OR-ing:
               // acc <= acc | coOperand
        input [9:0] value;
        begin   cOpCode     = `cbwor;
                cOperand    = `cop  ;
```

```
                cScalar     = value ;
        end
    endtask


// in ARRAY
    task VOR; // value OR-ing:
              // acc[i] <= acc[i] | aScalar
        input [23:0] value;
        begin   aOpCode     = `bwor ;
                aOperand    = `val  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task OR; // absolute OR-ing
             // acc[i] <= acc[i] | vectMem[i][aScalar]
        input [23:0] value;
        begin   aOpCode     = `bwor ;
                aOperand    = `imm  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task ROR; // relative OR-ing:
              // acc[i] <= acc[i] | vectMem[i][addrVect[i]
              // + aScalar[v-1:0]]
        input [23:0] value;
        begin   aOpCode     = `bwor ;
                aOperand    = `rel  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task RIOR; // relative OR-ing and increment:
               // acc[i] <= acc[i] | vectMem[i][addrVect[i]
               // + aScalar[v-1:0]]
               // addrVect[i] <= addrVect[i]+aScalar[v-1:0]
        input [23:0] value;
        begin   aOpCode     = `bwor ;
                aOperand    = `rei  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task COR; // co-operand OR-ing:
```

```
                // acc[i] <= acc[i] | acc
        begin    aOpCode      = `bwor ;
                 aOperand     = `cop  ;
                 aScalar      = 0     ;
                 endLineA             ;
        end
    endtask

    task CAOR; // co-operand absolute OR-ing
               // acc[i] <= acc[i] | vectMem[i][acc[v-1:0]]
        begin    aOpCode      = `bwor ;
                 aOperand     = `cim  ;
                 aScalar      = 0     ;
                 endLineA             ;
        end
    endtask

    task CROR; // co-operand relative OR-ing:
               // acc[i] <= acc[i] | vectMem[i][addrVect[i]
               // + acc[v-1:0]]
        begin    aOpCode      = `bwor ;
                 aOperand     = `crl  ;
                 aScalar      = 0     ;
                 endLineA             ;
        end
    endtask
```

XOR Instructions: `cgXOR.sv` is a non Verilog file.

```
/*************************************************************
File name: cgXOR.sv

                    BIT-WISE XOR INSTRUCTIONS

*************************************************************/
// in CONTROLLER
    task cVXOR; // immediate XOR-ing:
                // acc <= acc ^ cScalar
        input [23:0] value;
        begin    cOpCode      = `cbwxor   ;
                 cOperand     = `val      ;
                 cScalar      = value     ;
        end
    endtask

    task cXOR; // immediate addressing XOR-ing:
               // acc <= acc ^ mem[cScalar[s-1:0]]
```

```
        input [9:0] value;
        begin   cOpCode     = `cbwxor   ;
                cOperand    = `imm      ;
                cScalar     = value     ;
        end
    endtask

    task cRXOR; // relative addressing XOR-ing:
                // acc <= acc ^ mem[addr + cScalar[s-1:0]]
        input [9:0] value;
        begin   cOpCode     = `cbwxor   ;
                cOperand    = `rel      ;
                cScalar     = value     ;
        end
    endtask

    task cRIXOR; // relative addressing XOR; address update:
                // acc <= acc ^ mem[addr + cScalar[s-1:0]]
                // addr = addr + cScalar[s-1:0]
        input [9:0] value;
        begin   cOpCode     = `cbwxor   ;
                cOperand    = `rei      ;
                cScalar     = value     ;
        end
    endtask

    task cCXOR; // cooperand addressing XOR-ing:
                // acc <= acc ^ coOperandadditi
        input [9:0] value;
        begin   cOpCode     = `cbwxor   ;
                cOperand    = `cop      ;
                cScalar     = value     ;
        end
    endtask

// in ARRAY
    task VXOR; // value XOR-ing:
               // acc[i] <= acc[i] ^ aScalar
        input [23:0] value  ;
        begin   aOpCode     = `bwxor;
                aOperand    = `val  ;
                aScalar     = value ;
                endLineA            ;
        end
    endtask

    task XOR; // absolute XOR-ing
              // acc[i] <= acc[i] ^ vectMem[i][aScalar]
        input [23:0] value;
        begin   aOpCode     = `bwxor;
```

```
            aOperand    = `imm  ;
            aScalar     = value ;
            endLineA            ;
    end
endtask

task RXOR; // relative XOR-ing:
           // acc[i] <= acc[i] ^ vectMem[i][addrVect[i]
           // + aScalar[v-1:0]]
    input [23:0] value;
    begin  aOpCode     = `bwxor;
           aOperand    = `rel  ;
           aScalar     = value ;
           endLineA            ;
    end
endtask

task RIXOR; // relative XOR-ing and increment:
            // acc[i] <= acc[i] ^ vectMem[i][addrVect[i]
            // + aScalar[v-1:0]]
            // addrVect[i] <= addrVect[i]+aScalar[v-1:0]
    input [23:0] value;
    begin  aOpCode     = `bwxor;
           aOperand    = `rei  ;
           aScalar     = value ;
           endLineA            ;
    end
endtask

task CXOR; // co-operand XOR-ing:
           // acc[i] <= acc[i] ^ acc
    begin  aOpCode     = `bwxor;
           aOperand    = `cop  ;
           aScalar     = 0     ;
           endLineA            ;
    end
endtask

task CAXOR; // co-operand absolute XOR-ing
          // acc[i] <= acc[i] ^ vectMem[i][acc[v-1:0]]
    begin  aOpCode     = `bwxor;
           aOperand    = `cim  ;
           aScalar     = 0     ;
           endLineA            ;
    end
endtask

task CRXOR; // co-operand relative XOR-ing:
           // acc[i] <= acc[i] ^ vectMem[i][addrVect[i]
           // + acc[v-1:0]]
```

```
        begin   aOpCode     = `bwxor;
                aOperand    = `crl   ;
                aScalar     = 0      ;
                endLineA             ;
        end
    endtask
```

Reduce Instructions: `cgREDUCE.sv` is a non Verilog file.

```
/*************************************************************
File name: cgREDUCE.sv

                    REDUCE INSTRUCTIONS

*************************************************************/
// in REDUCE NETWORK
    task REDADD;
        begin   aOpCode     = `setred    ;
                aOperand    = `ctl       ;
                aScalar     = 24'b00     ;
                endLineA                 ;
        end
    endtask

    task REDOR;
        begin   aOpCode     = `setred    ;
                aOperand    = `ctl       ;
                aScalar     = 24'b01     ;
                endLineA                 ;
        end
    endtask

    task REDMAX;
        begin   aOpCode     = `setred    ;
                aOperand    = `ctl       ;
                aScalar     = 24'b10     ;
                endLineA                 ;
        end
    endtask

    task REDMIN;
        begin   aOpCode     = `setred    ;
                aOperand    = `ctl       ;
                aScalar     = 24'b11     ;
                endLineA                 ;
        end
    endtask
```

Global Instructions: `cgGLOBAL.sv` is a non Verilog file.

```
/*************************************************************
File name: cgGLOBAL.sv

                    GLOBAL INSTRUCTIONS

*************************************************************/
    task GRSHIFT; // global right shift with one position
        begin   aOpCode     = `grshift  ;
                aOperand    = `ctl      ;
                aScalar     = 0         ;
                endLineA                ;
        end
    endtask

    task GLSHIFT; // global left shift with one position
        begin   aOpCode     = `glshift  ;
                aOperand    = `ctl      ;
                aScalar     = 0         ;
                endLineA                ;
        end
    endtask

    task SENDSR;     // serialReg[i] <= acc[i]
        begin   aOpCode     = `sendsr   ;
                aOperand    = `ctl      ;
                aScalar     = 0         ;
                endLineA                ;
        end
    endtask

    task REDINS; // insert reduction output in shift register
        begin   aOpCode     = `redins   ;
                aOperand    = `ctl      ;
                aScalar     = 0         ;
                endLineA                ;
        end
    endtask

    task INSERT; // insert value at first
        input [23:0] value;
        begin   aOpCode     = `insert   ;
                aOperand    = `ctl      ;
                aScalar     = value     ;
                endLineA                ;
        end
    endtask

    task  CINSERT; // insert co-operand at first
```

```
        begin   aOpCode     = `insert   ;
                aOperand    = `cop      ;
                aScalar     = 0         ;
                endLineA                ;
        end
    endtask

    task DELETE; // delete the first active accumulator
        begin   aOpCode     = `delete   ;
                aOperand    = `ctl      ;
                aScalar     = 0         ;
                endLineA                ;
        end
    endtasK
```

Transfer Instructions: `cgTRANSFER.sv` is a non Verilog file.

```
/*************************************************************
File name: cgTRANSFER.sv

                    TRANSFER INSTRUCTIONS

*************************************************************/
// in CONTROLLER
    task cPRUN; // program run from value
        input [23:0] value;
        begin   cOpCode     = `prun ;
                cOperand    = `ctl  ;
                cScalar     = value ;
        end
    endtask

    task cPLOAD; // program load starting from value
        input [23:0] value;
        begin   cOpCode     = `pload;
                cOperand    = `ctl  ;
                cScalar     = value ;
        end
    endtask

    task cPARAM; // parameter load
        begin   cOpCode     = `param;
                cOperand    = `ctl  ;
                cScalar     = 0     ;
        end
    endtask
```

```
    task cDATAINS; // pop from inFIFO
        input [23:0] value;
        begin   cOpCode     = `datains  ;
                cOperand    = `ctl      ;
                cScalar     = value     ;
        end
    endtask

    task cDATAEXT; // push in outFIFO
        input [23:0] value;
        begin   cOpCode     = `dataext  ;
                cOperand    = `ctl      ;
                cScalar     = value     ;
        end
    endtask

    task cSETINT; // set interrupt
        begin   cOpCode     = `setint   ;
                cOperand    = `ctl      ;
                cScalar     = 0         ;
        end
    endtask
```

Search Instructions: `cgSEARCH.sv` is a non Verilog file.

```
/*************************************************************
File name: cgSEARCH.sv

                    SEARCH INSTRUCTIONS

*************************************************************/
    task SEARCH; // search co-operand
        begin   aOpCode     = `search   ;
                aOperand    = `cop      ;
                aScalar     = 0         ;
                endLineA                ;
        end
    endtask

    task VSEARCH; // search value
        input [23:0] value;
        begin   aOpCode     = `search   ;
                aOperand    = `val      ;
                aScalar     = value     ;
                endLineA                ;
        end
    endtask
```

```
task CSEARCH; // search co-operand
    begin   aOpCode     = `csearch  ;
            aOperand    = `cop      ;
            aScalar     = 0         ;
            endLineA                ;
    end
endtask

task VCSEARCH; // search value
    input [23:0] value;
    begin   aOpCode     = `csearch  ;
            aOperand    = `val      ;
            aScalar     = value     ;
            endLineA                ;
    end
endtask
```

C.2.5 Assembly Language

Programming or heterogeneous system in assembly language means to launch two programs:

- `0_hProgram.sv`, whose form is exemplified for the generation and addition of matrices:

```
        hVALUE(0,0); // selects the program generated in the data memory
                     // of HOST
        hPSEND(0,0); // send the self delimited program

    // GEN MATRICES X & UNIT MATRIX AND ADD THEM
        hSQGENX;                // to array: generate matrix X
        hUNIT(`p);              // to array: generate matrix UNIT
        hSQMADD(2*`p, 0, `p);   // to array: add matrices
    // END ADDING PROGRAM

        hHALT;
```

  allowing to load in the program memory of CONTROLLER `0_aProgram.sv`

- `0_aProgram.sv`, is the program designed to run on CONTROLLER whose typical form is:

```
                cPLOAD(0);      ACTIVATE;
                cNOP;           GETIO(1);
                cNOP;           IXLOAD;

                `include "00_theKernel.sv"

        LB(32); cHALT;          NOP;
                cPRUN(0);       NOP;
```

including the kernel library.

Host Assembly Language

```
/*************************************************************************
File name: 0_hProgram.sv
                    HOST PROGRAM
*************************************************************************/
    hVALUE(0,0);      //
    hPSEND(0,0);
// GEN MATRICES X & UNIT MATRIX MULTIPLY ADD THEM AND MAC
/*
    //hSTART;
    hSQGENX;                    // to array: generate matrix X at 0
    hDIAG(`p,1);                // to array: generate diagonal matrix at p
    hSQMMULT(2*`p,2*`p-1,0);    // to array: multiply matrices
    hSQMMAC(2*`p,2*`p-1,0);     // to array: mult & acc matrices
    //hSTOP;
//*/
// GEN MATRICES DIAG 23 AT 0 & DIAG 3 AT 16 MULTIPLY THEM AND MAC
//*
    //hSTART;
    hDIAG(0,23);                    // to array: diagonal matrix of 23 at 0
    hDIAG(`p,3);                    // to array: diagonal matrix of 23 at 16
    hSTART;
    hSQMMULT(2*`p,2*`p-1,0);        // to array: multiply matrices
    hSQMMAC(2*`p,2*`p-1,0);         // to array: mult & acc matrices
    hSTOP;  // p=16: 1748, 6.82c/s; p=32: 5604, 5.47c/s
            // without matrices genereation: p=16: 1618, 6.32c/s
//*/
// GEN MATRICES X & UNIT MATRIX AND ADD THEM
/*
    hSQGENX;                   // to array: generate matrix X
    hUNIT(`p);                 // to array: generate matrix UNIT
    hSQMADD(2*`p, 0, `p);      // to array: add matrices
//*/
//  GEN MATRICES X & UNIT MATRIX AND MULTIPLY THEM
/*
    hSQGENX;                        // to array: generate matrix X
    hUNIT(`p);                      // to array: generate matrix UNIT
    hSTART;
    hSQMMULT(2*`p,2*`p-1,0);        // to array: multiply matrices
    hSTOP; // p=32: 2676; 2.61   cycles/scalar
//*/
// GEN MATRICES X & MULTIPLY WITH VECTOR [N, N, ..., N] FOR N=3
/*
    hSQGENX;                        // to array: generate matrix X
    hVGENN(4,`p);                   // to array: generate vector 3...3 at p
    hSTART;
    hSQMVMULT(`p-1, `p, `p+1);  // to array: matrix-vector multiply
```

```
    hSTOP; // p=32: 89; 2.78 cycles/scalar
//*/
// GENEERATE INDEX VECTOR AND VCETOR [N,...,N, ... N] AT ADDRESS
/*
    hVGENX(13);       // to array: generate index vector at 13
    hVGENN(14,23);    // to array: generate vector [... 14 ...] at 23
//*/
// GEN MATRICES, SEND TO HOST, READ BACK IN ARRAY, MULTIPLY,
// SEND RESULT TO HOST, READ BACK THE RESULT
/*
// GENERATE MATRIX WITH INDEX
    hSQGENX;                  // to array: generate matrix X
    hMSEND(0, `p);            // to array: send matrix from 0 of 16 lines
// to array: send matrix
    hVALUE(1,(`p*`p)/2-1);    // to host: end address in host
    hVALUE(0,0);              // to host: start address in host
    hDGET(0,0,1);             // to host: get data in host
// REPEAT FOR N
    hSQGENN;                  // to array: generate matrix N
    hMSEND(0, `p);            // to array: send matrix
    hVALUE(1,(`p*`p)-1);      // to host: end address in host
    hVALUE(0,(`p*`p)/2);      // to host: start address in host
    hDGET(0,0,1);             // to host: get data in host
// LOAD MATRICES FROM HOST
    hSTART;
    hMGET(`p, `p);            // to array: get at 16 matrix of 16 lines
    hVALUE(1,(`p*`p)/2-1);    // to host: end address in host
    hVALUE(0,0);              // to host: start address in host
    hDSEND(0,0,1);            // to host: send data in host

    hMGET(0, `p);             // to array: get at 16 matrix of 16 lines
    hVALUE(1,(`p*`p)-1);      // to host: end address in host
    hVALUE(0,(`p*`p)/2);      // to host: start address in host
    hDSEND(0,0,1);            // to host: send data in host
// MULTIPLICATION
    hSQMMULT(2*`p,2*`p-1,0);     // to array: multiply matrices
// SEND RESULTS
    hMSEND(2*`p, `p);         // to array: send matrix from 0 of 16 lines
// to array: send matrix
    hVALUE(1,(`p*`p)/2-1);    // to host: end address in host
    hVALUE(0,0);              // to host: start address in host
    hDGET(0,0,1);             // to host: get data in host
    hSTOP;   // p=16: 1590 cycles; 6.2 cycles/scalar_of_result
             // p=32: 5014 cycles; 4.89 cycles/scalar of result
// READ BACK THE RESULT
    hMGET(`p, `p);            // to array: get at 16 matrix of 16 lines
    hVALUE(1,(`p*`p)/2-1);    // to host: end address in host
    hVALUE(0,0);              // to host: start address in host
    hDSEND(0,0,1);            // to host: send data in host
//*/
```

```
    hHALT;
```

Accelerator Assembly Language

```
/*************************************************************
File name: 0_aProgram.sv
                    ACCELERATOR PROGRAM
*************************************************************/
            cPLOAD(0);      ACTIVATE;
            cNOP;           GETIO(1);
            cNOP;           IXLOAD;


            //`include "10_examples.v"

            `include "00_theKernel.sv"


    LB(32); cHALT;          NOP;
            cPRUN(0);       NOP;
```

```
/*************************************************************************
EXAMPLES: 10_examples.v
Simple programs used to demonstrate some features of the accelerator
*************************************************************************/

/*************************************************************************
Test program 1:
    - activate reduction add function
    - acc[i] <= i, for i = 0,1,...,15
    - 2x latency steps
    - acc <= acc[0]+acc[1]+...+acc[15]
*************************************************************************/
/*   version 1: 2x NOPs are used to wait for reduction latency
    cVLOAD(13); REDADD;     // activate reduction add
    cNOP;       VSUB(7);
    cNOP;       WHERENCARRY;
    cNOP;       NOP;        // latency step for distibution
    cREDLOAD;   NOP;        // latency step for distibution
    cREDLOAD;   NOP;        // latency step for distibution
    cREDLOAD;   NOP;        // latency step for distibution
    cREDLOAD;   NOP;        // latency step for reduction
    cREDLOAD;   NOP;        // latency step for reduction
    cREDLOAD;   NOP;        // latency step for reduction
    cREDLOAD;   NOP;        // latency step for reduction
```

```
    cREDLOAD;    NOP;          // acc <= max of indexes
//*/

//*      version 2: a 2x cycles loop controls the latency
            cVLOAD(2*$clog2(`p));    REDADD;
    LB(1);  cBRNZDEC(1);             IXLOAD; // latency loop; acc[i] <= i;
            cREDLOAD;                NOP;    // acc <= sum of indexes
//*/

/*************************************************************************
Test program 2:
    - activate reduction add function
    - acc[i] <= i for i = 0,1,...,15
    - memVect[i][4] <= acc[i] for i = 0,1,...,15
    - acc[i] <= acc[i] x vectMem[i][4]
    - acc <= acc[0]+acc[1]+...+acc[15]
    - mem[24] <= acc = innerProduct(index, index)
*************************************************************************/
/*
            cNOP;                    IXLOAD;    // acc[i] <= index
            cNOP;                    STORE(4);  // memVect[i][4] <= acc[i]
            cNOP;                    REDADD;    // activate reduction add
            cVLOAD(2*$clog2(`p)-1);  MULT(4);   // initl latency loop;
                                                // acc[i] <=
                                                // acc[i] * memVect[i][4]
    LB(1);  cBRNZDEC(1);             NOP;       // latency loop
            cREDLOAD;                NOP;       // acc <= redAdd(acc[i])
            cSTORE(2);               NOP;       // mem[2] <= acc
//*/

/*************************************************************************
Test program 3: how many cells are active after two WHERE
    - activate reduction add function
    - acc[i} <= i
    - keep active cells where (acc[i] >= 5)
    - keep active cells where (acc[i] < 15)
    - acc[i] <= 1 only in all active cells
    - reactivate all cells and wait for latency in 8 = 2x cycles
    - acc <= acc[0]+acc[1]+...+acc[15] only for the active cells
*************************************************************************/
/*
            cNOP;                    REDADD;          // activate red. add
            cNOP;                    IXLOAD;          // acc[i] <= index
            cNOP;                    VSUB(5);         // {cr, acc[i]} <=
                                                      // acc[i] - 5
            cNOP;                    WHERENCARRY;     // where cr=1 active
            cNOP;                    VSUB(10);        // {{cr, acc[i]} <=
                                                      // acc[i] - (15 - 5)
            cNOP;                    WHERECARRY;      // where cr=0 active
            cNOP;                    VLOAD(1);
```

```
            cNOP;                          ENDWHERE;        // reactivate where
                                                            // second WHERE acted
            cVLOAD(2*$clog2(`p)-3); ENDWHERE;               // reactivate where
                                                            // first WHERE acted
    LB(1);  cBRNZDEC(1);                   NOP;             // latency loop
            cREDLOAD;                      NOP;             // acc <= number of
                                                            // active cells
//*/

/*************************************************************************
Test program 4:
    - activate reduction add function
    - acc[i} <= i; load index
    - 2x latency steps
    - acc <= acc[0]+acc[1]+...acc[15]
    - acc[i] <= acc[i] + acc
*************************************************************************/
/*
            cVLOAD(2*$clog2(`p)-1); REDADD;   // set loop size;
                                              // activate reduction add
    LB(1);  cBRNZDEC(1);            IXLOAD;   // latency loop; acc[i] <= i
            cREDLOAD;               NOP;      // acc<=acc[0]+...acc[15]
            cNOP;                   CADD;     // acc[i] <= acc[i] + acc
//*/

/*************************************************************************
Test program 5:
    - acc <= 10; initialize the loop counter and acc[i] = i
    - do (acc+1) times
          acc[i] <= acc[i]/2
          acc[i] <= acc[i] + 99
*************************************************************************/
/*
        cVLOAD(10);     NOP;
LB(1);  cNOP;           SHR;        // divide by 2
        cBRNZDEC(1);    VADD(99); // branch if acc=0, acc<=acc-1, add 99
//*/
```

C.3 Libraries of Functions

C.3.1 Low-Level Library

Low-Level Library Definition: cgHOST_LIBRARY.sv

```
/*************************************************************
File name: cgHOST_LIBRARY.sv
```

```
                    LOW LEVEL LIBRARY FUNCTIONS

*************************************************************/
    task hSSEND; // scalar send as parameter
        input   [25:0]  scalar              ;
        begin   opCode  = `hssend           ;
                {dest, left, value} = scalar;
                endLine                     ;
        end
    endtask

    task hSTART; // start controller's cycle counter
        begin   opCode  = `hfsend   ;
                dest    = 5'b0      ;
                left    = 5'b0      ;
                value   = 16'b0100  ;
                endLine             ;
        end
    endtask

    task hSTOP; // stop controller's cycle counter
        begin   opCode  = `hfsend   ;
                dest    = 5'b0      ;
                left    = 5'b0      ;
                value   = 16'b0101  ;
                endLine             ;
        end
    endtask

    task hINTRQ; // interrupt request
        begin   opCode  = `hfsend   ;
                dest    = 5'b0      ;
                left    = 5'b0      ;
                value   = 16'b0110  ;
                endLine             ;
        end
    endtask

    task hSQGENX; // square matrix X starting with vect[0]
        begin   opCode  = `hfsend   ;
                dest    = 5'b0      ;
                left    = 5'b0      ;
                value   = 16'b0111  ;
                endLine             ;
        end
    endtask

    task hSQGENN; // square matrix N starting with vect[0]
        begin   opCode  = `hfsend   ;
                dest    = 5'b0      ;
```

```
            left    = 5'b0      ;
            value   = 16'b1000  ;
            endLine             ;
    end
endtask

task hMSEND; // matrix send
    input   [25:0]  address ;
    input   [25:0]  size    ;
    begin   opCode  = `hfsend   ;
            dest    = 5'b0      ;
            left    = 5'b0      ;
            value   = 16'b1001  ;
            endLine             ;
            opCode  = `hssend               ;
            {dest, left, value} = address   ;
            endLine                         ;
            opCode  = `hssend               ;
            {dest, left, value} = size      ;
            endLine                         ;
    end
endtask

task hMGET; // matrix get
    input   [25:0]  address ;
    input   [25:0]  size    ;
    begin   opCode  = `hfsend   ;
            dest    = 5'b0      ;
            left    = 5'b0      ;
            value   = 16'b1010  ;
            endLine             ;
            opCode  = `hssend               ;
            {dest, left, value} = address   ;
            endLine                         ;
            opCode  = `hssend               ;
            {dest, left, value} = size      ;
            endLine                         ;
    end
endtask

task hUNIT; // square unit matrix starting at address
    input   [25:0]  address ;
    begin   opCode  = `hfsend   ;
            dest    = 5'b0      ;
            left    = 5'b0      ;
            value   = 16'b1011  ;
            endLine             ;
            opCode  = `hssend               ;
            {dest, left, value} = address   ;
            endLine                         ;
```

```
    end
endtask

task hSQMADD; // square matrices add
    input   [25:0]  d   ;
    input   [25:0]  l   ;
    input   [25:0]  r   ;
    begin   opCode  = 'hfsend   ;
            dest    = 5'b0      ;
            left    = 5'b0      ;
            value   = 16'b1100  ;
            endLine             ;
            opCode  = 'hssend       ;
            {dest, left, value} = d ;
            endLine                 ;
            opCode  = 'hssend       ;
            {dest, left, value} = l ;
            endLine                 ;
            opCode  = 'hssend       ;
            {dest, left, value} = r ;
            endLine                 ;
    end
endtask

task hVGENX; // index vector generate at address
    input   [25:0]  address ;
    begin   opCode  = 'hfsend   ;
            dest    = 5'b0      ;
            left    = 5'b0      ;
            value   = 16'b1101  ;
            endLine             ;
            opCode  = 'hssend               ;
            {dest, left, value} = address   ;
            endLine                         ;
    end
endtask

task hVGENN; // integer vector generate at address
    input   [25:0]  address ;
    input   [25:0]  scalar  ;
    begin   opCode  = 'hfsend   ;
            dest    = 5'b0      ;
            left    = 5'b0      ;
            value   = 16'b1110  ;
            endLine             ;
            opCode  = 'hssend               ;
            {dest, left, value} = address   ;
            endLine                         ;
            opCode  = 'hssend               ;
            {dest, left, value} = scalar    ;
```

```
            endLine                         ;
    end
endtask

task hSQMVMULT; // square matrice vector multiply
    input   [25:0]  d   ;
    input   [25:0]  l   ;
    input   [25:0]  r   ;
    begin   opCode  = `hfsend   ;
            dest    = 5'b0      ;
            left    = 5'b0      ;
            value   = 16'b1111  ;
            endLine             ;
            opCode  = `hssend       ;
            {dest, left, value} = d ;
            endLine                 ;
            opCode  = `hssend       ;
            {dest, left, value} = l ;
            endLine                 ;
            opCode  = `hssend       ;
            {dest, left, value} = r ;
            endLine                 ;
    end
endtask

task hSQMMULT; // square matrices multiply
    input   [25:0]  d   ;
    input   [25:0]  l   ;
    input   [25:0]  r   ;
    begin   opCode  = `hfsend   ;
            dest    = 5'b0      ;
            left    = 5'b0      ;
            value   = 16'b10000 ;
            endLine             ;
            opCode  = `hssend       ;
            {dest, left, value} = d ;
            endLine                 ;
            opCode  = `hssend       ;
            {dest, left, value} = l ;
            endLine                 ;
            opCode  = `hssend       ;
            {dest, left, value} = r ;
            endLine                 ;
    end
endtask

task hSQMMAC; // square matrices multiply-accumulate
    input   [25:0]  d   ;
    input   [25:0]  l   ;
    input   [25:0]  r   ;
```

```
        begin   opCode  = `hfsend   ;
                dest    = 5'b0      ;
                left    = 5'b0      ;
                value   = 16'b10001 ;
                endLine             ;
                opCode  = `hssend       ;
                {dest, left, value} = d ;
                endLine                 ;
                opCode  = `hssend       ;
                {dest, left, value} = l ;
                endLine                 ;
                opCode  = `hssend       ;
                {dest, left, value} = r ;
                endLine                 ;
        end
    endtask
```

Low-Level Library Implementation: `00_theKernel.v`

```
/*************************************************************
File name: 00_theKernel.sv
Description: LINEAR ALGEBRA KERNEL LIBRARY FUNCTIONS
*************************************************************/
            cHALT;      NOP;
            cJMP(1);    NOP; // hSTART
            cJMP(2);    NOP; // hSTOP
            cJMP(3);    NOP; // hINTRQ
            cJMP(4);    NOP; // hSQGENX
            cJMP(5);    NOP; // hSQGENN
            cJMP(6);    NOP; // hMSEND(addr,size)
            cJMP(7);    NOP; // hMGET(addr,size)
            cJMP(8);    NOP; // hUNIT(addr)
            cJMP(9);    NOP; // hSQADD(dest,left,right)
            cJMP(10);   NOP; // hVGENX(addr)
            cJMP(11);   NOP; // hVGENN(addr,value)
            cJMP(12);   NOP; // hSQMVMULT(dest,left,right)
            cJMP(13);   NOP; // hSQMMULT(dest,left,right)
            cJMP(14);   NOP; // hSQMMAC(dest,left,right)
            cJMP(15);   NOP; // hDIAG(dest,value)
//********** START COUNTER ***********************************
    LB(1);  cSTART;         VLOAD(22);
            cJMP(32);       NOP;
//********** STOP COUNTER ************************************
    LB(2);  cSTOP;          VLOAD(33);
            cJMP(32);       NOP;
//********** INTERRUPT REQEST ********************************
    LB(3);  cSETINT;        VLOAD(44);
```

```
                cJMP(32);          NOP;
//********** SQUARE MATRIX X GENERATE **********************
     LB(4);     cVLOAD(`p-1);      VLOAD(-1);
                cNOP;              ADDRLD;
                cNOP;              IXLOAD;
     LB(17);    cNOP;              RISTORE(1);
                cBRNZDEC(17);      VADD(1);
                cJMP(32);          NOP;
//********** SQUARE MATRIX N GENERATE **********************
     LB(5);     cVLOAD(`p-1);      VLOAD(-1);
                cNOP;              ADDRLD;
                cNOP;              VLOAD(0);
     LB(18);    cNOP;              RISTORE(1);
                cBRNZDEC(18);      VADD(1);
                cJMP(32);          NOP;
//********** SEND MATRIX ***********************************
     LB(6);     cPARAM;                    NOP;
                cVSUB(1);                  NOP;
                cPARAM;                    CLOAD;
                cNOP;                      ADDRLD;
                cNOP;                      NOP;
                cNOP;                      RISENDIO(1);
     LB(19);    cNOP;                      NOP;
                cBRZDEC(32);               NOP;
                cSTORE(0);                 NOP;      //
cVLOAD($clog2(`p)-4);
                cVLOAD($clog2(`p)-4);      NOP;      //LB(34);     cBRZDEC(34);
     LB(34);    cBRZDEC(34);               NOP;
                cLOAD(0);                  NOP;
                cDATAEXT(`p/2);            NOP;
                cJMP(19);                  RISENDIO(1);
//********** GET MATRIX ************************************
     LB(7);     cPARAM;                    NOP;
                cVSUB(1);                  NOP;
                cPARAM;                    CLOAD;
                cVSUB(1);                  ADDRLD;
                cSTORE(0);                 NOP;
                cVLOAD($clog2(`p));        RGETIO(1);
     LB(28);    cBRNZDEC(28);              NOP;
     LB(20);    cDATAINS(`p/2);            NOP;
                cLOAD(0);                  RIGETIO(1);
                cBRZDEC(32);               NOP;
                cSTORE(0);                 NOP;
                cVLOAD($clog2(`p)-4);      NOP;
     LB(29);    cBRNZDEC(29);              NOP;
                cJMP(20);                  NOP;
//********** UNIT MATRIX GENERATE **************************
     LB(8);     cPARAM;            NOP;
                cVSUB(1);          NOP;
                cVLOAD(`p-1);      CLOAD;
```

```
                cNOP;               ADDRLD;
                cNOP;               IXLOAD;
                cNOP;               WHEREZERO;
                cNOP;               VLOAD(1);   //!!!!
                cNOP;               ELSEWHERE;
                cNOP;               VLOAD(0);
                cNOP;               SENDSR;
                cNOP;               ENDWHERE;
                cNOP;               NOP; // VLOAD(23);
    LB(22);     cNOP;               GETSR;
                cNOP;               RISTORE(1);
                cBRNZDEC(22);       GRSHIFT;
                cJMP(32);           NOP;
//********** ADD SQUARE MATRICES ***************************
    LB(9);      cPARAM;             NOP;
                cSTORE(3);          NOP;      // dest at mem[3]
                cPARAM;             NOP;
                cSTORE(4);          NOP;      // left at mem[4]
                cPARAM;             NOP;
                cSTORE(5);          NOP;      // right at mem[5]
                cSUB(4);            NOP;
                cSTORE(0);          NOP;      // right-left at mem[0]
                cLOAD(3);           NOP;
                cSUB(5);            NOP;
                cSTORE(1);          NOP;      // dest-right at mem[1]
                cLOAD(4);           NOP;
                cSUB(3);            NOP;
                cVADD(1);           NOP;
                cSTORE(2);          NOP;      // left-dest+1 at mem[2]
                cVLOAD('p+1);       NOP;
                cSTORE(6);          NOP;
    LB(23);     cLOAD(4);           NOP;
                cADD(0);            CDLOAD;
                cADD(1);            CAADD;
                cADD(2);            CSTORE;
                cSTORE(4);          NOP;
                cLOAD(6);           NOP;
                cVSUB(1);           NOP;
                cSTORE(6);          NOP;
                cBRNZ(23);          NOP;
                cJMP(32);           NOP;
//*********** INDEX VECTOR GENERATE ************************
    LB(10);     cPARAM;             IXLOAD;
                cJMP(32);           CSTORE;
//*********** ...N... VECTOR GENERATE ************************
    LB(11);     cPARAM;             NOP;
                cPARAM;             CLOAD;
                cJMP(32);           CSTORE;
//*********** MATRIX-VECTOR MULTIPLY ***********************
    LB(12);     cPARAM;                   NOP;
```

```
            cPARAM;                     CLOAD;
            cSTORE(0);                  VADD(1);      // mem[0] = vector addr
            cPARAM;                     ADDRLD;       // matrix end addr + 1
            cSTORE(1);                  REDADD;       // mem[1] = result vector addr
            cVLOAD(`p);                 RILOAD(-1);
    LB(24); cNOP;                       MULT(`p);
            cBRNZDEC(24);               RILOADI(-1);
            cVLOAD($clog2(`p)-4);       NOP;
    LB(30); cNOP;                       LREDINS;
            cBRNZDEC(30);               NOP;
            cLOAD(1);                   GETSR;
            cJMP(32);                   CSTORE;
//********** MULTIPLY SQUARE MATRICES **********************
    LB(13); cPARAM;                     REDADD;
            cVSUB(1);                   NOP;
            cSTORE(3);                  NOP;      // dest => 3
            cPARAM;                     NOP;
            cSTORE(0);                  NOP;      // left => 0
            cPARAM;                     CLOAD;
            cSTORE(2);                  ADDRLD;   // right => 2
            cVLOAD(`p-1);               NOP;
            cSTORE(1);                  NOP;
            cLOAD(2);                   NOP;
            cVADD(1);                   CDLOAD;
            cSTORE(2);                  STORE(3*`p);
            cVLOAD(`p);                 RILOAD(0);
    LB(25); cNOP;                       MULT(3*`p);
            cBRNZDEC(25);               RILOADI(-1);
            cVLOAD($clog2(`p)-4);       NOP;
    LB(31); cNOP;                       LREDINS;
            cBRNZDEC(31);               NOP;
            cLOAD(3);                   NOP;
            cVADD(1);                   NOP;
            cSTORE(3);                  GETSR;
            cLOAD(2);                   CSTORE;   // CSADD;
            cVADD(1);                   CDLOAD;
            cSTORE(2);                  NOP;
            cLOAD(0);                   STORE(3*`p);
            cLOAD(1);                   CLOAD;
            cBRZDEC(32) ;               ADDRLD;
            cSTORE(1);                  NOP;
            cVLOAD(`p);                 RLOAD(0);
            cJMP(25);                   NOP;
//********** MULTIPLY & ACCUMULATE SQUARE MATRICES **********
    LB(14); cPARAM;                     REDADD;
            cVSUB(1);                   NOP;
            cSTORE(3);                  NOP;      // dest => 3
            cPARAM;                     NOP;
            cSTORE(0);                  NOP;      // left => 0
            cPARAM;                     CLOAD;
```

```
                cSTORE(2);                  ADDRLD; // right => 2
                cVLOAD('p-1);               NOP;
                cSTORE(1);                  NOP;
                cLOAD(2);                   NOP;
                cVADD(1);                   CDLOAD;
                cSTORE(2);                  STORE(3*'p);
                cVLOAD('p);                 RILOAD(0);
     LB(26);    cNOP;                       MULT(3*'p);
                cBRNZDEC(26);               RILOADI(-1);
                cVLOAD($clog2('p)-4);       NOP;
     LB(33);    cNOP;                       LREDINS;
                cBRNZDEC(33);               NOP;
                cLOAD(3);                   NOP;      //MULT(3*'p);
                cVADD(1);                   NOP;
                cNOP;                       GETSR;
                cSTORE(3);                  CAADD; // CSTORE;
                cLOAD(2);                   CSTORE;
                cVADD(1);                   CDLOAD;
                cSTORE(2);                  NOP;
                cLOAD(0);                   STORE(3*'p);
                cLOAD(1);                   CLOAD;
                cBRZDEC(32) ;               ADDRLD;
                cSTORE(1);                  NOP;
                cVLOAD('p);                 RLOAD(0);
                cJMP(26);                   NOP;
//********** DIAGONAL MATRIX GENERATE **************************
     LB(15);    cPARAM;             NOP;
                cVSUB(1);           NOP;
                cPARAM;             CLOAD;
                cNOP;               ADDRLD;
                cNOP;               IXLOAD;
                cNOP;               WHEREZERO;
                cNOP;               CLOAD;   //!!!!
                cNOP;               ELSEWHERE;
                cNOP;               VLOAD(0);
                cNOP;               SENDSR;
                cNOP;               ENDWHERE;
                cVLOAD('p-1);       NOP;
     LB(27);    cNOP;               GETSR;
                cNOP;               RISTORE(1);
                cBRNZDEC(27);       GRSHIFT;
                cJMP(32);           NOP;
```

C.3.2 High-Level Libraries

C.3.3 Performance Evaluation

Bibliography

[1] K. Asanovic, *et al.* (2006) *The Landscape of Parallel Computing Research: A View from Berkeley*, Electrical Engineering and Computer Sciences University of California at Berkeley Technical Report No. UCB/EECS-2006-183, `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`

[2] K. Asanovic, *et al.* (2009) *A View of the Parallel Computing Landscape*, Communications of the ACM, **52**(10):56–67.

[3] John Backus (1978) Can programming be liberated from the von neumann style? a functional style and its algebra of programs, *Communications of the ACM*, **21**(8):613–641, August 1978.

[4] Niloy Banerjee (2019) A Versal ACAP is Significantly Different Than a Regular FPGA or SoC, *BISinfotech*, January 11.

[5] Gerrit Anne Blaauw and Fred P. Brooks, Jr. (1964) The structure of SYSTEM/360, Part I: Outline of the logical structure, *IBM Systems Journal* **3**(2):119–135.

[6] M. Bourne Inverse of a Matrix using Gauss-Jordan Elimination, *Interactive Mathematics*, `https://www.intmath.com/matrices-determinants/inverse-matrix-gauss-jordan-elimination.php`

[7] Gregory Chaitin (1966) On the Length of Programs for Computing Binary Sequences, *Journal of the Association for Computing Machinery*, **13**(4):547–569.

[8] Gregory Chaitin (1970) On the Difficulty of Computation, *IEEE Trans. of Information Theory*, **16**(1):5–9.

[9] Noam Chomsky (1956) Three Models for the Description of Language, *IEEE Trans. on Information Theory*, **2**(3):113–124.

[10] Noam Chomsky (1959) On Certain Formal Properties of Grammars, *Information and Control*, **2**(2):137–167.

[11] Noam Chomsky (1963) Formal Properties of Grammars, D. Luce (ed.) *Handbook of Mathematical Psychology*, Wiley, New-York, 1963.

[12] Alonzo Church (1936) An unsolvable problem of elementary number theory. *The American Journal of Mathematics* **58**(2):345–363.

[13] Jack B. Copeland, ed. (2006) *Colossus: The Secrets of Bletchley Park's Codebreaking Computers* Oxford University Press.

[14] Marc Peter Deisenroth, A. Aldo Faisal, Cheng Soon Ong (2020) *Mathematics for Machine Learning*, Cambridge University Press.

[15] Edsger W. Dijkstra (1965) Co-operating sequential processes. *Programming Languages* Academic Press, New York, pp. 43—112. Reprinted from: Technical Report EWD-123, Technological University, Eindhoven, the Netherlands.

[16] Mihai Drăagăanescu (1983) *Spre o teorie generală a informației*, (Towards a general theory of information) Preprint, Bucuresti, ICI.

[17] Mihai Drăagăanescu (1984) Information, Heuristics, Creation, in I. Plander (Ed.), *Artificial Intelligence and Information, Control Systems of Robots*, Elsevier (North-Holland), pp. 25-29.

[18] Steven Fortune, James C. Wyllie (1978) Parallelism in random access machines, *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 114–118.

[19] Simson L. Garfinkel, Rachel H. Grunspan (2018) *The Computer Book*, Sterling.

[20] Leslie M. Goldschlager (1982) A universal interconnection pattern for parallel computers, *Journal of the ACM* **29**(4):1073–1086.

[21] David Hilbert (1902) Mathematical problems, *Bull. Amer. Math. Soc*, **8**(10):437–479.

[22] David Hilbert, Wilhelm Ackermann (1928) *Grundzüge der theoretischen Logik* (Principles of Mathematical Logic), Springer-Verlag.

[23] Walter Isaacson (2014) *The Innovators: How a Group of Hackers, Geniuses, and Geeks Created the Digital Revolution*, Simon & Schuster.

[24] Aurélien Géron (2017) *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, O'Reilly.

[25] Berndt Klauer (2013) The Convey Hybride-Core Architecture, in Wim Vanderbauwhede, Khaled Benkrid, (eds.) *High-Performance Computing Using FPGAs*, Springer.

[26] Stephen Kleene (1936) General recursive functions of natural numbers, *Mathematische Annalen* **112**(5):727–742.

[27] A. N. Kolmogorov (1965) Three approaches to the definition of the quantity of information, *Problems of Information Transmission*, **1**(1):3-11.

[28] Yann LeCun, et al. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, **1**(4):541–551.

[29] Yann LeCun, et al. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**(11):2278–2324.

[30] Gottfried Wilhelm Leibniz ()1703 *Explication de l'arithmetique binaire*, `chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.apmep.fr/IMG/pdf/Huyghens.pdf`, or `chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://kastalia.medienhaus.udk-berlin.de/odl/Leibniz.pdf`

[31] André Leroi-Gourhan (1964–65) *Le geste et la parole*, 2 vols. Albin Michel. Romanian translation: *Gesstul și cuvântuul*, Ed. Meridiane, 1983.

[32] M. McCool, A. D. Robison, J. Reinders (2012) Structured Parallel Programming. Patterns for Efficient Computation, *Elsevier*.

[33] G. Moore (1964) The Future of Integrated Electronics. *Fairchild Semiconductor internal publication*.

[34] G. Moore 1975 Progress in Digital Integrated Electronics, *Technical Digest 1975. International Electron Devices Meeting, IEEE*, pp. 11-13.

[35] Witold Pedrycz, Shyi-Ming Chen (ed.) (2020) *Deep Learning: Concepts and Architectures*, Springer.

[36] E. Post (1936) Finite combinatory processes. Formulation I, *The Journal of Symbolic Logic*, **1**(3):103–105.

[37] Mihaela Maliţa, George Vladuţ Popescu, Gheorghe M. Ştefan (2019) Heterogenous Computing System for Deep Learning, in Witold Pedrycz, Shyi-Chen (Eds.) *Deep Learning: Concepts and Architectures*, Springer International Publishing, pp 287–319.

[38] Warren S. McCulloch, Walter H. Pitts (1943) A Logical Calculus of the Ideas Immansnt in Nervous Activity, *Bulletin of Mathematical Biiophysics*, **5**:115–133.

[39] Vaughan R. Pratt, Michael O. Rabin, Larry J. Stockmeyer (1974) A characterization of the power of vector machines. *Proceedings of STOC'1974*, pp. 122—134.

[40] Shipra Saxena (2023) The Architecture of Lenet-5, *Analytics Vidhya* `https://www.analyticsvidhya.com/blog/2021/03/the-architecture-of-lenet-5/`

[41] R. Solomonoff (1964) A Formal Theory of Inductive Inference Part I, and Part II, *Information and Control*, **7**(1/2):1-22;224-254.

[42] Gheorghe M. Ştefan (2021) Pseudo-Reconfigurable Systems, *ROMJIST*, **24**(12):366-383.

[43] Gheorghe M. Stefan (2021) Let's consider Moore's law in its entirety, *2021 International Semiconductor Conference (CAS)*, pp. 3-10,

[44] Max Tegmark (2017) *LIFE 3.0. Being Human in the Age of Artificial Intelligence*, Alfred A. Knopf.

[45] Alan M. Turing (1936-1937) On computable Numbers with an Application to the Eintscheidungsproblem, *Proc. London Mathematical Society,* **42**(1):230–256 and a correction in **43**(6):544–546.

[46] John von Neumann, (1945) First draft of a report on the EDVAC, reprinted in *IEEE Annals of the History of Computing* **15**(4):27–75, 1993.

[47] Jacek M. Zurada (1995) *Introductin toArtificial Neural network*, PWS Pub. Company.

[48] Vincent Weaver, Sally McKee (2009) Code Density Concerns for New Architectures, *2009 IEEE International Conference on Computer Design*, pp. 459–464.

[49] Adam Brooks Webber (2002) *Modern Programming Languages: a Practical Introduction*, Franklin, Beedle & Associates.