FUNCTIONAL ELECTRONICS

*

Lecture Notes on Parallel Embedded Systems

(work in progress)

Gheorghe M. Ştefan

– 2018 version –

Introduction

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

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

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

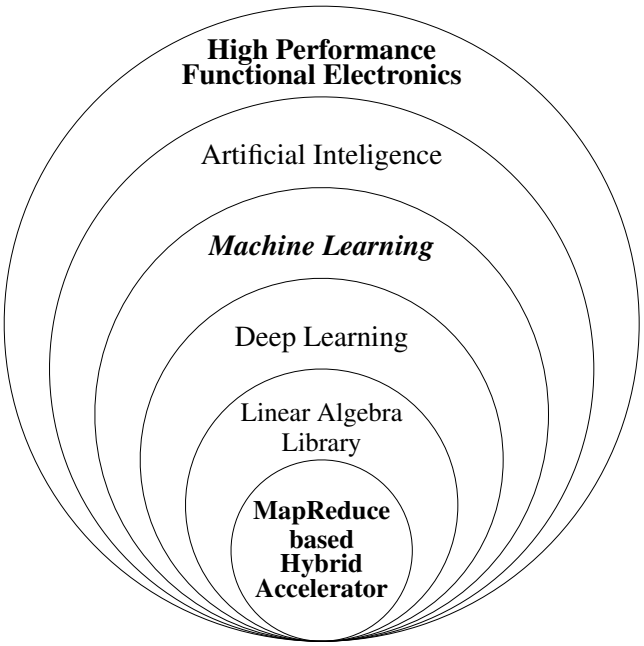


Figure 1:

- AI
- ML
- Deep Learning

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

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

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

Chapter 1 introduces, starting form the computational model proposed by Stephen Kleene, the concept of ***parallel computation***.

Chapter 2 describes the generic version of the parallel accelerator used in these lectures.

Chapter 3 contains the improvements added as a consequence of implementing the first applications (Berkeley's dwarfs).

Chapter 4 presents the 13 class of problems ("dwarfs") typical for parallel accelerators.

Chapter 5 presents the dense linear algebra problems.

Chapter 6 presents the sparse linear algebra problems.

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

Appendix B provides the behavioral description of the parallel accelerator used as an embedded parallel engine.

Appendix C provides the simulator for the accelerator.

[1] PAL: Personal Assistant Linker

Contents

Part I

Engine

Chapter 1

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

1.1 From Kleene's model to MapReduce engine

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

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

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

1.1.1 *Kleene Machine*: a Parallel Model of Computation

According to Theorem A.1 and Theorem A.2 only the composition rule must be considered in defining what means computation.

Definition 1.1 *(*Kleene Machine*) The two-layer construct, associated to the composition rule, see Figure 1.1, with:*

1. ***map*** *level: the independent functions h_i, for $i = 0, 1, \ldots$*

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

where h_i, for $i = 0, 1, \ldots$ and g are initial functions or compositions, is ***Kleene Machine*** *(KM).*

◇

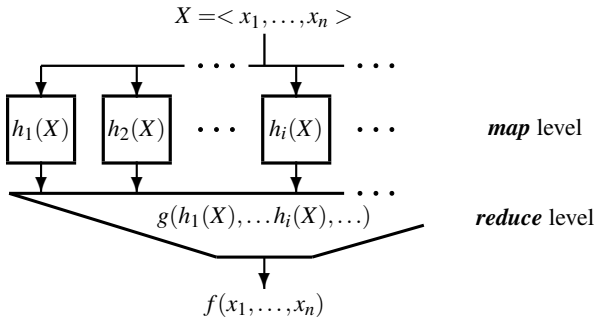


Figure 1.1: Kleene Machine.

Because Kleene's model is equivalent with the Turing Machine model the next corollary is true.

Diachronic parallelism is the parallelism developed at the *map* level.

Synchronic parallelism is the parallelism due to the possible pipeline connection between the *map* level and the *reduce* level.

Corollary 1.1 ***Kleene Machine*** *represents a mathematical model for parallel computation.*

◇

1.1.2 *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 to define any KM on the same structure and to compose KMs.

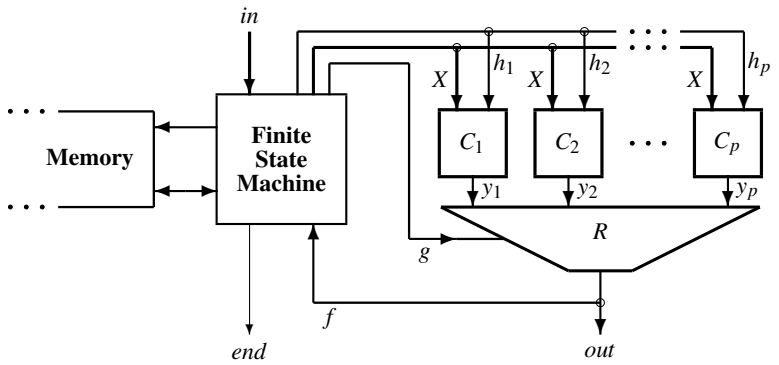


Figure 1.2: Universal Kleene Machine.

Definition 1.2 ***Universal Kleene Machine*** *is a finite KM, with p cells on the map-level, loop connected with a Finite State Machine, FSM, with access to a non-finite Memory. The map-level of KM contains p identical cells, $C_1,\ldots,C_p$, each having the function:*

$$C(h_i,X) = SEL(h_i,H_0(X),H_1(X),\ldots,H_{q-1}(X))$$

while the reduction-level has the function:

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

where: Y is the p-component sequence $< y_1,y_2,\ldots,y_p >$, and the two finite sets of functions

$$\mathbb{H} = \{H_0(X),H_1(X),\ldots,H_{q-1}(X)\}$$
$$\mathbb{G} = \{G_0(X),G_1(X),\ldots,G_{r-1}(X)\}$$

represent the characteristic set of functions, $\mathbb{F} = \mathbb{H}\cup\mathbb{G}$, used to compose any computable function, while $\mathbb{A} = \mathbb{H}\cup\mathbb{G}\cup\mathbb{N}$ represents the alphabet of UKM. Formally:

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

where:

- *S is the finite states set of the FSM*
- *$\mathbb{A}$ is the alphabet of the UKM*
- *$S_0 \in S$ is the initial state of the Finite State Machine*
- *f is the transition function*

$$f : S\times\mathbb{A}\times\mathbb{N} \to S\times\mathbb{H}^p\times\mathbb{N}^v\times\mathbb{G}\times\mathbb{A}$$

which, in each cycle, according to the current state of FSM, the element of $\mathbb{A}$ read from Memory and the output provided by the reduction R, generate the next state of FSM, a stream of p functions for the map-level, a stream of v input values, the function for the reduce-level and the element from $\mathbb{A}$ to be written in Memory.

The UKM is initialized in the state S_0, and after a finite number of cycles, if the computation is possible, the output "end" is activated indicating the end of computation with the result on the output "out".

◇

Theorem 1.1 *The minimal UKM is defined for $p = 3$ with each cells having only one function, as follows:*

$$C_1(X) = ZERO,\ C_2 = INC(SEL(k,X)),\ C_3 = SEL(i,X)$$

while $R = SEL(g,y_0,y_1,y_2)$.

◇

Proof 1.1 *The FSM is used to build the sequence X from the string of outputs (see Figure 1.2). The minimal UKM becomes an UTM.*

◇

1.2 *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 1.3 *A computation model is **competent** if the computation it supports ends in a finite number of steps.*

◇

Definition 1.4 *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.

1.2.1 Forms of Parallelism

Five forms of simplified parallelism (see Figure 1.3) are emphasized as the meaningful set of particular compositions able to provide the transition from a competent model to a performant one.

Definition 1.5 Data-parallel *computation is defined for MC computation (see Definition A.2) when $n = m$ with $h_i(x_1,\ldots,x_n) = h(x_i)$, for $i = 1,\ldots,n$.*

◇

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

Definition 1.6 Reduction-parallel *computation is defined for RC computation (see Definition A.3) when $n = m$ with $h_i(x_1,\ldots,x_n) = h(x_i) = x_i$, for $i = 1,\ldots,n$.*

◇

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

Definition 1.7 Speculative-parallel *computation is defined for MC computation when $n = 1$.*

◇

Each function h_i has the same input variable x_1.

Definition 1.8 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 1.9 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:

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

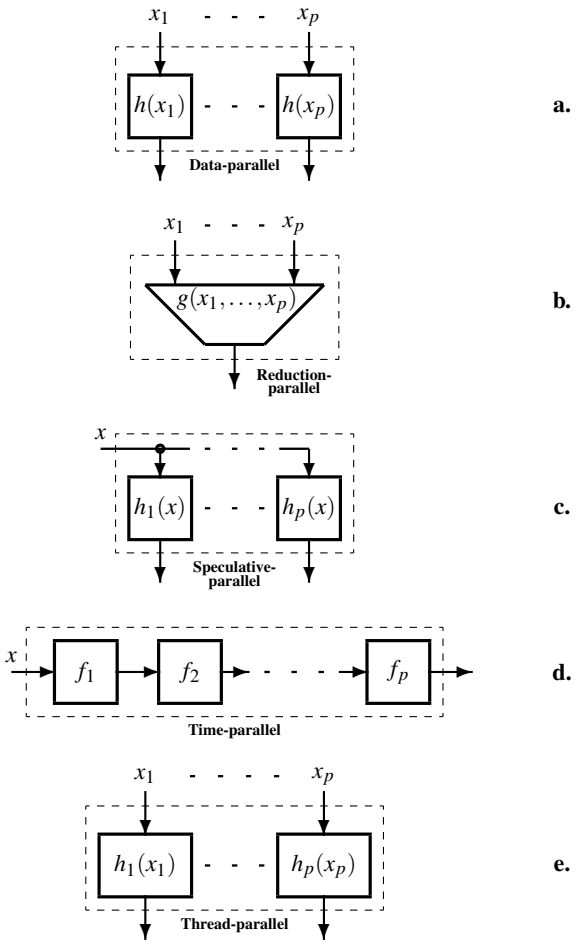


Figure 1.3: Five types of parallelism as particular forms of composition (see Figure 1.2)

1.2.2 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 1.4, the MapReduce recursive parallel abstract model for parallel computation is presented. It consists of:

- pairs *eng-mem* in the MAP section; they correspond to the cells C_i from UKM, and consist of:
  - *eng*, the engine, which is an execution unit or a processing unit
  - *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 1.4 can be seen as a two-part entity:

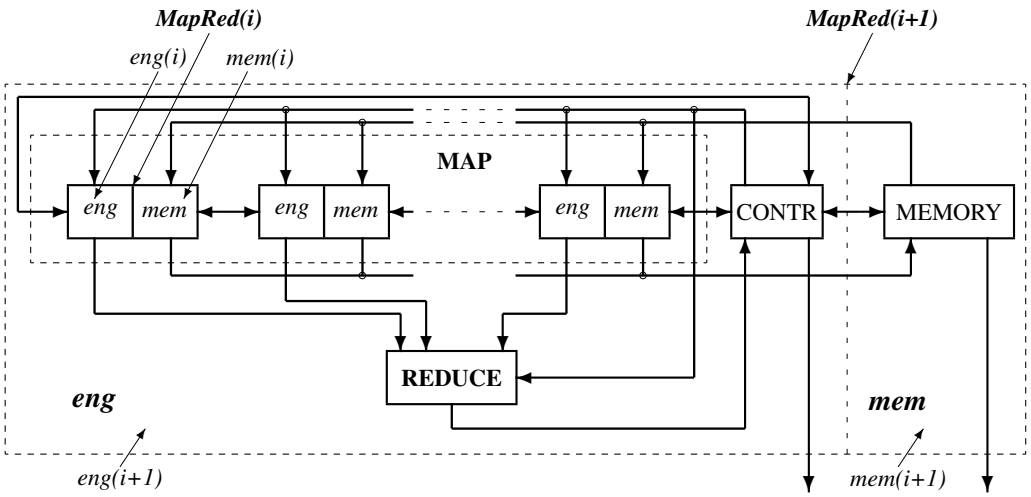


Figure 1.4: ***MapReduce recursive abstract model*** for parallel computation.

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

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

1.3 A Programming Model

1.3.1 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 [**?**], 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 [**?**], 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.

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

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

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

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

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

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

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

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

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

$$id : x \equiv x$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- **Permute** : the argument is a sequence of two equally length sequences; the first defines the permutation, while the second is submitted to the permutation.
  $perm : x \equiv$
  $(x =<< y_1, \ldots, y_p >, < x_1, \ldots, x_p >>) \rightarrow$
  $< x_{y_1}, \ldots, x_{y_p} >$
  With no special hardware support it is performed in time $O(p)$. An optimal implementation, in time belonging to $O(log\, p)$, involves a *redLoopNet* containing a Waksman permutation network, with $< y_1, \ldots, y_p >$ used to program it.

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

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

- **Arithmetic & logic operations** :

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

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

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

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

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

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

Functional Forms

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

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

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

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

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

- **Construction** : represents the ***speculative-parallel*** computation. The same argument is used by a sequence of functions.

  $$[f_1, \ldots, f_n] : x \equiv < f_1 : x, \ldots, f_n : x >$$

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

  - sequential computation, if only one object x is considered as input variable
  - pipelined *time-parallel* computation, if a ***stream*** of objects, $|x_n, \ldots, x_1|$, are considered to be inserted, starting with x_1, in c_1 in the MANY-CORE section (see Figure **??**) so as in each successive two cells, c_i and c_{i+1}, are performed

    $$f_i(f_{i-1} : (f_{i-2} : (\ldots : (f_1 : x_j) \ldots)))$$

    $$f_{i+1}(f_i : (f_{i-1} : (\ldots : (f_1 : x_{j-1}) \ldots)))$$

    Thus, the array of cells $c_1, \ldots, c_p$ can be involved to compute in parallel the function

    $$f(x) = (f_q \circ f_{q-1} \circ \ldots \circ f_1) : x$$

    for maximum q values of x.

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

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

- **Binary to unary** : is used to express any function as an unary function.

$$(bu\, f\, x) : y \equiv f :< x,y >$$

  This function allows the algebraic manipulation of programs.

Definitions

Definitions are used to write programs conceived as functional forms.

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

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

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

1.3.2 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.

1.3.3 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`.

1.3.4 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 1.1 *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 2

The Generic Parallel Engine

2.1 The Structure

The cellular structure of the generic version is accompanied by a generic scalar processing structure used as controller. In the cellular structure all the resources are of vectorial type. The instruction set architecture works on four storage resources:

- scalar computational resources, in the controller
- sequential control resources, in the controller
- vectorial computational resources, distributed along the array of cells
- spatial control resources, distributed along the array of cells
- evaluation resources, used to evaluate the performance of the execution

described as follows:

```
// vectorial resources
reg [x-1:0] ixVect[0:(1<<x)-1]              ; // index read-only vector
reg [a-1:0] actVect[0:(1<<x)-1]             ; // activation vector
reg         boolVect[0:(1<<x)-1]            ; // Boolean vector
reg [n-1:0] accVect[0:(1<<x)-1]             ; // accumulator vector
reg         crVect[0:(1<<x)-1]              ; // carry vector
reg [v-1:0] addrVect[0:(1<<x)-1]            ; // address vector
reg [n-1:0] vectMem[0:(1<<x)-1][0:(1<<v)-1] ; // vector memory
// scalar resources
reg [n-1:0] acc                             ; // scalar accumulator
reg         cr                              ; // scalar carry
reg [s-1:0] addr                            ; // scalar address
reg [n-1:0] mem[0:(1<<s)-1]                 ; // scalar memory
// control resources
reg [p-1:0] pc                              ; // program counter
reg [31:0]  ir                              ; // instruction register
reg [31:0]  progMem[0:(1<<p)-1]             ; // program memory
```

```
// evaluation resources
reg [31:0]  cc                              ; // cycle counter
reg         ccEnable                        ; // cycle counter enable
```

The sizes of different parts of the engine are defined in the file `00_parameters`. A possible version of this file (used in our simulations) is:

```
parameter   n = 16  , // word size
            x = 4   , // index size -> number of cells is 2^4 = 16
            v = 6   , // vector memory address size
            s = 8   , // scalar memory address size
            p = 8   , // program memory address size
            c = 8   , // value size in instruction
            a = 5     // size of activation counter
```

The generic structure starts with the simplest and smallest resources, like:

- each of the 2^x cell's engine is an execution unit (not a processing unit)
- both, the execution unit of the controller and the execution unit of each cell are designed as accumulator based engines
- 32-bit interface to the external memory

The above introduced generic structure is used as the starting point in featuring the MapReduced abstract model with those structural resources which fit better with the just born parallel engine. Overfeaturing the staring point provides the danger to introduce useless, parasitic elements in our growing structure. In this early stage of developing an accelerating engine it is hard to answer correct questions like:

- do we need execution units or processing units to equip cells
- what kind of execution unit is to be preferred? Stack oriented or register file oriented?
- if the register file is the solution, then how many register to use?
- is the stack solution is preferred, then how deep it must be?
- ...

Only after starting to use the generic structure to solve real problems, we will be able to answer questions like the previously listed and many others. Then, we will learn what are the features to be added, and we will find the appropriate sizes for the subsystems of the MapReduce accelerator.

The minimal structural requirements associated to the physical resources just listed before are:

computational resources :

- scalar to scalar operations:
  `acc <= acc OP mem[address]`
  `acc <= UOP(acc)`

- vector to vector operations:
  `accVect <= accVect OP vectMem[address]`
  `accVect <= UOP(accVect)`
- vector to scalar operations (reduction operations):
  `acc <= ROP({accVect[0], ..., accVect[m-1]})`

where: `OP` is a binary operation, `UOP` is a unary operation, `ROP` is a reduction (m-ary) operation, while $m = 2^x$ is the number of cells in the array.

control resources :

- sequential control resources (usual jumps and conditioned branches in program)
- spatial control resources (used to select the vector components involved in computation)

the Therefore, each cell of the array is designed to act on one vector component, while a the controller takes care of the scalar variables. For the scalar to vector operations, a log-depth reduction circuit is provided. The reduction circuit takes the vector from the array of cells and deliver the scalar to the controller unit.

2.2 The 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)

2.2.1 The Instruction Structure

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

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

where:

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

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

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

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

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

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

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

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

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

`mac = 3'b101` : absolute, from the local memory of the controller addressed with co-operand `op = mem[coop]`

`mrc = 3'b110` : relative from local memory with co-operand `op = mem[value + coop]`

`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: $\Sigma_0^p\ bool_i$

2.2.2 Instruction Set Architecture

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

```
add      = 5'b00000, // {cr, acc} <= acc + op;
addc     = 5'b00001, // {cr, acc} <= acc + op + cr;
sub      = 5'b00010, // {cr, acc} <= acc - op;
rsub     = 5'b00011, // {cr, acc} <= op - acc;
subc     = 5'b00100, // {cr, acc} <= acc - op - cr;
rsubc    = 5'b00101, // {cr, acc} <= op - acc - cr;
div      = 5'b00110, // acc <= acc/op;
rdiv     = 5'b00111, // acc <= op/acc;
mult     = 5'b01000, // acc <= acc * op;
bwand    = 5'b01001, // acc <= acc & op;
bwor     = 5'b01010, // acc <= acc | op;
bwxor    = 5'b01011, // acc <= acc ^ op;
load     = 5'b01100, // acc <= op;
store    = 5'b01101, // op <= acc;
search   = 5'b01110, // boolVect <= (acc=op) ? 1 : 0;
csearch  = 5'b01111, // boolVect <= (acc=op) & (boolVect >> 1) ? 1 : 0;
insert   = 5'b10000, // insert op ar first
compare  = 5'b10001, // {cr, acc} <= (acc - op) & (100...0) | {0, acc};
fadd     = 5'b10010, // float add

jmp      = 5'b00000, // pc <= pc + scalar;
brz      = 5'b00001, // pc <= acc=0 ? pc+scalar : pc+1;
brnz     = 5'b00010, // pc <= acc=0 ? pc+1 : pc+scalar;
brzdec   = 5'b00011, // pc <= acc=0 ? pc+scalar : pc+1; acc <= acc - 1
brnzdec  = 5'b00100, // pc <= acc=0 ? pc+1 : pc+scalar; acc <= acc - 1
brcr     = 5'b00101, // pc <= cr ? pc+scalar : pc+1;
brncr    = 5'b00110, // pc <= cr ? pc+1 : pc+scalar;

start    = 5'b01110, // start cycle counter
stop     = 5'b01111, // stop cycle counter
```

```
where    = 5'b00000, // boolVect <= (boolVect & cond[scalar[1:0]]) ? 1 : 0;
wheren   = 5'b00001, // boolVect <= (boolVect & cond[scalar[1:0]]) ? 0 : 1;
elsew    = 5'b00010, // boolVect <= ~boolVect;
endwhere = 5'b00011, // boolVect <= 1;

grotate  = 5'b01000, // global rotate
glshift  = 5'b01001, // global leftt shift
grshift  = 5'b01010, // global right shift
brshift  = 5'b01011, // Boolean vector right shift; for read from array
delete   = 5'b01100, // delete first
ixload   = 5'b01101, // index load: acc[i] <= i

insval   = 5'b10000, // acc <= {acc[23:0], scalar}
shrightc = 5'b10001, // {cr, acc} <= {acc[0], cr, acc[n-1:1]}
shright  = 5'b10010, // {cr, acc} <= {acc[0], 1'b0, acc[n-1:1]}
sharight = 5'b10011, // {cr, acc} <= {acc[0], acc[n-1], acc[n-1:1]}
penc     = 5'b10100, // {cr, acc} <= {(acc==0), 27'b0, pe(acc)}

vload    = 5'b11110, // i=0...p vectMem[aScalar][i] <= mem[acc + i*cScalar]
vstore   = 5'b11111; // i=0...p mem[acc + i*cScalar] <= vectMem[aScalar][i]
```

2.2.3 The Assembler Language

The assembly language provides a sequence of lines each containing an instruction for Controller (with the prefix `c`) and another for Array. For jumps and branches, some of the line are labeled (`LB(n)`, where *n* is a positive integer).

Two-operand instructions

The pattern for the two-operand instruction is presented using the function `ADD` (addition). Each of the two-operand instruction has the following 12 forms (5 for Controller and 7 for Array) according to the way the second operand is selected. (For the sake of simplicity, in the following, `acc[i]` stands for `accVect[i]` and `cr[i]` stands for `crVect`.)

immediate add :

```
cVADD:  {carry, acc}       <= acc + {(n-8){cScalar[7]}}, cScalar}
VADD:   {carry[i], acc[i]} <= acc[i] + {(n-8){aScalar[7]}}, aScalar}
```

absolute add :

```
cADD:   {carry, acc}       <= acc + mem[cScalar]
ADD:    {carry[i], acc[i]} <= acc[i] + vectMem[i][aScalar]
```

relative add :

```
cRADD:  {carry, acc}       <= acc + mem[cScalar]
RADD:   {carry[i], acc[i]} <= acc[i] + vectMem[i][aScalar]
```

relative add & increment :

```
cRIADD: {carry, acc}       <= acc + mem[addr + cScalar]
        addr               <= addr + cScalar
RIADD:  {carry[i], acc[i]} <= acc[i] + vectMem[i][addrVect[i] + aScalar]
        addrVect[i]        <= addrVect[i] + aScalar
```

co-operand add :

```
cCADD:  {carry, acc}       <= acc + cop
                                - scalar = 00: cop = reduction min
                                - scalar = 01: cop = reduction add
                                - scalar = 10: cop = reduction max
                                - scalar = 11: cop = reduction flag
CADD:   {carry[i], acc[i]} <= acc[i] + acc
CAADD:  {carry[i], acc[i]} <= acc[i] + vectMem[i][acc]
CRADD:  {carry[i], acc[i]} <= acc[i] + vectMem[i][addrVect[i] + acc]
```

For the following mnemonics, the previously described 12 instructions forms are the same:

```
ADDC    - add with carry: {carry, acc} <= acc + op + carry
SUB     - subtract:       {carry, acc} <= acc - op
RSUB    - reverse SUB:    {carry, acc} <= op - acc
SUBC    - SUB with carry: {carry, acc} <= acc - op - carry
RSUBC   - reverse SUBC:   {carry, acc} <= op - acc - carry
DIV     - division:                     acc <= acc / op
RDIV    - reverse DIV:                  acc <= op / acc
MULT    - multiplication:               acc <= acc * op
AND     - bitwise and:                  acc <= acc & op
OR      - bitwise or:                   acc <= acc | op
XOR     - bitwise xor:                  acc <= acc ^ op
COMPARE - compare:        {carry, acc} <= (acc - op)&(10...0)|{0, acc};
```

Thus, instead of the suffix `ADD` in one of the previous 12 instruction descriptions, one of the previous can be used. For example: `VADDC`, instead of `VADD`. Thus, 12×13 instructions are already described.

Shift instructions

shift right one bit position :

```
cSHRIGHT:  {cr, acc}       <= {acc[0], 1'b0, acc[n-1:1]}
SHRIGHT:   {cr[i], acc[i]} <= {acc[i][0], 1'b0, acc[i][n-1:1]}
```

shift right one bit position with carry :

```
cSHRIGHTC:  {cr, acc}       <= {acc[0], cr, acc[n-1:1]}
SHRIGHTC:   {cr[i], acc[i]} <= {acc[i][0], cr[i], acc[i][n-1:1]}
```

shift right arithmetic one bit position :

```
cSHARIGHT:  {cr, acc}       <= {acc[0], acc[n-1], acc[n-1:1]}
SHARIGHT:   {cr[i], acc[i]} <= {acc[i][0], acc[i][n-1], acc[i][n-1:1]}
```

insert `value` **on the least positions** :

```
cINSVAL:  acc    <= {acc[(n-c):0], cScalar}
INSVAL:   acc[i] <= {acc[i][(n-c):0], aScalar}
```

Address register load instructions

are used to instantiate the value of the address register in controller, `addr`, and in each cell of the array, `addrVect[i]`.

```
cADDRLD:  addr        <= acc
ADDRLD:   addrVect[i] <= acc[i]
CADDRLD   addrVect[i] <= acc
```

Load instructions

immediate load :

```
cVLOAD:  acc    <= {(n-8){cScalar[7]}}, cScalar}
VLOAD:   acc[i] <= {(n-8){aScalar[7]}}, aScalar}
```

absolute load :

```
cLOAD:   acc    <= mem[cScalar]
LOAD:    acc[i] <= vectMem[i][aScalar]
```

relative load :

```
cRLOAD:  acc    <= mem[addr + cScalar]
RLOAD:   acc[i] <= vectMem[i][addrVect[i] + aScalar]
```

relative load & increment :

```
cRILOAD:  acc         <= mem[addr + cScalar
          addr        <= addr + cScalar
RILOAD:   acc[i]      <= vectMem[i][addrVect[i] + aScalar]
          addrVect[i] <= addrVect[i] + aScalar
```

co-operand load :

```
cCLOAD:  acc    <= cop
                        - scalar = 00: cop = reduction min
                        - scalar = 01: cop = reduction add
                        - scalar = 10: cop = reduction max
                        - scalar = 11: cop = reduction flag
CLOAD:   acc[i] <= acc
CALOAD:  acc[i] <= vectMem[i][acc]
CRLOAD:  acc[i] <= vectMem[i][addrVect[i] + acc]
```

index load :

```
IXLOAD:  acc[i] <= i
```

Store instructions

absolute store :

```
cSTORE:  mem[cScalar]        <= acc
STORE:   vectMem[i][aScalar] <= acc[i]
```

relative store :

```
cRSTORE: mem[addr + cScalar]               <= acc
RSTORE:  vectMem[i][addrVect[i] + aScalar] <= acc[i]
```

relative store & increment :

```
cRISTORE: mem[addr + cScalar]               <= acc
          addr                              <= addr + cScalar
RISTORE:  vectMem[i][addrVect[i] + aScalar] <= acc[i]
          addrVect[i]                       <= addrVect[i] + aScalar
```

co-operand store :

```
CSTORE:   vectMem[i][acc]               <= acc[i]
CRSTORE:  vectMem[i][addrVect[i] + acc] <= acc[i]
```

Sequential control instructions

unconditioned jump :

```
cJMP:     pc <= pc + cScalar
```

branch if `acc` **is zero** :

```
cBRZ:     pc <= (acc = 0) ? pc + cScalar : pc + 1
```

branch if `acc` **is not zero** :

```
cBRNZ:    pc <= (acc = 0) ? pc + 1 : pc + cScalar
```

```
cBRZDEC:  pc  <= (acc = 0) ? pc + cScalar : pc + 1
          acc <= acc - 1
```

branch if `acc` **is not zero** :

```
cBRNZDEC: pc  <= (acc = 0) ? pc + 1 : pc + cScalar
          acc <= acc - 1
```

halt :

```
cHALT:    pc  <= pc
```

Spatial control instructions

are used to select the active cells. The cell `i` is active if `boolVect[i] = 1`, where: `boolVect[i] = (actVect[i] == 0)`.

activate all cells :

```
ACTIVATE: actVect[i] <= 0
```

keep active where zero :

```
WHEREZERO: actVect[i] <= (boolVect[i]&(condVect[i][0])) ?  actVect[i] :
                                                           actVect[i]+1
```

keep active where carry :

```
WHERECARRY: actVect[i] <= (boolVect[i]&(condVect[i][1])) ?  actVect[i] :
                                                            actVect[i]+1
```

keep active where first :

```
WHEREFIRST: actVect[i] <= (boolVect[i]&(condVect[i][2])) ?  actVect[i] :
                                                            actVect[i]+1
```

keep active where next :

```
WHERENEXT: actVect[i] <= (boolVect[i]&(condVect[i][3])) ?  actVect[i] :
                                                           actVect[i]+1
```

keep active where not zero :

```
WHERENZERO: actVect[i] <= (boolVect[i]&!(condVect[i][0])) ?  actVect[i] :
                                                             actVect[i]+1
```

keep active where not carry :

```
WHERENCARRY: actVect[i] <= (boolVect[i]&!(condVect[i][1])) ?  actVect[i] :
                                                              actVect[i]+1
```

keep active where not first :

```
WHERENFIRST: actVect[i] <= (boolVect[i]&!(condVect[i][2])) ?  actVect[i] :
                                                              actVect[i]+1
```

keep active where not next :

```
WHERENNEXT: actVect[i] <= (boolVect[i]&!(condVect[i][3])) ?  actVect[i] :
                                                             actVect[i]+1
```

activate the cells inactivated by the last `where` :

```
ELSEWHERE: actVect[i] <= (actVect[i]==0) ? 1 : ((actVect[i]==1) ? 0 :
                                                                  actVect[i])
```

restore `actVect` **before the corresponding** `where` :

```
ENDWHERE: actVect[i] <= (actVect[i] == 0) ? actVect[i] : (actVect[i] - 1)
```

Global instructions

global rotate with one position :

```
GROTATE: acc[i] <= acc[(i+1)%(1<<x)]
```

global right shift with one position :

```
GRSHIFT:  accVect[i] <= (i==0) ? 0 : accVect[i-1]
```

global left shift with one position :

```
GLSHIFT: acc[i] <= (i < (1<<x)-1) ? acc[i+1] : 0
```

search for co-operand :

```
SEARCH: boolVect[i] <= (boolVect[i] & (acc[i] == acc)) ? 1'b1 : 1'b0
```

search for value :

```
VSEARCH: boolVect[i] <= (boolVect[i] & (acc[i] == aScalar)) ? 1'b1 : 1'b0
```

conditioned search for co-operand :

```
CSEARCH: boolVect[i] <= (i==0) ? 0 : ((acc[i] == acc) & boolVect[i-1]) ? 1 : 0
```

conditioned search for value :

```
VCSEARCH: boolVect[i] <= (i==0) ? 0 : ((acc[i]==aScalar)&boolVect[i-1]) ? 1 : 0
```

insert value in the first active position :

```
INSERT: acc[i] <= (firstVect[i]) ? aScalar : ((nextVect[i]) ? acc[i-1] : acc[i])
```

insert co-operand in the first active position :

```
CINSERT: acc[i] <= (firstVect[i]) ? acc : ((nextVect[i]) ? acc[i-1] : acc[i])
```

delete the first active accumulator :

```
DELETE: acc[i] <= (firstVect[i] | nextVect[i]) ?
                  ((i == ((1<<x)-1)) ? 0 : acc[i+1]) : acc[i]
```

move selections of active cells one position right :

```
SELSHIFT: boolVect[i] <= (i == 0) ? 1'b0 : boolVect[i-1]
```

2.2.4 How to Use the Assembler

In order to evaluate what can be the maximum of performance of our architecture, the assembly language must be used. Therefore, the initial stage of evaluation must be done in assembly language. (The next stage, beyond the scope of our approach, is to provide an efficient compiler from a high level language to the machine language.) We provide few simple example of using the previously described assembly language. The behavioral description of the generic structure is simulated on **ISE Design Suite 14.2** provided by Xilinx.

For simulation reasons, the engine is kept small. It is defined by the content of `00_parameters.v` file:

```
parameter
    n = 32  , // word size
    x = 4   , // index size -> 16 cells
    v = 6   , // vector memory address size -> 64 vectors
    s = 8   , // scalar memory address size -> 256 scalars
    p = 8   , // program memory address size -> 256 32-bit instructions
    c = 8   , // value size in instruction
    a = 5     // size of activation counter
```

For editorial reasons, the simulator's monitor has the following, "compressed" form:

```
initial begin
  $monitor("t=%0d pc=%d a=%0d a[0]=%0d a[1]=%0d a[2]=%0d ... a[6]=%0d a[7]=%0d
            b = %0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d cc=%0d",
            $time/2,
            dut.pc,
            dut.acc,
            dut.accVect[0],
            dut.accVect[1],
            dut.accVect[2],
            dut.accVect[14],
            dut.accVect[15],
            dut.boolVect[0],  dut.boolVect[1],  dut.boolVect[2],  dut.boolVect[3],
            dut.boolVect[4],  dut.boolVect[5],  dut.boolVect[6],  dut.boolVect[7],
            dut.boolVect[8],  dut.boolVect[9],  dut.boolVect[10], dut.boolVect[11],
            dut.boolVect[12], dut.boolVect[13], dut.boolVect[14], dut.boolVect[15],
            dut.cc
        );
end
```

The following examples are presented in order to show how the main features of the MapReduce engine, with the generic architecture, works.

Example 2.1 *The program which provide in the controller's accumulator,* `acc`*, the sum of indexes loaded in the accumulators of each cell,* `accVect[i]`*, is:*

```
/*****************************************************************************
Test program: 03_examples.v (appropriately commented)
    - activate all cells
    - acc[i] <= i, for i = 0,1,...,15
    - acc <= acc[0]+acc[1]+...+acc[15]
*****************************************************************************/
//*
    cSTART;     ACTIVATE; // start cycle counter; activate all cells
    cNOP;       IXLOAD;   // load the index of each cell in accumulator
    cNOP;       NOP;      // latency step 1
    cNOP;       NOP;      // latency step 2
    cNOP;       NOP;      // latency step 3
    cCLOAD(0);  NOP;      // acc <= sum of indexes
    cSTOP;      NOP;      // stop cycle counter
    cNOP;       NOP;      // to show cycle coounter stoped
    cHALT;      NOP;
//*/
//============================================================================
```

Appropriately commented means `//*` *instead of* `/*` *before the first line of code.*
The assembled code, provided by the simulator, is:

```
progMem[0]   = 00110111000000000111011100000000
progMem[1]   = 01101111000000000000000000000000
progMem[2]   = 00000000000000000000000000000000
progMem[3]   = 00000000000000000000000000000000
progMem[4]   = 00000000000000000000000000000000
progMem[5]   = 00000000000000000110010000000000
progMem[6]   = 00000000000000000111111100000000
progMem[7]   = 00000000000000000000000000000000
progMem[8]   = 00000000000000000000011100000000
```

The result of simulation is:

```
t=0 pc=  x a=x   a[0]=x a[1]=x a[2]=x ... a[14]=x  a[15]=x  b=xxxxxxxxxxxxxxxx cc=x
t=0 pc=255 a=x   a[0]=x a[1]=x a[2]=x ... a[14]=x  a[15]=x  b=xxxxxxxxxxxxxxxx cc=0
t=1 pc=  0 a=x   a[0]=x a[1]=x a[2]=x ... a[14]=x  a[15]=x  b=xxxxxxxxxxxxxxxx cc=0
t=2 pc=  1 a=x   a[0]=x a[1]=x a[2]=x ... a[14]=x  a[15]=x  b=1111111111111111 cc=0
t=3 pc=  2 a=x   a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=1
t=4 pc=  3 a=x   a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=2
t=5 pc=  4 a=x   a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=3
t=6 pc=  5 a=x   a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=4
t=7 pc=  6 a=120 a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=5
```

```
t=8 pc=  7 a=120 a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=6
t=9 pc=  8 a=120 a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=6
```

In the initial cycle (`t=0`*) the system reset resets the cycle counter,* `cc = 0`*. In the first cycle (*`t=1`*) the program activate all the cells of the array (the Boolean vector is filled up with 1s). The operation is validated in the next cycle when* `b <= 11...1`*. Then the accumulator in each cell takes the value of* `index`*. During three cycles the reduction network computes the sum of indexes. Then in* `t=7` *the controller's accumulator is loaded with the sum of indexes, i.e., the sum of the numbers* `acc = 0 + 1 + 2 + ... + 15 = 120`*, because we instantiated for our simulation an array with 16 cells. The cycle counter stops in the next cycle on the value 6, which means: the program performed the task in* `cc - 1 = 5` *cycles (the instruction which stops the counter is also counted).*

◇

Example 2.2 *The program which stores at* `mem[24]` *the inner product of the index vector with itself is:*

```
/*****************************************************************************
Test program: 03_examples.v (appropriately commented)
    - activate all cells
    - acc[i] <= i for 1 = 0,1,...,15
    - memVect[i][4] <= acc[i] for 1 = 0,1,...,15
    - acc[i] <= acc[i] x vectMem[i][4]
    - acc <= acc[0]+acc[1]+...+acc[15]
    - mem[24] <= acc = innerProduct(index, index)
*****************************************************************************/
//*
    cSTART;     ACTIVATE; // activate all cells
    cNOP;       IXLOAD;   // acc[i] <= index
    cNOP;       STORE(4); // memVect[i][4] <= acc[i], for all i
    cNOP;       MULT(4);  // acc[i] <= acc[i] * memVect[i][4]
    cNOP;       NOP;      // latency step 1
    cNOP;       NOP;      // latency step 2
    cNOP;       NOP;      // latency step 3
    cCLOAD(0);  NOP;      // acc <= reductionAdd(acc[i])
    cSTORE(24); NOP;      // mem[24] <= acc
    cSTOP;      NOP;      // stop cycle counter
    cHALT;      NOP;
//*/
//============================================================================
```

The simulation provides, a little edited to fit in page, the following results:

```
t=1  pc= 0 a=x    a[0]=x a[1]=x a[2]=x ... a[14]=x   a[15]=x   b=xx...x cc=0
t=2  pc= 1 a=x    a[0]=x a[1]=x a[2]=x ... a[14]=x   a[15]=x   b=11...1 cc=0
t=3  pc= 2 a=x    a[0]=0 a[1]=1 a[2]=2 ... a[14]=14  a[15]=15  b=11...1 cc=1
t=4  pc= 3 a=x    a[0]=0 a[1]=1 a[2]=2 ... a[14]=14  a[15]=15  b=11...1 cc=2
t=5  pc= 4 a=x    a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=3
t=6  pc= 5 a=x    a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=4
t=7  pc= 6 a=x    a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=5
t=8  pc= 7 a=x    a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=6
```

```
t=9  pc= 8 a=1240 a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=7
t=10 pc= 9 a=1240 a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=8
t=11 pc=10 a=1240 a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=9
```

In cycle 9 *the controllers accumulator is loaded with the value of the inner product and in the next cycle its content is stored in the local scalar memory.*

◇

Example 2.3 *The program which provide in* acc *the number of components of the index vector bigger than 5 and smaller than 15 is:*

```
/*****************************************************************************
Test program: 03_examples.v (appropriately commented)
    - activate all cells
    - 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
    - acc <= acc[0]+acc[1]+...+acc[15] only for the active cells
*****************************************************************************/
//*
    cNOP;       ACTIVATE;    // activate all cells
    cNOP;       IXLOAD;      // acc[i] <= index
    cNOP;       VSUB(5);     // {cr, acc[i]} <= acc[i] - 5
    cNOP;       WHERENCARRY; // where cr=1 remain active
    cNOP;       VSUB(10);    // {{cr, acc[i]} <= acc[i] - (15 - 5)
    cNOP;       WHERECARRY;  // where cr=0 remain active
    cNOP;       VLOAD(1);
    cNOP;       ENDWHERE;    // reactivate where the second WHERE acted
    cNOP;       ENDWHERE;    // reactivate where the first WHERE acted
    cNOP;       NOP;         // latency step 3
    cCLOAD(0);  NOP;         // acc <= number of active cells
    cHALT;      NOP;
//*/
//============================================================================
```

The simulation provides:

```
t=1  pc= 0 a=x  a[0]=x          ... a[6]=x          a[7]=x   b=xxxxxxxxxxxxxxxx
t=2  pc= 1 a=x  a[0]=x          ... a[6]=x          a[7]=x   b=1111111111111111
t=3  pc= 2 a=x  a[0]=0          ... a[6]=14         a[7]=15  b=1111111111111111
t=4  pc= 3 a=x  a[0]=4294967291 ... a[14]=9         a[15]=10 b=1111111111111111
t=5  pc= 4 a=x  a[0]=4294967291 ... a[14]=9         a[15]=10 b=0000011111111111
t=6  pc= 5 a=x  a[0]=4294967291 ... a[14]=4294967295 a[15]=0  b=0000011111111111
t=7  pc= 6 a=x  a[0]=4294967291 ... a[14]=4294967295 a[15]=0  b=0000011111111110
t=8  pc= 7 a=x  a[0]=4294967291 ... a[14]=1         a[15]=0  b=0000011111111110
t=9  pc= 8 a=x  a[0]=4294967291 ... a[14]=1         a[15]=0  b=0000011111111111
t=10 pc= 9 a=x  a[0]=4294967291 ... a[14]=1         a[15]=0  b=1111111111111111
t=11 pc=10 a=x  a[0]=4294967291 ... a[14]=1         a[15]=0  b=1111111111111111
t=12 pc=11 a=10 a[0]=4294967291 ... a[14]=1         a[15]=0  b=1111111111111111
```

Indeed, the index vector contains 10 components bigger then 5 and smaller than 15.
◇

Example 2.4 *Load index in cell's accumulators and do $l = 9$ times: divide by 2 (integer operation) and increment with 99 each accumulator. The program is:*

```
/*****************************************************************************
Test program: 03_examples.v (appropriately commented)
    - activate all cells
    - acc <= 8; initialize the loop counter with l-1
    - acc[i} <= i; load index
    - do (acc+1) times
          acc[i] <= acc[i]/2
          acc[i] <= acc[i] + 99
*****************************************************************************/
//*
        cNOP;           ACTIVATE;
        cVLOAD(8);      IXLOAD;
LB(1);  cNOP;           SHRIGHT;
        cBRNZDEC(1);    VADD(99); // branch if acc=0 and acc<=acc-1
        cHALT;          NOP;
//*/
//============================================================================
```

The simulation provides:

```
t=1  pc=0 a=x          a[0]=x   a[1]=x   a[2]=x   ... a[14]=x   a[15]=x
t=3  pc=2 a=8          a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15
t=4  pc=3 a=8          a[0]=0   a[1]=0   a[2]=1   ... a[14]=7   a[15]=7
t=5  pc=2 a=7          a[0]=99  a[1]=99  a[2]=100 ... a[14]=106 a[15]=106
t=6  pc=3 a=7          a[0]=49  a[1]=49  a[2]=50  ... a[14]=53  a[15]=53
t=7  pc=2 a=6          a[0]=148 a[1]=148 a[2]=149 ... a[14]=152 a[15]=152
t=8  pc=3 a=6          a[0]=74  a[1]=74  a[2]=74  ... a[14]=76  a[15]=76
t=9  pc=2 a=5          a[0]=173 a[1]=173 a[2]=173 ... a[14]=175 a[15]=175
t=10 pc=3 a=5          a[0]=86  a[1]=86  a[2]=86  ... a[14]=87  a[15]=87
t=11 pc=2 a=4          a[0]=185 a[1]=185 a[2]=185 ... a[14]=186 a[15]=186
t=12 pc=3 a=4          a[0]=92  a[1]=92  a[2]=92  ... a[14]=93  a[15]=93
t=13 pc=2 a=3          a[0]=191 a[1]=191 a[2]=191 ... a[14]=192 a[15]=192
t=14 pc=3 a=3          a[0]=95  a[1]=95  a[2]=95  ... a[14]=96  a[15]=96
t=15 pc=2 a=2          a[0]=194 a[1]=194 a[2]=194 ... a[14]=195 a[15]=195
t=16 pc=3 a=2          a[0]=97  a[1]=97  a[2]=97  ... a[14]=97  a[15]=97
t=17 pc=2 a=1          a[0]=196 a[1]=196 a[2]=196 ... a[14]=196 a[15]=196
t=18 pc=3 a=1          a[0]=98  a[1]=98  a[2]=98  ... a[14]=98  a[15]=98
t=19 pc=2 a=0          a[0]=197 a[1]=197 a[2]=197 ... a[14]=197 a[15]=197
t=20 pc=3 a=0          a[0]=98  a[1]=98  a[2]=98  ... a[14]=98  a[15]=98
t=21 pc=4 a=4294967295 a[0]=197 a[1]=197 a[2]=197 ... a[14]=197 a[15]=197
```

The initial value of `accVect` *is* {`0, 1, ..., 14, 15`}. *After 8 execution of the two cycles loop it becomes:* {`197, 197, ..., 197, 197`}.

◇

Example 2.5 *Add in* `accVect` *index with the sum of all indexes. The program is:*

```
/*************************************************************************
Test program: 03_examples.v (appropriately commented)
    - activate all cells
    - acc[i} <= i; load index
    - acc <= acc[0]+acc[1]+...acc[15]
    - acc[i] <= acc[i] + acc
*************************************************************************/

cSTART;     ACTIVATE; // ccEnable <= 1; boolVect <= 11...1
cNOP;       IXLOAD;   // acc[i] <= i
cNOP;       NOP;      // latency step 1
cNOP;       NOP;      // latency step 2
cNOP;       NOP;      // latency step 3
cCLOAD(0);  NOP;      // acc <= acc[0] + acc[1] + ... acc[15]
cNOP;       CADD;     // acc[i] <= acc[i] + acc
cSTOP;      NOP;      // stop cycle counter
cHALT;      NOP;
```

The result of simulation is:

```
t=1 pc=0 a=x   a[0]=x   a[1]=x   a[2]=x   ... a[14]=x   a[15]=x   cc=0
t=2 pc=1 a=x   a[0]=x   a[1]=x   a[2]=x   ... a[14]=x   a[15]=x   cc=0
t=3 pc=2 a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=1
t=4 pc=3 a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=2
t=5 pc=4 a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=3
t=6 pc=5 a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=4
t=7 pc=6 a=120 a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=5
t=8 pc=7 a=120 a[0]=120 a[1]=121 a[2]=122 ... a[14]=134 a[15]=135 cc=6
t=9 pc=8 a=120 a[0]=120 a[1]=121 a[2]=122 ... a[14]=134 a[15]=135 cc=7
```

In 6 cycles is performed a computation consisting of 29 additions. In the general case, for a number of p cells, the execution time is $3+(1+0.5\times log\,p)=4+0.5\times log\,p$, *where* $1+0.5\times log\,p$ *is the latency of the reduction net.*

Therefore, $2p-1$ *are performed in* $4+0.5\times log\,p$ *cycles by an engine with p cells. The acceleration belongs to*

$$O(\frac{p}{log\,p})$$

which is normal for a computation involving communication.

◇

Example 2.6

◇

2.3 Implementation

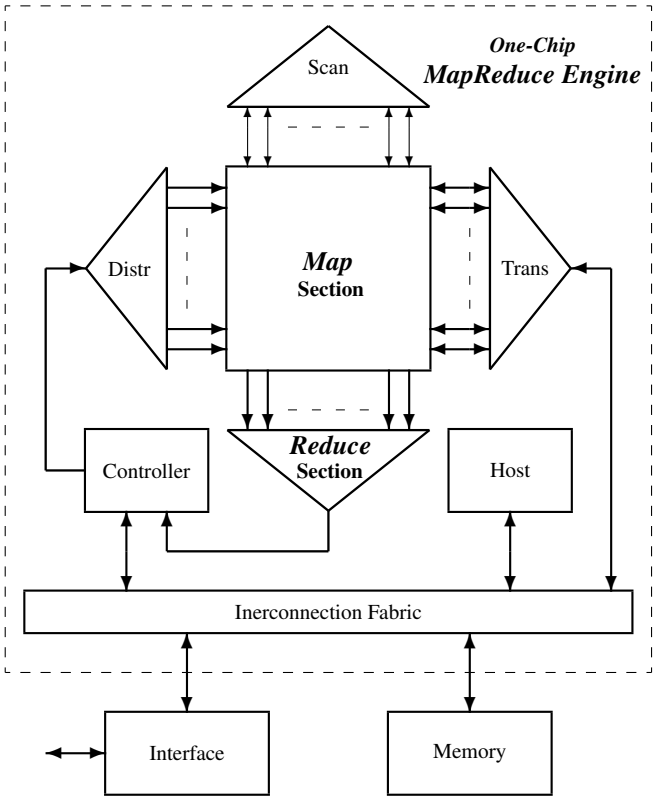


Figure 2.1:

Chapter 3

The Improved Version of pRISC

The generic version of pRISC accelerator is submitted to successive improvements as a consequence of the limitations emphasized in using it to solve various problems. We consider only solutions that are quite general, useful for an enough wide space of applications.

3.1 The serial register

The matrix-vector product is a very frequent operation. Therefore, it deserve a special treatment. The operation suppose a series of scalar (dot, inner) products whose results must be assembled in a vector stored back into the array distributed memory. The generic pRISC MapReduce accelerator performs very efficiently the vector multiplication (in the Map section of the engine) and then the *n*-ary addition (in the Reduce section of the engine). But, composing the resulting vector request the following embarrassingly long sequence of operations (let us call it `ipLoop.v`):

```
00  LB(6);  cSTORE(5);   RILOAD(63); // load next matrix line
01          cNOP;        MULT(0);    // multiply line with vector
02          cLOAD(5);    IXLOAD;     // acc <= ixCounter; acc[i] <= ixVector[i]
03          cNOP;        CSUB;       // acc[i] <= acc[i]-acc;
04          cNOP;        WHEREZERO;  // for reduction latency
05          cNOP;        NOP;        // for reduction latency if P>16
06          cNOP;        NOP;        // for reduction latency if P>64
                                     // ...
07          cCLOAD(0);   NOP;        // acc <= reduceAdd
08          cNOP;        CLOAD;      // acc[i] <= acc;
09          cLOAD(5);    STORE(1);   // acc <= ixCounter; mem[i][1] <= acc[i]
10          cBRNZDEC(6); ENDWHERE;   // test end of loop;
```

Lines 01 to 07 provide in controller's accumulator the scalar product of the vector with a line of the matrix. Because the latency of the reduction network for addition is $1+0.5logP$, the above example is for an array of 256 cells. The minimum length of the loop is of 9 cycles, for $P=16$, because in the lines 02 to 04, in array is selected the cell which will receive the currently computed scalar product. The time for this loop is: $7+0.5logP$.

The execution time for matrix-vector multiplication results in $O(NlogP)$ for $N \leq P$ starting from:

$$T_{matrixMatrixMultiplication} = 0.5N^2log_2N + k_1N^2 + k_2N + k_3 \in O(N^2logN)$$

the improvement we are looking for seeks to reduce the effect of the latency introduced by the reduction operation. Thus, the contribution of the `ipLoop.v` code section, of $7+0.5logP$, to as smallest as possible. We seek for a small constant in order to obtain

$$T_{matrixMatrixMultiplication} \in O(N^2)$$

The solution is to add a serial-parallel two-direction shift register of $(1 << x) \times n$ bits distributed along the array, one n-bit parallel register per cell. The resource list from 2.1 must be filled with a line, as follows:

```
// vectorial resources
...
reg [n-1:0] serialReg[0:(1<<x)-1]           ; // serial register
// scalar resources
...
```

In the instruction set we must add three instructions:

```
pushl  = 5'b01110, // push left op in the global shift register
pushr  = 5'b01111, // push right op in the global shift register
...
srload = 5'b01110, // serial register load in acc[i]<=serialReg[i]
```

The assembly language is enlarged with a subset of instructions associated to machine instructions: `pushl`, `pushr` and `srload`. They are used to load serially, n-bit word by n-bit word, from the left or right side using, and then, to load the serial register into the accumulator registers in each cell. The assembly language is upgraded with the following instructions:

value push left :

```
cVPUSHL: serialReg[i] <= (i==(1<<x-1)) ? cScalar : serialReg[i+1]
```

`mem[cScalar]` **push left** :

```
cPUSHL: serialReg[i] <= (i==(1<<x)-1)) ? mem[cScalar] : serialReg[i+1]
```

`mem[cScalar + addr]` **push left** :

```
cRPUSHL: serialReg[i] <= (i==(1<<x)-1)) ? mem[cScalar+addr] : serialReg[i+1]
```

`mem[cScalar + addr]` **push left & increment** `addr` :

```
cRIPUSHL: serialReg[i] <= (i==(1<<x)-1)) ? mem[cScalar+addr] : serialReg[i+1]
          addr <= cScalar+addr
```

cOp push left :

```
cCPUSHL(0): serialReg[i] <= (i==(1<<x-1)) ? reductionAdd : serialReg[i+1]
cCPUSHL(1): serialReg[i] <= (i==(1<<x-1)) ? reductionMin : serialReg[i+1]
cCPUSHL(2): serialReg[i] <= (i==(1<<x-1)) ? reductionMax : serialReg[i+1]
cCPUSHL(3): serialReg[i] <= (i==(1<<x-1)) ? reductionFlg : serialReg[i+1]
            reductionAdd = acc[0] + ... + acc[1<<x-1]
            reductionMin = min(acc[0], ... , acc[1<<x-1])
            reductionMax = max(acc[0], ... , acc[1<<x-1])
            reductionFlg = boolVect[0] | ... | boolVect[1<<x-1]
```

value push right :

```
cVPUSHR: serialReg[i] <= (i==0) ? cScalar : serialReg[i-1]
```

`mem[cScalar]` **push right** :

```
cPUSHR: serialReg[i] <= (i==0) ? mem[cScalar] : serialReg[i-1]
```

`mem[cScalar + addr]` **push right** :

```
cRPUSHR: serialReg[i] <= (i==0) ? mem[cScalar+addr] : serialReg[i-1]
```

`mem[cScalar + addr]` **push right & increment** `addr` :

```
cRIPUSHR: serialReg[i] <= (i==0) ? mem[cScalar+addr] : serialReg[i-1]
          addr <= cScalar+addr
```

co-operand push right :

```
cCPUSHR(0): serialReg[i] <= (i==0) ? reductionAdd : serialReg[i-1]
cCPUSHR(1): serialReg[i] <= (i==0) ? reductionMin : serialReg[i-1]
cCPUSHR(2): serialReg[i] <= (i==0) ? reductionMax : serialReg[i-1]
cCPUSHR(3): serialReg[i] <= (i==0) ? reductionFlg : serialReg[i-1]
            reductionAdd = acc[0] + ... + acc[1<<x-1]
            reductionMin = min(acc[0], ... , acc[1<<x-1])
            reductionMax = max(acc[0], ... , acc[1<<x-1])
            reductionFlg = boolVect[0] | ... | boolVect[1<<x-1]
```

serial register load :

```
SRLOAD:  acc[i] <= serial Reg[i]
```

Once instantiated in design the shift register is used also for other ways to communicate from controller to array. Besides the co-operand push left/right, words generated inside the controller are organized in a vector sent to the array.

Using this new feature, the previous loop for computing the matrix-vector product, `ipLoop.v`, becomes:

```
LB(6); cCPUSHL(0);   RILOAD(63); // push redSum; load next matrix line
       cBRNZDEC(6);  MULT(0);    // test end; multiply line with vector
```

and for the matrix-vector multiplication it is integrated in the program `matrixVectMult.v` as follows:

```
        cLOAD(0);       IXLOAD;     // acc <= N; acc[i] <= index
        cNOP;           CSUB;       // acc[i] <= index - N
        cLOAD(3);       WHERECARRY; // acc <= M=last line of matrix; select N cells
        cLOAD(1);       CADDRLD;    // acc <= vector adddress; addr[i] <= M
        cLOAD(0);       CALOAD;     // acc <= N = matrix size; acc[i] <= vector[i]
        cVSUB(1);       STORE(0);   // acc <= N-1; mem[i][0] <= vector[i]
        cNOP;           RLOAD(0);   // load next matrix line
        cNOP;           MULT(0);    // multiply line with vector
LB(2);  cCPUSHL(0);     RILOAD(63); // push reduction sum; load next matrix line
        cBRNZDEC(2);    MULT(0);    // test end of loop; multiply line with vector
```

```
                                    // latency = 1 + 0.5 log N
        cNOP;           NOP;
        cNOP;           NOP;
        cLOAD(2);       SRLOAD;     // stop cycle counter; load result in acc
        cNOP;           CSTORE;     // store at D in vector memory
```

The execution time is $T_{vectMatrixMult} = 10 + 2 \times N + 0.5 \times logN \in O(N)$.

Then, the execution time for matrix-matrix multiplication using the previous matrix-vector multiplication becomes:

$$T_{matrixMatrixMultiplication} = (2N + 0.5log_2N + c_1)N + N^2 + c_2 \in O(N^2)$$

Thus, both, the magnitude order and the constant is small, because, for big N, $T_{matrixMatrixMultiplication} \rightarrow 3N^2$.

Example 3.1 *The program that compute in* `vect[22]` *the matrix-vector multiplication for a* 13×13 *matrix, stored in the local memory in vectors* `vect[4], ..., vect[16]` *and a vector stored in* `vect[21]`. *The program has the parameters set by the following sequence:*

```
        cVLOAD(13);     NOP; // set the size of matrix NxN at N=13
        cSTORE(0);      NOP; // mem[0] <= N
        cVLOAD(21);     NOP; // set the address of vector: S=21
        cSTORE(1);      NOP; // mem[1] <= S
        cVLOAD(22);     NOP; // set the address of the result: D=22
        cSTORE(2);      NOP; // mem[2] <= D
        cVLOAD(16);     NOP; // set the address of the last line of matrix: M=16
        cSTORE(3);      NOP; // mem[3] <= M
```

The parameters are stored in the first 4 locations in the scalar memory `mem`. *The location 0 in* `vectMem` *is used as "working regiter".*

Then, the whole program for a matrix of 13×13 *elements is:*

```
        cNOP;           ENDWHERE;   // activate all cells
                                    // SET VECTOR: v16 = 11...1
        cNOP;           VLOAD(1);   // acc[i] = 1;
        cNOP;           STORE(21);  // vectMem[16][i] = acc[i]
                                    // SET MATRIX
                                    // v0  = 00...0
                                    // v1  = 11...1
                                    // ...
                                    // v15 = ff...f
        cVLOAD(16);     VLOAD(3);
        cNOP;           ADDRLD;
        cNOP;           VLOAD(255);
LB(1);  cVSUB(1);       VADD(1);
        cBRNZ(1);       RISTORE(1);

        cVLOAD(13);     NOP;        // set the size of matrix NxN at N=13
        cSTORE(0);      NOP;        // mem[0] <= N
        cVLOAD(21);     NOP;        // set the addres of vector: S=21
        cSTORE(1);      NOP;        // mem[1] <= S
        cVLOAD(22);     NOP;        // set the adress of the result: D=22
```

```
cSTORE(2);      NOP;        // mem[2] <= D
cVLOAD(16);     NOP;        // set the address of the last line: M=16
cSTORE(3);      NOP;        // mem[3] <= M

cSTART;         NOP;        // start cycle counter

`include "matrixVectMult.v"

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

The initial content of vector memory is:

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

The result of the computation modifies the content of the vector memory as follows:

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

```
vect[19] = 15 15 15 15 15 15 15 15 15  15  15  15  15  15 15 15
vect[20] = x  x  x  x  x  x  x  x  x   x   x   x   x   x  x  x
vect[21] = 1  1  1  1  1  1  1  1  1   1   1   1   1   1  1  1
vect[22] = 0  13 26 39 52 65 78 91 104 117 130 143 156 x  x  x
```

The cycles counter provides: `cc = 39`. *Indeed, for* $N = 13$ *the theoretical estimation provides:* $T_{vectMatrixMult} = 39$.

◇

3.2 Tightly closed reduce loop

The assembly instructions associated to the machine instruction `send` provide the co-operand for array, besides `acc` which is the by default co-operand.

The instruction added is:

```
send = 5'b10000, // send op as value to array
```

The assembly language is enlarged with the following instructions:

send `mem[cScalar]` **as co-operand for array** :

```
cSEND: opVect[i] = mem[cScalar]
```

send `mem[cScalar+addr]` **as co-operand for array** :

```
cRSEND: opVect[i] = mem[cScalar+addr]
```

send `mem[cScalar+addr]` **as co-operand for array & increment** `addr` :

```
cRISEND: opVect[i] = mem[cScalar+addr]
         addr <= cScalar+addr
```

send the output of the reduction unit as co-operand for array :

```
cCSEND(0): opVect[i] = reductionAdd
cCSEND(1): opVect[i] = reductionMin
cCSEND(2): opVect[i] = reductionMax
cCSEND(3): opVect[i] = reductionFlg
           reductionAdd = acc[0] + ... + acc[1<<x-1]
           reductionMin = min(acc[0], ... , acc[1<<x-1])
           reductionMax = max(acc[0], ... , acc[1<<x-1])
           reductionFlg = boolVect[0] | ... | boolVect[1<<x-1]
```

Example 3.2 *Let us revisit Example 2.5 using the just added feature. The modified program, which is doing the same thing (adds index with the sum of indexes), is:*

```
// activate all cells
// acc[i} <= i; load index
// acc[i] <= acc[i] + (acc[0]+acc[1]+...acc[15])

cSTART;     ACTIVATE; // ccEnable <= 1; boolVect <= 11...1
```

```
cNOP;       IXLOAD;   // acc[i] <= i
cNOP;       NOP;      // latency step 1
cNOP;       NOP;      // latency step 2
cNOP;       NOP;      // latency step 3
cCSEND(0);  CADD;     // acc[i] <= acc[i] + (acc[0] + acc[1] + ... acc[15])
cSTOP;      NOP;      // stop cycle counter
cHALT;      NOP;
```

The result of simulation is:

```
t=1 pc=0 a=x a[0]=x   a[1]=x   a[2]=x   ... a[14]=x   a[15]=x
t=2 pc=1 a=x a[0]=x   a[1]=x   a[2]=x   ... a[14]=x   a[15]=x
t=3 pc=2 a=x a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15
t=4 pc=3 a=x a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15
t=5 pc=4 a=x a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15
t=6 pc=5 a=x a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15
t=7 pc=6 a=x a[0]=120 a[1]=121 a[2]=122 ... a[14]=134 a[15]=135
t=8 pc=7 a=x a[0]=120 a[1]=121 a[2]=122 ... a[14]=134 a[15]=135
```

The program was shortened by one cycle, the `acc` *is not touched and let free for other purposes. The saved cycle, even it is only one cycle, is important because the many-cell engine avoids a step in which is doing nothing.*

◇

3.3 Activating cells

Activating spatial control instructions are used to activate inactive cells. The instructions added in the instruction set are:

```
actwhere = 5'b00100, // act[i] <= (!boolVect[i]&(acc[i]==acc)) ? 0 : act[i]
saveact  = 5'b00101, // act[i] <= act[i] - 1; save activation
restact  = 5'b00111, // act[i] <= act[i] + 1; restore activation
```

The assembly language is enriched with the following instructions:

activate inactive cells where `accVect[i] == acc` :

```
ACTWHERE: actVect[i] <= (!boolVect[i] && (accVect[i] == acc)) ? 0 : actVect[i]
```

save activation :

```
SAVEACT: actVect[i] <= actVect[i] - 1
```

restore activation :

```
RESTACT: actVect[i] <= actVect[i] + 1
```

3.4 The Resulting Architecture

The instruction set associated to the field `instr[4:0]` is:

```
add      = 5'b00000, // {cr, acc} <= acc + op;
addc     = 5'b00001, // {cr, acc} <= acc + op + cr;
sub      = 5'b00010, // {cr, acc} <= acc - op;
rsub     = 5'b00011, // {cr, acc} <= op - acc;
subc     = 5'b00100, // {cr, acc} <= acc - op - cr;
rsubc    = 5'b00101, // {cr, acc} <= op - acc - cr;
div      = 5'b00110, // acc <= acc/op;
rdiv     = 5'b00111, // acc <= op/acc;
mult     = 5'b01000, // acc <= acc * op;
bwand    = 5'b01001, // acc <= acc & op;
bwor     = 5'b01010, // acc <= acc | op;
bwxor    = 5'b01011, // acc <= acc ^ op;
load     = 5'b01100, // acc <= op;
store    = 5'b01101, // op <= acc;
search   = 5'b01110, // boolVect <= (acc = op) ? 1 : 0;
csearch  = 5'b01111, // boolVect <= (acc = op) & (boolVect >> 1) ? 1 : 0;
insert   = 5'b10000, // insert op ar first
pushl    = 5'b01110, // push left op in the global shift register (NF)
pushr    = 5'b01111, // push right op in the global shift register (NF)
send     = 5'b10000, // send op as value to array (NF)
compare  = 5'b10001, // {cr, acc} <= (acc - op) & (100...0) | {0, acc};

jmp      = 5'b00000, // pc <= pc + scalar;
brz      = 5'b00001, // pc <= acc=0 ? pc + scalar : pc + 1;
brnz     = 5'b00010, // pc <= acc=0 ? pc + 1 : pc + scalar;
brzdec   = 5'b00011, // pc <= acc=0 ? pc + scalar : pc + 1; acc <= acc - 1
brnzdec  = 5'b00100, // pc <= acc=0 ? pc + 1 : pc + scalar; acc <= acc - 1
brcr     = 5'b00101, // pc <= cr ? pc + scalar : pc + 1;
brncr    = 5'b00110, // pc <= cr ? pc + 1 : pc + scalar;

start    = 5'b01110, // start cycle counter
stop     = 5'b01111, // stop cycle counter

where    = 5'b00000, // &boolVect <= (boolVect & cond[scalar[1:0]]) ? 1:0;
wheren   = 5'b00001, // &boolVect <= (boolVect & cond[scalar[1:0]]) ? 0:1;
elsew    = 5'b00010, // boolVect <= ~boolVect;
endwhere = 5'b00011, // boolVect <= 1;
actwhere = 5'b00100, // boolVect <=  (NF)
saveact  = 5'b00101, // act[i] <= act[i] - 1; save activation (NF)
activate = 5'b00110, // actVect[i] = 0 (NF)
restact  = 5'b00111, // act[i] <= act[i] + 1; restore activation (NF)

grotate  = 5'b01000, // global rotate
```

```
glshift  = 5'b01001, // global leftt shift
grshift  = 5'b01010, // global right shift
brshift  = 5'b01011, // Boolean vector right shift; for read from array
delete   = 5'b01100, // delete first
ixload   = 5'b01101, // index load: acc[i] <= i
srload   = 5'b01110, // serial register load  in acc[i] <= serialReg[i](NF)

insval   = 5'b10000, // acc <= {acc[23:0], scalar}
shrightc = 5'b10001, // {cr, acc} <= {acc[0], cr, acc[n-1:1]}
shright  = 5'b10010, // {cr, acc} <= {acc[0], 1'b0, acc[n-1:1]}
sharight = 5'b10011, // {cr, acc} <= {acc[0], acc[n-1], acc[n-1:1]}
penc     = 5'b10100, // {cr, acc} <= {(acc==0), 27'b0, pe(acc)}

vload    = 5'b11110, // vectMem[i][aScalar] <= mem[acc + i*cScalar]
vstore   = 5'b11111; // mem[acc + i*cScalar] <= vectMem[i][aScalar]
```

The added instructions are marked with (NF).

Part II

Applications

Chapter 4

Berkeley's View

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

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

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

while the BC code is:

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

allowing a linear acceleration of the computation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 5

The First Dwarf: Dense Linear Algebra

The problems approached in this chapter are:

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

5.1 Matrix Transpose

5.1.1 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;
================================================================================
```

5.1.2 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]
//=============================================================================================
```

5.1.3 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 5.1 *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.

◇

5.2 Matrix-Vector Multiplication

The algorithm is standard.

5.2.1 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.

5.3 Matrix-Matrix Multiplication

The algorithm is standard.

5.3.1 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
```

5.3.2 The Verification

Example 5.2 *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:* $T_{vectMatrixMult} = 462$.

◇

5.4 Matrix Move

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

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

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

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

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

Chapter 6

The Second Dwarf: Sparse Linear Algebra

Part III

ANNEXES

Appendix A

Kleene's Mathematical Model of Computation

Two theorems about the model.

The Recursive function Definition From [Kleene '36] the following definition for partial recursive functions is extracted:

Definition A.1 *Any partial recursive function* $f : \mathbb{N}^n \to \mathbb{N}$ *can be computed using the* ***initial functions****:*

- $ZERO(x) = 0$*: the variable x takes the value zero*
- $INC(x) = x + 1$*: increments the variable* $x \in \mathbb{N}$
- $SEL(i, x_1, \ldots, x_n) = x_i$*: i selects the value of* x_i *from the sequence* $X =< x_1, \ldots, x_n >$ *(called* identity function *in Kleene's paper)*

and the application of the following ***rules****:*

- Composition:
  $f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_p(x_1, \ldots, x_n))$ *where:* f *is a total function if* $g : \mathbb{N}^p \to \mathbb{N}$ *and* $h_i : \mathbb{N}^n \to \mathbb{N}$*, for* $i = 1, \ldots p$*, are total functions*
- Primitive recursion:
  $f(x_1, \ldots, x_n, y) = g(x_1, \ldots, x_n, f(x_1, \ldots, x_n, y-1))$ *while*
  $f(x_1, \ldots, x_n, 0) = h(x_1, \ldots, x_n)$*, where:* f *is a total function if g and h are total functions (called* ordinary recursion *in Kleene's paper)*
- Minimization (least-search):
  $f(x, y) = \mu y[g(x, y) = 0]$*, i.e., the rule computes the value of function* $f : \mathbb{N}^2 \to \mathbb{N}$ *as the smallest y for which the function* $g(x, y) = 0$*, if any.*

$\diamond$

Preliminary definitions used to prove the two theorems.

Definition A.2 *The* reduction-less composition*, or* map composition*, MC, is the composition:* $f(x_1,\ldots,x_n) =< h_1(x_1,\ldots,x_n),\ldots h_p(x_1,\ldots,x_n) >$ *because:* $g(y_1,\ldots,y_p) =< y_1,\ldots,y_p >$.

◇

Definition A.3 *The* reduction composition*, RC, is the composition:*
$f(y_1,\ldots y_p) = g(y_1,\ldots,y_p)$ *with* $h_i(x_1,\ldots,x_n) = SEL(i,x_1,\ldots,x_n) = x_i$*, for* $i = 1,\ldots,n$ *and* $n = p$.

◇

According to the previous two definitions, the composition rule could be considered as having a ***map-reduce*** structure.

Definition A.4 *Let be the reduction-less composition* $C_i : \mathbb{N}^i \to \mathbb{N}^{i+1}$ *with:* $h_{i+1}(x_1,\ldots,x_i) = P_i(SEL(i,x_1,\ldots,x_i)) = P_i(x_i)$ *while* $h_k(x_1,\ldots,x_i) = SEL(k,x_1,\ldots,x_i) = x_k$ *for* $k = 1,2,\ldots,i$*. The actual form of* C_i *results:*

$$C_i(x_1,\ldots,x_i) =< x_1,x_2,\ldots,x_i,P_i(x_i) >$$

◇

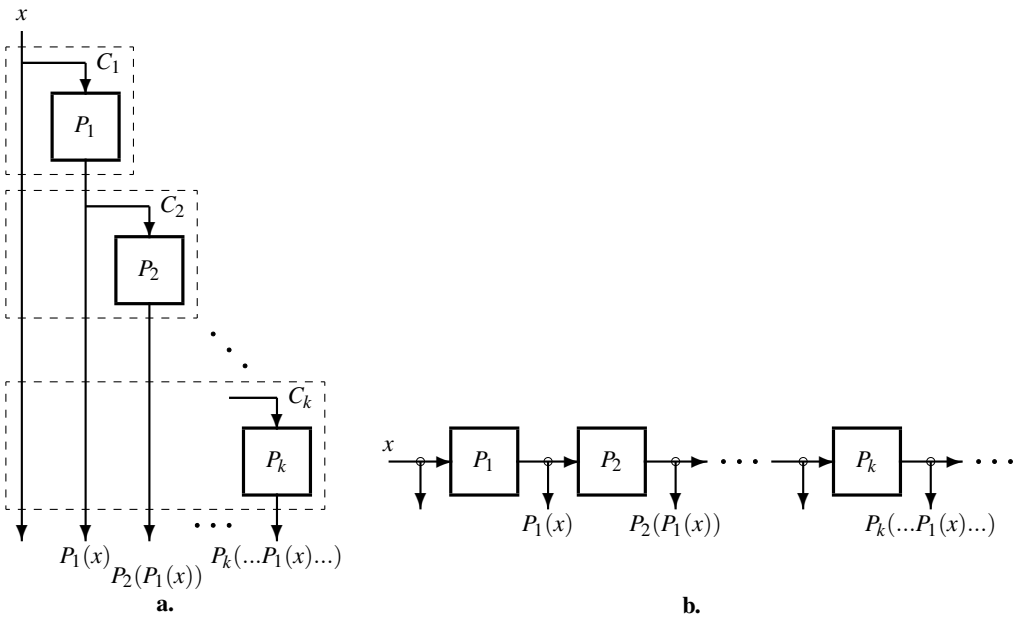


Figure A.1: The multi-output pipeline structure, MOP, as a repeated application of the composition C_i, for $i = 1,2,\ldots,k,\ldots$. **a**. The explicit application of C_i. **b**. The resulting MOP circuit structure.

Definition A.5 *The repeated application of C_i (see Figure A.1a[1]), starting from $i = 1$ with $x_1 = x$ defines the* multi-output pipelined*, MOP, function $MOP : \mathbb{N} \to \mathbb{N}^n$, as*

$$MOP(x) = (x, P_1(x), P_2(P_1(x)), \ldots, P_k(P_{k-1}(\ldots, (P_1(x)\ldots)), \ldots)$$

◇

The function $MOP(x)$ is a total function if the functions P_i are total functions, since it is computed using only the repeated application of the composition C_i. For the theoretical model, n is not limited to a specific value.

Definition A.6 *The function* predicated selection*, $PS : \{\{0,1\} \times \mathbb{N}\}^n \to \mathbb{N}$ is*

$$PS(S) = ADD(MULT(SEL(0,S)), \ldots)$$

where: $S =< (p_0, s_0), (p_1, s_1), \ldots >$ takes a sequence of pairs $(predicate, scalar)$, and returns a scalar. The functions ADD and MULT add and multiply, correspondingly, the components of the sequence received as argument.

◇

MULT is the function *mapped* on the first level of composition, while ADD is the *reduction* function. The function PS is used when one and only one p_i takes the value 1, i.e., for $i = k$ $p_i = 1$ and for $i \neq k$ $p_i = 0$:

$$PS(S) = ADD(\ldots, MULT(0, s_{k-1}), MULT(1, s_k), MULT(0, s_{k+1}), \ldots)$$

$$PS(S) = ADD(0, 0, \ldots, 0, s_k, 0, \ldots) = s_k$$

Definition A.7 *Let be the Boolean sequence $X =< x_0, x_1, \ldots >$. The function $FIRST(X) : \{0,1\}^n \to \{0,1\}^n$ is computed, by definition, using a two stage structure as follows:*

1. *the function $MOP_{OR}(x, X) =< y_0, y_1, \ldots >$, for $x = SEL(0, X)$, computes the* OR prefix *function, $ORPX(X) : \{0,1\}^n \to \{0,1\}^n$, with the following computation performed in each P_i:*

   $$P_i(x, X) =< y_i, X >=< OR(x, SEL(i, X)), X >$$

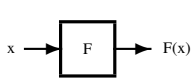
   for $i = 1, 2, \ldots$, which allows to select from its multi-output the sequence of OR prefixes from the binary sequence X:

   $$ORPX(X) = MOP_{OR}(SEL(0, X), X) =< y_0, y_1, \ldots >=$$

   $$< x_0, (x_0 + x_1), (x_0 + x_1 + x_2), \ldots >$$

   where "+" stands for the logic OR function

[1] The graphic representation of functions uses blocks of form



which stand for the computation $F(x)$.

2. *a reduction-less composition with* $h_i(ORPX(X)) = first_i$, *where*

$$first_i = AND(SEL(i, ORPX(X)), NOT(SEL((i-1), ORPX(X))))$$

for $i = 0, 1, \ldots$, *where* $SEL((i-1), ORPX(X)) = 0$ *for* $i = 0$,

such that

$$FIRST(X) =< first_0, first_1, \ldots >$$

is a sequence of Booleans with a single 1, **if any**.

◇

The first theorem about the primitive recursion.

Theorem A.1 *The primitive recursive rule is reducible to repeated applications of specific compositions.*

◇

Proof A.1 *The primitive recursion rule (see Definition A.1) could be applied in its iterative form using the following expression:*

$$f(x,y) = \underbrace{g(x, g(x, g(x, \ldots g}_{y\,times}(x, h(x)) \ldots)))$$

Let be the specific instantiation of the MOP function (see Definition A.5), with the predicate $p_i = (y == i)$, *where the functions* P_i *compute a quintuple, as follows:*

$$P_1 = H(i,x,y) =< INC(i), x, y, f(x,i), p_i >=< z_1, z_2, z_3, z_4, z_5 >$$

where: because, $i = 0$, $f(x,i) = h(x)$

$$P_k = G(i,x,y,f(x,i-1)) =< INC(i), x, y, f(x,i), p_i >=< z_1, z_2, z_3, z_4, z_5 >$$

for $k = 2, 3, \ldots$ *and* $i = 1, 2, \ldots$, *where each* P_k, *function is computed for the quadruple* $< z_1, z_2, z_3, z_4 >$ *generated by* P_{k-1}. *From the outputs of the MOP function we select the sequences of pairs* $< z_4, z_5 >$ *in order to obtain:*

$$S =<< f(x,0), (y == 0) >, < f(x,1), (y == 1) >, \ldots >$$

S is used as input for the function PS which provides the result $f(x,y)$.

◇

Figure A.2 presents the function (see Definition A.6) obtained by composing a specific MOP function with PS function. The two stage computation just described, as a structure indefinitely extensible to the right, is a theoretical model, because the index i takes values no matter how large, similar with the indefinitely extensible ("infinite") tape of Turing's machine. But, it is very important that the algorithmic complexity of the description is in $O(1)$, because the functions H, G, MOP and PS have constant size descriptions.

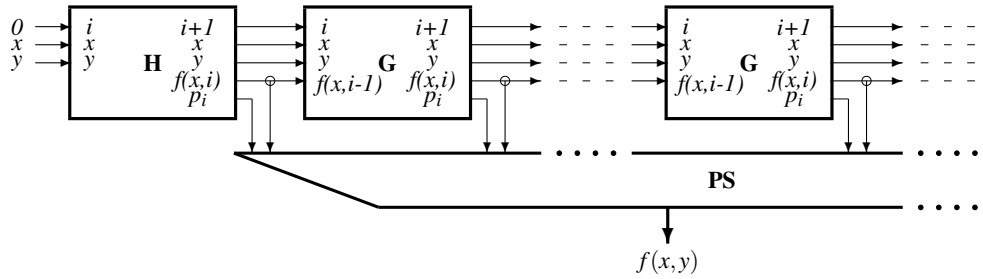


Figure A.2: The specific MOP structure for partial recursive computation.

The second theorem about the minimization rule.

Theorem A.2 *The minimization (least-search) rule is reducible to repeated applications of specific compositions.*

◇

Proof A.2 *The minimization (least-search) rule computes the value of $f(x)$ as the smallest y for which $g(x,y) = 0$. The three steps used in the evaluation of the function are $f(x) = F_3(F_2(F_1(x)))$, three specific compositions, as follows :*

1. *$F_1 : \mathbb{N} \to \{0,1\}^n$, is a "speculative" reduction-less composition, which returns the "infinite" sequence of predicates, X_1:*

$$F_1(x) =< h_0(x), h_1(x), \ldots >=< p_0, p_1, \ldots >= X_1$$

with $h_i(x) = (g(x,i) == 0) = p_i$; in this step the function g is computed for x and "all" positive integers starting with 0.

2. *$F_2 : \{0,1\}^n \to \{0,1\}^n$ is a reduction-less composition with:*

$$F_2(X_1) = FIRST(X_1) = X_2$$

It provides a sequence of predicates with no more than one 1, ***if any****.*

3. *$F_3 : \{0,1\}^n \to \mathbb{N}$, is a composition with:*

$$h_i(X_2) = ADD(SEL(i,X_2), (SEL(i,X_2)\ ?\ i : 0))$$

$$g = ADD(h_0(X_2), h_1(X_2), \ldots) = result$$

If *result* $= 0$, **then** *the function $f(x)$ is not defined for the input variable x,* **else**, *if result* > 0, *the function is defined for x and*

$$f(x) = result - 1$$

It selects as solution, if it exists, the incremented index corresponding to the occurrence of 1 in X_2, ***if any****.*

◇

The computation just described is only a theoretical model, because the index i has an indefinitely large value. But, the size of algorithmic description remains $O(1)$.

Corollary A.1 *Any computation defined in Definition A.1 can be done, according to Theorem A.1 and Theorem A.2, using the initial functions and the repeated application of the composition rule.*

◇

Kleene's approach defines the other two rules, ordinary (primitive) recursion and minimization (least-search), for helping in the taxonomy of the recursive functions. For defining the computation the Corollary A.1 makes the necessary delimitation.

Appendix B

The Behavioral Description of the Engine

The design consists of three `.v` files:

- the first – `00_parameters.v` – contains the parameters which defines the engine
- the second – `01_isa.v` – contains the instruction set architecture
- the last – `01_mapReduceAccelerator.v` – is the behavioral description of the engine with parameters and instruction set previously defined.

The behavioral description of the MapReduce accelerator is used for:

- validating the architecture
- for simulations
- for verification as witness

Parameters are defined in the following file:

```
/*********************************************************************************
PARAMETERS: 00_parameters.v
*********************************************************************************/
    parameter   n = 32  , // word size
                x = 4   , // index size
                v = 6   , // vector memory address size
                s = 8   , // scalar memory address size
                p = 8   , // program memory address size
                c = 8   , // value size in instruction
                a = 5     // size of activation counter
//================================================================================
```

Instruction Set Architeecture (ISA) is defined in the following file:

```
/***********************************************************************
INSTRUCTION SET ARCHITECTURE: 01_isa.v

Instruction's structure for cell/controller:
instr[15:0] = { opCode[15:11]   , // operation code
                operand[10:8]   , // operand select code
                scalar[c-1:0]   } // immediate value or address

For c = 8 the pair issued in each cycle is:
instruction[31:0] = {   arrayOpCode[15:11]  , // operation code
                        arrayOperand[10:8]  , // operand select code
                        arrayScalar[7:0]    , // immediate value or address
                        contrOpCode[15:11]  , // operation code
                        contrOperand[10:8]  , // operand select code
                        contrScalar[7:0]    } // immediate value or address

    op: the operand in Controller
    cScalar: the scalar for Controller
    addr: the address register for data memory in Controller

    aScalar: the scalar for any cell
For: i = 0,1,...,(2^x)-1
    op[i]: the operand in cell i
    addr[i]: the address register for data memory in cell i
    acc[i]: the accumulator of the cell i
    bool[i]: the Boolean variable of the cell i; bool[i]=1 => cell i is active

Version: 1 (November 2015)
Author: Gheorghe M. Stefan
***********************************************************************/
parameter         //  selects the right operand
    val = 3'b000, //  immediate value:
                  //    op    = cS = {24{cScalar[c-1]}}, cScalar}
                  //    op[i] = aS = {24{arrayScalar[c-1]}}, arrayScalar}
    mab = 3'b001, //  absolute data memory:
                  //    op    = mem[cS[s/v-1:0]]
                  //    op[i] = vectMem[aS[s/v-1:0]]
    mrl = 3'b010, //  relative in data memory:
                  //    op    = mem[addr + cS[s-1:0]]
                  //    op[i] = vectMem[addr + aS[v-1:0]]
    mri = 3'b011, //  relative and increment addr with scalar:
                  //    op       = mem[addr + cS[s-1:0]];
                  //    addr    <= addr + cS[s-1:0]
                  //    op[i]    = vectMem[addr + aS[v-1:0]];
```

```
                //    addr[i] <= addr + aS[v-1:0]
    cop = 3'b100, //  co-operand:
                //    op = case(cS[1:0])
                //         2'b00: reduction add = add(acc[0],acc[1],...)
                //         2'b01: reduction min = min(acc[0],acc[1],...)
                //         2'b10: reduction max = max(acc[0],acc[1],...)
                //         2'b11: reduction flag = = or(bool[0],bool[1],...)
                //    op[i] = (contrOpCode==send) ? op : acc
    mac = 3'b101, //$ op[i] = mem[(contrOpCode==send) ? op : acc]
    mrc = 3'b110, //$ op[i] = mem[addr + ((contrOpCode==send) ? op : acc)]
    ctl = 3'b111; //  control instructions

parameter             // conditions tested by the instructions where, wheren
    zero    = 2'b00, // accVect[i] == 0
    carry   = 2'b01, // crVect[i] == 1
    first   = 2'b10, // firstVect[i] == 1
    next    = 2'b11; // nextVect[i] == 1

parameter
    // operand = 000, ..., 110
    // #: for controller only
    // $: for array only
    add         = 5'b00000, //  {cr, acc} <= acc + op;
    addc        = 5'b00001, //  {cr, acc} <= acc + op + cr;
    sub         = 5'b00010, //  {cr, acc} <= acc - op;
    rsub        = 5'b00011, //  {cr, acc} <= op - acc;
    subc        = 5'b00100, //  {cr, acc} <= acc - op - cr;
    rsubc       = 5'b00101, //  {cr, acc} <= op - acc - cr;
    div         = 5'b00110, //  acc <= acc/op;
    rdiv        = 5'b00111, //  acc <= op/acc;
    mult        = 5'b01000, //  acc <= acc * op;
    bwand       = 5'b01001, //  acc <= acc & op;
    bwor        = 5'b01010, //  acc <= acc | op;
    bwxor       = 5'b01011, //  acc <= acc ^ op;
    load        = 5'b01100, //  acc <= op;
    store       = 5'b01101, //  op <= acc;
    search      = 5'b01110, //$ boolVect <= (acc=op) ? 1 : 0;
    csearch     = 5'b01111, //$ boolVect <= (acc=op) & (boolVect >> 1) ? 1 : 0;
    insert      = 5'b10000, //$ insert op ar first
    pushl       = 5'b01110, //# push left op in the global shift register (NF)
    pushr       = 5'b01111, //# push right op in the global shift register (NF)
    send        = 5'b10000, //# send op as coOperand to array (NF2)
    compare     = 5'b10001, // cr <= (acc - op)[n]
    fadd        = 5'b10010, // float add

    // operand = 110; ctl
```

```
    jmp         = 5'b00000, //# pc <= pc + scalar;
    brz         = 5'b00001, //# pc <= acc=0 ? pc + scalar : pc + 1;
    brnz        = 5'b00010, //# pc <= acc=0 ? pc + 1 : pc + scalar;
    brzdec      = 5'b00011, //# pc <= acc=0 ? pc + scalar : pc+1; acc <= acc-1
    brnzdec     = 5'b00100, //# pc <= acc=0 ? pc+1 : pc + scalar; acc <= acc-1
    brcr        = 5'b00101, //# pc <= cr ? pc + scalar : pc + 1;
    brncr       = 5'b00110, //# pc <= cr ? pc + 1 : pc + scalar;
    //          = 5'b00111, //#
    restart     = 5'b01000, //#&restart a new program in controller
    mainprog    = 5'b01001, //#&load controller's program
    allcells    = 5'b01010, //#&load the same program in all cells
    cells       = 5'b01011, //#&load program in cells[scalar[...]]
    endload     = 5'b01100, //#&end loading the current program
    //          = 5'b01101, //#
    start       = 5'b01110, //# start cycle counter
    stop        = 5'b01111, //# stop cycle counter

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

    insval      = 5'b10000, // acc <= {acc[23:0], scalar}
    shrightc    = 5'b10001, // {cr, acc} <= {acc[0], cr, acc[n-1:1]}
    shright     = 5'b10010, // {cr, acc} <= {acc[0], 1'b0, acc[n-1:1]}
    sharight    = 5'b10011, // {cr, acc} <= {acc[0], acc[n-1], acc[n-1:1]}
    penc        = 5'b10100, // {cr, acc} <= {(acc==0), 27'b0, pe(acc)}
    vload       = 5'b11110, // vectMem[i][arrayScalar]<=mem[acc+i*contrScalar]
    vstore      = 5'b11111; // mem[acc+i*contrScalar]<=vectMem[i][arrayScalar]

    // #: controller only
    // $: array only
    // &: TBD
//=============================================================================
```

The behavioral description of MapReduce Accelerator is defined in the following file:

```
/*******************************************************************************
MAP-REDUCE ACCELERATOR: 01_mapReduceAccelerator.v

The behavioral description of the MapReduce accelerator:
- scalar seection
- vectorial section
Architecture: accumulator based

Version: 1 (November 2015)
Author: Gheorghe M. Stefan
*******************************************************************************/
module mapReduceAccelerator #(`include "00_parameters.v")
        (   output  reg [n-1:0] counter ,         // for performance evaluation
            input               reset   ,
            input               cycle   );
    // vectorial resources
    reg [x-1:0] ixVect[0:(1<<x)-1]              ; // index vector
    reg [a-1:0] actVect[0:(1<<x)-1]             ; // activation vector
    reg         boolVect[0:(1<<x)-1]            ; // Boolean vector
    reg [n-1:0] accVect[0:(1<<x)-1]             ; // accumulator vector
    reg         crVect[0:(1<<x)-1]              ; // carry vector
    reg [v-1:0] addrVect[0:(1<<x)-1]            ; // address vector
    reg [n-1:0] vectMem[0:(1<<x)-1][0:(1<<v)-1] ; // vector memory
    reg [n-1:0] serialReg[0:(1<<x)-1]           ; // serial register
    // scalar resources
    reg [n-1:0] acc                             ; // scalar accumulator
    reg         cr                              ; // scalar carry
    reg [s-1:0] addr                            ; // scalar address
    reg [n-1:0] mem[0:(1<<s)-1]                 ; // scalar memory
    // scalar float resources
    reg [7:0]   exp                             ; // exponent register
    reg [7:0]   modDeltaExp                     ; // |exp(a) - exp(b)|
    reg         sgnDeltaExp                     ; // sgn(exp(a) - exp(b))
    // control resources
    reg [p-1:0] pc                              ; // program counter
    reg [31:0]  ir                              ; // instruction register
    reg [31:0]  progMem[0:(1<<p)-1]             ; // program memory
    // evaluation resources
    reg [31:0]  cc                              ; // cycle counter
    reg         ccEnable                        ; // cycle counter enable

    // variables
    reg [p-1:0] nextPc                          ; // next program counter
    reg [15:0]  inc                             ; // clock cycles/instruction
    reg [n-1:0] op                              ; // the right operand for ALU
    reg [n-1:0] delayedOp                       ; // cooperand for controller
    reg [n-1:0] opVect[0:(1<<x)-1]              ; // operand vector
    reg         pxVect[0:(1<<x)-1]              ; // prefix vector
    reg         firstVect[0:(1<<x)-1]           ; // first vector
    reg         nextVect[0:(1<<x)-1]            ; // next to the first vector
```

```
reg [3:0]   condVect[0:(1<<x)-1]            ; // condition vector
reg [n-1:0] ob[0:(1<<x)-1]                  ; // accVect | ~{n{boolVect}}

integer     i, j, k, l, m                   ;

`include "01_isa.v"

wire    [4:0]   contrOpCode         ;
wire    [2:0]   contrOperand        ;
wire    [7:0]   contrScalar         ;
wire    [4:0]   arrayOpCode         ;
wire    [2:0]   arrayOperand        ;
wire    [7:0]   arrayScalar         ;
reg     [n-1:0] reductionAdd        ;
reg     [n-1:0] reductionMin        ;
reg     [n-1:0] reductionMax        ;
reg     [n-1:0] reductionFlg        ;
reg     [n-1:0] delayedRedAdd[x/2:0]; // delayed reduction output
reg     [n-1:0] delayedRedMin[x/2:0]; // delayed reduction output
reg     [n-1:0] delayedRedMax[x/2:0]; // delayed reduction output
reg     [n-1:0] delayedRedFlg[x/2:0]; // delayed reduction output
reg     [15:0]  delayedCinstr[x/2:0]; // delayed shift

assign contrOpCode  = ir[15:11] ; // operation code for controller
assign contrOperand = ir[10:8]  ; // right selection operand for controller
assign contrScalar  = ir[7:0]   ; // immediate value for controller
assign arrayOpCode  = ir[31:27] ; // operation code for the array of cells
assign arrayOperand = ir[26:24] ; // right selection operand for array
assign arrayScalar  = ir[23:16] ; // immediate value for array
// CONTROLLER
// control section
always @(posedge cycle) if (reset)  begin   pc  <= {p{1'b1}}                ;
                                            ir  <= 0                        ;
                                            cc  <= 31'b0                    ;
                                    end
                        else    begin   pc  <= nextPc                       ;
                                        ir  <= progMem[nextPc]              ;
                                        if (ccEnable) cc <= cc + 1'b1       ;
                                    end
always @(*) if (contrOperand == ctl)
 case(contrOpCode)
    jmp     : nextPc = pc + contrScalar[p-1:0]                              ;
    brz     : nextPc = (acc == 0) ? (pc + contrScalar[p-1:0]) : (pc + 1'b1) ;
    brnz    : nextPc = (acc == 0) ? (pc + 1'b1) : (pc + contrScalar[p-1:0]) ;
    brzdec  : nextPc = (acc == 0) ? (pc + contrScalar[p-1:0]) : (pc + 1'b1) ;
    brnzdec : nextPc = (acc == 0) ? (pc + 1'b1) : (pc + contrScalar[p-1:0]) ;
    brcr    : nextPc = cr ? (pc + contrScalar[p-1:0]) : (pc + 1'b1)         ;
    brncr   : nextPc = cr ? (pc + 1'b1) : (pc + contrScalar[p-1:0])         ;
            // float instrucions
    fadd    : nextPc = (((op[30:23] - acc[30:23]) > 8'b10111) ||
                        (( ~(op[30:23] - acc[30:23]) + 1) > 8'b10111)) ?
                            (pc + contrScalar[p-1:0]) : (pc + 1'b1)         ;
```

```
    default   nextPc = pc + 1'b1                                                ;
  endcase
 else       if (contrOperand == mab) // for float instrucions
  case(contrOpCode)
    fadd    : nextPc = (((op[30:23] - acc[30:23]) > 8'b10111) ||
                        (( ~(op[30:23] - acc[30:23]) + 1) > 8'b10111)) ?
                            (pc + contrScalar[p-1:0]) : (pc + 1'b1)          ;
    default   nextPc = pc + 1'b1                                             ;
  endcase
 else         nextPc = pc + 1'b1                                             ;

// data section
always @(*)
 case(contrOperand) // selects the right operand for controller
    mab:    op = mem[contrScalar[s-1:0]]                ;
    mrl:    op = mem[addr + contrScalar[s-1:0]]         ;
    cop:    case(contrScalar[1:0])
                2'b00: op = delayedRedAdd[x/2]  ; // redAdd
                2'b01: op = delayedRedMin[x/2]  ; // redMin
                2'b10: op = delayedRedMax[x/2]  ; // redMax
                2'b11: op = delayedRedFlg[x/2]  ; // redFlg
            endcase
    default op = {{(n-8){contrScalar[7]}}, contrScalar} ; // val
 endcase

always @(posedge cycle) begin
 if (!(contrOperand == ctl))
    case(contrOpCode)
        add     : {cr, acc} <= acc + op                                  ;
        addc    : {cr, acc} <= acc + op + cr                             ;
        sub     : {cr, acc} <= acc - op                                  ;
        rsub    : {cr, acc} <= op - acc                                  ;
        subc    : {cr, acc} <= acc - op - cr                             ;
        rsubc   : {cr, acc} <= op - acc - cr                             ;
        div     : {cr, acc} <= {cr, acc/op}                              ;
        rdiv    : {cr, acc} <= {cr, op/acc}                              ;
        mult    : {cr, acc} <= {cr, acc * op}                            ;
        load    : {cr, acc} <= {cr, op}                                  ;
        store   : case(contrOperand)
                    val : addr                            <= acc[s-1:0]  ;
                    mab : mem[contrScalar[s-1:0]]         <= acc         ;
                    mrl : mem[contrScalar[s-1:0] + addr]  <= acc         ;
                  endcase
        bwand   : {cr, acc} <= {cr, acc & op}                            ;
        bwor    : {cr, acc} <= {cr, acc | op}                            ;
        bwxor   : {cr, acc} <= {cr, acc ^ op}                            ;
        compare : {cr, acc} <= (acc - op)&{1'b1, {n{1'b0}}}|{1'b0, acc}  ;
        fadd    : begin
                    exp <= (acc[30:23] < op[30:23]) ?   op[30:23] :
                                                        acc[30:23]       ;
                    modDeltaExp <= (acc[30:23] < op[30:23]) ?
                                        op[30:23] - acc[30:23] :
```

```
                                        acc[30:23] - op[30:23]          ;
                    sgnDeltaExp <= (acc[30:23] < op[30:23])             ;
                    mem[2] <= {op[31], 8'b1, op[22:0]}                  ;
                  end
    endcase
 if (contrOperand == ctl)
    case(contrOpCode)
        shrightc: {cr, acc} <= {acc[0], cr, acc[n-1:1]}         ;
        shright : {cr, acc} <= {acc[0], 1'b0, acc[n-1:1]}       ;
        sharight: {cr, acc} <= {acc[0], acc[n-1], acc[n-1:1]}   ;
        insval  : acc <= (n == c) ? contrScalar :
                                 {acc[(n-c-1):0], contrScalar}  ;
        brzdec  : acc <= acc - 1'b1                             ;
        brnzdec : acc <= acc - 1'b1                             ;
        start   : ccEnable  <= 1'b1                             ;
        stop    : ccEnable  <= 1'b0                             ;
    endcase
end
// ARRAY
always @(*) for (k=0; k<(1<<x); k=k+1)
 begin
    case(arrayOperand) // selcts the right operand in each cell
        mab: opVect[k] = vectMem[k][arrayScalar[v-1:0]]                     ;
        mrl: opVect[k] = vectMem[k][addrVect[k] + arrayScalar[v-1:0]]       ;
        mri: opVect[k] = vectMem[k][addrVect[k] + arrayScalar[v-1:0]]       ;
        cop: opVect[k] = (contrOpCode == send) ? op : acc                   ;
        mac: opVect[k] =
                vectMem[k][(contrOpCode == send) ? op[v-1:0] : acc[v-1:0]]  ;
        mrc: opVect[k] =
                vectMem[k][addrVect[k] +
                        ((contrOpCode == send) ? op[v-1:0] : acc[v-1:0])]   ;
     default opVect[k] = {{(n-c){arrayScalar[7]}}, arrayScalar}             ;
    endcase
    pxVect[k]   = (k == 0) ? boolVect[0] : (boolVect[k] | pxVect[k-1])      ;
    firstVect[k] = (k == 0) ?   pxVect[0] : (pxVect[k] & ~pxVect[k-1])      ;
    nextVect[k] = (k == 0) ?    1'b0 : pxVect[k-1]                          ;
    condVect[k] = {nextVect[k], firstVect[k], crVect[k], (accVect[k] == 0)} ;
    boolVect[k] = (actVect[k] == 0) || (actVect[k][a-1] == 1)               ;
 end

always @(posedge cycle)
for (i=0; i<(1<<x); i=i+1) begin
 if (boolVect[i]) begin
  if (arrayOperand == mri) addrVect[i] <= addrVect[i] + arrayScalar[v-1:0]  ;
  if (!(arrayOperand == ctl))
  case(arrayOpCode)
    add     : {crVect[i], accVect[i]} <= accVect[i] + opVect[i]             ;
    addc    : {crVect[i], accVect[i]} <= accVect[i] + opVect[i] + crVect[i] ;
    sub     : {crVect[i], accVect[i]} <= accVect[i] - opVect[i]             ;
    rsub    : {crVect[i], accVect[i]} <= opVect[i] - accVect[i]             ;
    subc    : {crVect[i], accVect[i]} <= accVect[i] - opVect[i] - crVect[i] ;
    rsubc   : {crVect[i], accVect[i]} <= opVect[i] - accVect[i] - crVect[i] ;
```

```
   div     : {crVect[i], accVect[i]} <= {crVect[i], accVect[i]/opVect[i]}  ;
   rdiv    : {crVect[i], accVect[i]} <= {crVect[i], opVect[i]/accVect[i]}  ;
   mult    : {crVect[i], accVect[i]} <= {crVect[i], accVect[i] * opVect[i]};
   bwand   : {crVect[i], accVect[i]} <= {crVect[i], accVect[i] & opVect[i]};
   bwor    : {crVect[i], accVect[i]} <= {crVect[i], accVect[i] | opVect[i]};
   bwxor   : {crVect[i], accVect[i]} <= {crVect[i], accVect[i] ^ opVect[i]};
   compare : {crVect[i], accVect[i]} <= (accVect[i] - opVect[i])&
                                        {1'b1, {n{1'b0}}}|{1'b0, accVect[i]};
   load    : {crVect[i], accVect[i]} <= {crVect[i], opVect[i]}             ;
   store   : case(arrayOperand)
               val : addrVect[i] <= accVect[i][v-1:0]                      ;
               mab : vectMem[i][arrayScalar[v-1:0]] <= accVect[i]          ;
               mrl : vectMem[i][arrayScalar[v-1:0] + addrVect[i]]
                                  <= accVect[i]                            ;
               mri : vectMem[i][arrayScalar[v-1:0] + addrVect[i]]
                                  <= accVect[i]                            ;
               cop : addrVect[i] <= (contrOpCode == send) ? op[v-1:0] :
                                                              acc[v-1:0]   ;
               mac : vectMem[i][acc[v-1:0]] <= accVect[i]                  ;
               mrc : vectMem[i][acc[v-1:0] + addrVect[i]] <= accVect[i]    ;
             endcase
   search  : boolVect[i] <= (accVect[i] == opVect[i]) ? 1'b1 : 1'b0        ;
   csearch : boolVect[i] <= (i == 0) ? 1'b0 :
               (((accVect[i] == opVect[i]) & boolVect[i-1]) ? 1'b1 :
               1'b0)                                                      ;
   insert  : if (firstVect[i]) accVect[i] <= opVect[i]                    ;
                else if (nextVect[i]) accVect[i] <= accVect[i-1]          ;
 endcase
if (arrayOperand == ctl)
   case(arrayOpCode)
       delete  : if (firstVect[i] | nextVect[i])
                 accVect[i] <= (i == ((1<<x)-1)) ? 0 : accVect[i+1]        ;
       shrightc: {crVect[i], accVect[i]} <=
                   {accVect[i][0], crVect[i], accVect[i][n-1:1]}           ;
       shright : {crVect[i], accVect[i]} <=
                   {accVect[i][0], 1'b0, accVect[i][n-1:1]}                ;
       sharight: {crVect[i], accVect[i]} <=
                   {accVect[i][0], accVect[i][n-1], accVect[i][n-1:1]}     ;
       grotate : accVect[i] <= accVect[(i+1)%(1<<x)]                       ;
       glshift : if (i < (1<<x)-1) accVect[i] <= accVect[i+1]              ;
                   else accVect[i] <= 0                                    ;
       grshift : if (i==0) accVect[i] <= 0                                 ;
                   else accVect[i] <= accVect[i-1]                         ;
       brshift : boolVect[i] <= (i == 0) ? 1'b0 : boolVect[i-1]            ;
       ixload  : accVect[i] <= i                                           ;
       srload  : accVect[i] <= serialReg[i]                                ;
       insval  : accVect[i] <= (n == c) ? arrayScalar :
                                        {accVect[i][(n-c-1):0], arrayScalar};
       endcase
end
if (arrayOperand == ctl)
   case(arrayOpCode)
```

```
     where   : actVect[i] <= (boolVect[i] && (condVect[i][arrayScalar[1:0]])) ?
                         actVect[i] : actVect[i] + 1'b1                      ;
     wheren  : actVect[i] <= (boolVect[i] && !(condVect[i][arrayScalar[1:0]])) ?
                         actVect[i] : actVect[i] + 1'b1                      ;
     elsew   : actVect[i] <= (actVect[i] == 0) ? 1 : ((actVect[i] == 1) ?
                         0 : actVect[i])                                     ;
     endwhere: actVect[i] <= (actVect[i] == 0) ?
                         actVect[i] : (actVect[i] - 1'b1)                    ;
     activate: actVect[i] <= 0                                               ;
     actwhere: actVect[i] <= (!boolVect[i] && (accVect[i] == acc)) ?
                         0 : actVect[i]                                      ;
     saveact : actVect[i] <= actVect[i] - 1                                  ;
     restact : actVect[i] <= actVect[i] + 1                                  ;
    endcase
 if ((delayedCinstr[x/2][10:8] == cop) && (delayedCinstr[x/2][15:11] == pushl))
    if (i == 0)             serialReg[i] <= delayedOp       ;
        else                serialReg[i] <= serialReg[i-1]  ;
 if ((delayedCinstr[x/2][10:8] == cop) && (delayedCinstr[x/2][15:11] == pushr))
    if (i == ((1<<x) - 1))  serialReg[i] <= delayedOp       ;
        else                serialReg[i] <= serialReg[i+1]  ;
 if (((contrOperand == val) || (contrOperand == mab) || (contrOperand == mrl) ||
      (contrOperand == mri)) && (contrOpCode == pushl))
    if (i == 0)             serialReg[i] <= op      ;
        else                serialReg[i] <= serialReg[i-1]  ;
 if (((contrOperand == val) || (contrOperand == mab) || (contrOperand == mrl) ||
      (contrOperand == mri)) && (contrOpCode == pushr))
    if (i == ((1<<x) - 1))  serialReg[i] <= op      ;
        else                serialReg[i] <= serialReg[i+1]  ;
end

// reduction section
always @(*) begin
    reductionAdd = boolVect[0] ? accVect[0] : 0                                ;
    reductionMin = boolVect[0] ? accVect[0] : {n{1'b1}}                        ;
    reductionMax = accVect[0]                                                  ;
    reductionFlg = {{(n-1){1'b0}}, boolVect[0]}                                ;
    for (j=1; j<(1<<x); j=j+1) begin
        reductionAdd = reductionAdd + (boolVect[j] ? accVect[j] : 0)           ;
        ob[j]        = boolVect[j] ? accVect[j] : {n{1'b1}}                    ;
        reductionMin = (reductionMin > ob[j]) ? ob[j] : reductionMin           ;
        reductionMax = (reductionMax < accVect[j]) ? accVect[j] : reductionMax ;
        reductionFlg = {{(n-1){1'b0}}, reductionFlg[0] | boolVect[j]}          ;
    end
    case(delayedCinstr[x/2][1:0])
        2'b00: delayedOp = delayedRedAdd[x/2]   ; // redAdd
        2'b01: delayedOp = delayedRedMin[x/2]   ; // redMin
        2'b10: delayedOp = delayedRedMax[x/2]   ; // redMax
        2'b11: delayedOp = delayedRedFlg[x/2]   ; // redFlg
    endcase
end
always @(posedge cycle) begin
    delayedRedAdd[0] <= reductionAdd                          ;
```

```
        delayedRedMin[0] <= reductionMin                          ;
        delayedRedMax[0] <= reductionMax                          ;
        delayedRedFlg[0] <= reductionFlg                          ;
        delayedCinstr[0] <= {contrOpCode, contrOperand, contrScalar};
        for (m=1; m<(x/2+1); m=m+1) begin
            delayedRedAdd[m] <= delayedRedAdd[m-1]  ;
            delayedRedMin[m] <= delayedRedMin[m-1]  ;
            delayedRedMax[m] <= delayedRedMax[m-1]  ;
            delayedRedFlg[m] <= delayedRedFlg[m-1]  ;
            delayedCinstr[m] <= delayedCinstr[m-1]  ;
        end
    end
    // end reduction section

    // CONTROLLER & ARRAY
    always @(posedge cycle) if ((arrayOperand == ctl) && (contrOperand == ctl))
        for (l=0; l<(1<<x); l=l+1)
            case(contrOpCode)
                vload   : vectMem[l][arrayScalar] <= mem[acc + l*contrScalar]   ;
                vstore  : mem[acc + l*contrScalar] <= vectMem[l][arrayScalar]   ;
            endcase
endmodule
//================================================================================
```

Appendix C

The Simulator

The steps for using the simulator:

- instal the simulator using the files:
  - `01_mapReduceAcceleratorSimulator.v`
  - `01_mapReduceAccelerator.v`
  - in the context of the files:
    - `00_parameters.v`
    - `01_isa.v`
    - `codeGenerator.v`
    - `theProgram.v`
    - `03_examples.v`
    - `cgADD.v`
    - `cgADDC.v`
    - `... .v`
- write the program (for example: `program.v`), include uncommented its name in `theProgram.v` as follows:

```
//‘include "03_matrixVectMult.v"
//‘include "03_matrixMatrixMult.v"
//‘include "03_matrixTranspose.v"
//‘include "03_matrixLoad.v"
//‘include "03_examples.v"
‘include "program.v"
```

- run simulation

The simulator is defined in the following file:

```
/*******************************************************************************
MAP-REDUCE SIMULATOR: 01_mapReduceAcceleratorSimulator.v

Version: 1 (May 2015)
Author: Gheorghe M. Stefan
*******************************************************************************/

module mapReduceAcceleratorSimulator #(`include "00_parameters.v");
    reg             reset, cycle;
    wire    [n-1:0] counter     ;
    integer         i, j, k     ;

    // the clock signal for the system
    initial begin               cycle = 0       ;
                    forever #1  cycle = ~cycle  ;
            end

    // the assembly process
    `include "codeGenerator.v"

    // display the resulting binary code
    initial
        for (i=0; i<64; i=i+1) $display("progMem[%0d] \t = %b", i, dut.progMem[i]);

    // the simulation
    initial begin           reset = 1   ;
                    #2      reset = 0   ;
                    #5000   $stop       ;
            end

    // display the content of the vector memory
    initial begin #5000 for (j=0; j<64; j=j+1)
     $display("vect[%0d] \t = %0d \t %0d ... // (...) for editorial purpose
      j,
      dut.vectMem[0][j], dut.vectMem[1][j], dut.vectMem[2][j], dut.vectMem[3][j],
      dut.vectMem[4][j], dut.vectMem[5][j], dut.vectMem[6][j], dut.vectMem[7][j],
      dut.vectMem[8][j], dut.vectMem[9][j], dut.vectMem[10][j], dut.vectMem[11][j],
      dut.vectMem[12][j], dut.vectMem[13][j], dut.vectMem[14][j], dut.vectMem[15][j]);
    end

    mapReduceAccelerator dut(counter, reset, cycle);

    // monitor the process
    initial begin
        $monitor("t=%0d \t pc=%d \t a=%0d  ...// (...) for editorial purpose
            $time,
            dut.pc,
            dut.acc,
            dut.op,
            dut.addrVect[0], dut.accVect[0], dut.accVect[1], dut.accVect[2],
```

```
            dut.accVect[14], dut.accVect[15],
            dut.boolVect[0], dut.boolVect[1], dut.boolVect[2], dut.boolVect[3],
            dut.boolVect[4], dut.boolVect[5], dut.boolVect[6], dut.boolVect[7],
            dut.boolVect[8], dut.boolVect[9], dut.boolVect[10], dut.boolVect[11],
            dut.boolVect[12], dut.boolVect[13], dut.boolVect[14], dut.boolVect[15],
            dut.cc);
    end
endmodule
//=============================================================================================
```

The code generator used by the simulator is defined in the following file:

```
/*********************************************************************************
CODE GENERATOR: codeGenerator.v

Two-pas code generator which provides the binary form of the program

Version: 1
Author: Gheorghe M. Stefan
*********************************************************************************/

    reg [4:0]   aOpCode             ;
    reg [2:0]   aOperand            ;
    reg [7:0]   aScalar             ;
    reg [4:0]   cOpCode             ;
    reg [2:0]   cOperand            ;
    reg [c-1:0] cScalar             ;
    reg [p-1:0] addrCounter         ;
    reg [p-1:0] labelTab[0:(1<<p)-1];

    `include "01_isa.v"

    // line generator in memory at addrCounter
    task endLine;
        begin
            dut.progMem[addrCounter] = {aOpCode ,
                                        aOperand,
                                        aScalar ,
                                        cOpCode ,
                                        cOperand,
                                        cScalar }   ;
            addrCounter = addrCounter + 1           ;
        end
    endtask

    // sets labelTab in the first pass associating 'counter' with 'labelIndex'
    task LB ;
        input [4:0] labelIndex;

        labelTab[labelIndex] = addrCounter;
    endtask

    // uses the content of labelTab in the second pass
    task ULB;
        input [4:0] labelIndex;

        cScalar = labelTab[labelIndex] - addrCounter;
    endtask

    // generates the binary form for instruction NOP in array
    task NOP;
        begin   aOpCode     = add       ;
```

```
                aOperand    = val       ;
                aScalar     = {c{1'b0}} ;
                endLine                 ;
        end
    endtask

    // generates the binary form for instruction NOP in controller
    task cNOP;
        begin   cOpCode     = add       ;
                cOperand    = val       ;
                cScalar     = {c{1'b0}} ;
        end
    endtask

    // generates the binary form for all instructions
    `include "cgCONTROL.v"      // control instructions for controller
    `include "cgADD.v"          // addition
    `include "cgADDC.v"         // addition with carry
    `include "cgSUB.v"          // subtract
    `include "cgSUBC.v"         // subtract with carry
    `include "cgRVSUB.v"        // reverse subtract
    `include "cgRVSUBC.v"       // reverse subtract with carry
    `include "cgMULT.v"         // multiplication
    `include "cgDIV.v"          // division
    `include "cgRDIV.v"         // reverse division
    `include "cgSHIFT.v"        // shift
    `include "cgLOAD.v"         // load accumulator
    `include "cgSEND.v"         //
    `include "cgSTORE.v"        // store accumulator
    `include "cgAND.v"          // bit-wise AND
    `include "cgOR.v"           // bit-wise OR
    `include "cgXOR.v"          // bit-wise XOR
    `include "cgCOMPARE.v"      // set carrry generated by subtract
    `include "cgARRAYcONTR.v"   // array control instructions
    `include "cgGLOBAL.v"       // global operations

    // RUNNING
    initial begin   addrCounter = 0;
                    `include "theProgram.v"; // first pass
                    addrCounter = 0;
                    `include "theProgram.v"; // second pass
            end
//================================================================================
```

Binary form generator for the instruction ADD is defined in the following file:

```
/*****************************************************************************
BINARY GENERATOR FOR THE GROUP OF INSTRUCTIONS "ADD": cgADD.v


    val = 3'b000, //  immediate value: {(n-8){scalar[7]}}, scalar}
                  //  in `store` instruction selects addr as destination
    mab = 3'b001, //  absolute: mem[scalar[s/v-1:0]]
    mrl = 3'b010, //  relative: mem[addr + scalar[s/v-1:0]]
    cop = 3'b101, //  co-operand:
                  //    - for array: acc
                  //    - for controller:
                  //        - scalar[1:0] = 00: reduction min
                  //        - scalar[1:0] = 01: reduction add
                  //        - scalar[1:0] = 10: reduction max
                  //        - scalar[1:0] = 11: reduction flag
    mac = 3'b001, //  absolute: mem[cOp[s/v-1:0]]
    mrc = 3'b010; //  relative: mem[addr + cOp[s/v-1:0]]
*****************************************************************************/
    // in ARRAY
    task VADD; // value add:
               // acc[i] <= acc[i] + {(n-8){aScalar[7]}}, aScalar}
        input [c-1:0] value;

        begin   aOpCode     = add   ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task ADD; // absolute add
              // acc[i] <= acc[i] + vectMem[i][aScalar[v-1:0]]
        input [c-1:0] value;

        begin   aOpCode     = add   ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RADD; // relative add:
               // acc[i] <= acc[i] + vectMem[i][addrVect[i] + aScalar[v-1:0]]
        input [c-1:0] value;

        begin   aOpCode     = add   ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        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 [c-1:0] value;

    begin   aOpCode     = add   ;
            aOperand    = mri   ;
            aScalar     = value ;
            endLine             ;
    end
endtask

task CADD; // co-operand add:
           // acc[i] <= acc[i] + acc
    begin   aOpCode     = add       ;
            aOperand    = cop       ;
            aScalar     = {c{1'b0}} ;
            endLine                 ;
    end
endtask

task CAADD; // co-operand absolute add
          // acc[i] <= acc[i] + vectMem[i][acc[v-1:0]]
    input [c-1:0] value;

    begin   aOpCode     = add       ;
            aOperand    = mac       ;
            aScalar     = {c{1'b0}} ;
            endLine                 ;
    end
endtask

task CRADD; // co-operand relative add:
            // acc[i] <= acc[i] + vectMem[i][addrVect[i] + acc[v-1:0]]
    input [c-1:0] value;

    begin   aOpCode     = add       ;
            aOperand    = mrc       ;
            aScalar     = {c{1'b0}} ;
            endLine                 ;
    end
endtask

// in CONTROLLER
task cVADD; // value add:
            // acc <= acc + {(n-8){cScalar[7]}}, cScalar}
    input [c-1:0] value;

    begin   cOpCode     = add   ;
            cOperand    = val   ;
```

```
                cScalar     = value ;
        end
    endtask

    task cADD; // immediate add:
               // acc <= acc + mem[cScalar[s-1:0]]
        input [c-1:0] value;

        begin   cOpCode     = add   ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cRADD; // relative add:
                // acc <= acc + mem[addr + cScalar[s-1:0]]
        input [c-1:0] value;

        begin   cOpCode     = add   ;
                cOperand    = mrl   ;
                cScalar     = value ;
        end
    endtask

    task cRIADD; // relative add:
                // acc <= acc + mem[addr + cScalar[s-1:0]]
        input [c-1:0] value;

        begin   cOpCode     = add   ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCADD; // co-operand add:
                // acc <= acc + cop,
                //      - scalar = 00: cop = reduction min
                //      - scalar = 01: cop = reduction add
                //      - scalar = 10: cop = reduction max
                //      - scalar = 11: cop = reduction flag
        input [c-1:0] value;

        begin   cOpCode     = add   ;
                cOperand    = cop   ;
                cScalar     = value ;
        end
    endtask
//================================================================================
```

Binary form generator for the instruction XXX is defined similarly with for the instruction ADD, in files named `cgXXX.v`.

The set of programs are defined in the following file:

```
/*********************************************************************************
THE RUNNING PROGRAM: theProgram.v

Version: 1
Author: Gheorghe M. Stefan
*********************************************************************************/
    //`include "03_matrixVectMult.v"
    //`include "03_matrixMatrixMult.v"
    //`include "03_matrixTranspose.v"
    //`include "03_matrixLoad.v"
    //`include "04_cI2F.v"
    //`include "04_cF2I.v"
    `include "03_examples.v"
//================================================================================
```

This is the file sent to the code generator.

A set of simple programs used to test the simulator and to show some simple feature of the accelerator is defined in the following file:

```
/*****************************************************************************
EXAMPLES: 03_examples.v

Simple programs used to demonstarate some features of the accelerator
*****************************************************************************/

/*****************************************************************************
Test program: 03_examples.v (appropriately commented)
    - activate all cells
    - acc[i] <= i, for i = 0,1,...,15
    - acc <= acc[0]+acc[1]+...+acc[15]
*****************************************************************************/
/*
    cSTART;     ACTIVATE; // start cycle counter; activate all cells
    cNOP;       IXLOAD;   // load the index of each cell in accumulator
    cNOP;       NOP;      // latency step 1
    cNOP;       NOP;      // latency step 2
    cNOP;       NOP;      // latency step 3
    cCLOAD(0);  NOP;      // acc <= sum of indexes
    cSTOP;      NOP;      // stop cycle counter
    cNOP;       NOP;      // to show cycle coounter stoped
    cHALT;      NOP;
//*/
//============================================================================


/*****************************************************************************
Test program: 03_examples.v (appropriately commented)
    - activate all cells
    - acc[i] <= i for 1 = 0,1,...,15
    - memVect[i][4] <= acc[i] for 1 = 0,1,...,15
    - acc[i] <= acc[i] x vectMem[i][4]
    - acc <= acc[0]+acc[1]+...+acc[15]
    - mem[24] <= acc = innerProduct(index, index)
*****************************************************************************/
/*
    cSTART;     ACTIVATE; // activate all cells
    cNOP;       IXLOAD;   // acc[i] <= index
    cNOP;       STORE(4); // memVect[i][4] <= acc[i], for all i
    cNOP;       MULT(4);  // acc[i] <= acc[i] * memVect[i][4]
    cNOP;       NOP;      // latency step 1
    cNOP;       NOP;      // latency step 2
    cNOP;       NOP;      // latency step 3
    cCLOAD(0);  NOP;      // acc <= reductionAdd(acc[i])
    cSTORE(24); NOP;      // mem[24] <= acc
    cSTOP;      NOP;      // stop cycle counter
    cHALT;      NOP;
//*/
//============================================================================
```

```
/*****************************************************************************
Test program: 03_examples.v (appropriately commented)
    - activate all cells
    - 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
    - acc <= acc[0]+acc[1]+...+acc[15] only for the active cells
*****************************************************************************/
/*
    cNOP;       ACTIVATE;    // activate all cells
    cNOP;       IXLOAD;      // acc[i] <= index
    cNOP;       VSUB(5);     // {cr, acc[i]} <= acc[i] - 5
    cNOP;       WHERENCARRY; // where cr=1 remain active
    cNOP;       VSUB(10);    // {{cr, acc[i]} <= acc[i] - (15 - 5)
    cNOP;       WHERECARRY;  // where cr=0 remain active
    cNOP;       VLOAD(1);
    cNOP;       ENDWHERE;    // reactivate where the second WHERE acted
    cNOP;       ENDWHERE;    // reactivate where the first WHERE acted
    cNOP;       NOP;         // latency step 3
    cCLOAD(0);  NOP;         // acc <= number of active cells
    cHALT;      NOP;
//*/
//============================================================================


/*****************************************************************************
Test program: 03_examples.v (appropriately commented)
    - activate all cells
    - acc <= 8; initialize the loop counter with l-1
    - acc[i} <= i; load index
    - do (acc+1) times
          acc[i] <= acc[i]/2
          acc[i] <= acc[i] + 99
*****************************************************************************/
//*
        cNOP;           ACTIVATE;
        cVLOAD(8);      IXLOAD;
LB(1);  cNOP;           SHRIGHT;
        cBRNZDEC(1);    VADD(99); // branch if acc=0 and acc<=acc-1
        cHALT;          NOP;
//*/
//============================================================================
```

The last program is active in the previous file.

Bibliography

[Alfke '05] Peter Alfke: "Metastable Recovery in Virtex-II Pro FPGAs", *Application Note: Virtex-II Pro Family*, [Online]. Available: `http://www.xilinx.com/support/documentation/application\_notes/xapp094.pdf`, XILINX, 2005.

[Andonie '95] Răzvan Andonie, Ilie Gârbacea: *Algoritmi fundamentali. O perspectivă C^{++}*, Ed. Libris, Cluj-Napoca, 1995. (in Roumanian)

[Ajtai '83] M. Ajtai, et al.: "An O(n log n) sorting network", Proc. 15th Ann. ACM Symp. on Theory of Computing, Boston, Mass., 1983.

[Batcher '68] K. E. Batcher: "Sorting networks and their applications", in *Proc. AFIPS Spring Joint Computer Conference*, vol. 32, 1968.

[Benes '68] Václav E. Beneš: *Mathematical Theory of Connecting Networks and Telephone Traffic*. New York: Academic, 1968.

[Blakeslee '79] T. R. Blakeslee: *Digital Design with Standard MSI and LSI*, John Wiley & Sons, 1979.

[Booth '67] T. L. Booth: *Sequential Machines and Automata Theory*, John Wiley & Sons, Inc., 1967.

[Bremermann '62] H. G. Bremermann: "Optimization through Evolution and Recombination", in *Self-Organizing Systems*, ed.: M. C. Yovits, S. Cameron, Washington DC, Spartan, 1962.

[Calude '82] Cristian Calude: *Complexitatea calculului. Aspecte calitative* (The Complexity of Computation. Qualitative Aspects), Ed. Stiintifica si Enciclopedica, Bucuresti, 1982.

[Calude '94] Cristian Calude: *Information and Randomness*, Springer-Verlag, 1994.

[Casti '92] John L. Casti: *Reality Rules: II. Picturing the World in Mathematics - The Frontier*, John Wiley & Sons, Inc., 1992.

[Cavanagh '07] Joseph Cavanagh: *Sequential Logic. Analysis and Synthesis*, CRC Taylor & Francis, 2007.

[Chaitin '66] Gregory Chaitin: "On the Length of Programs for Computing Binary Sequences", *J. of the ACM*, Oct., 1966.

[Chaitin '70] Gregory Chaitin: "On the Difficulty of Computation", in *IEEE Transactions of Information Theory*, ian. 1970.

[Chaitin '77] Gregory Chaitin: "Algorithmic Information Theory", in *IBM J. Res. Develop.*, Iulie, 1977.

[Chaitin '87] Gregory Chaitin: *Algorithmic Information Theory*, Cambridge University Press, 1987.

[Chaitin '90] Gregory Chaitin: *Information, Randomness and Incompletness*, World Scientific,1990.

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

[Chaitin '06] Gregory Chaitin: "The Limit of Rason", in *Scientific American*, Martie, 2006.

[Chomsky '56] Noam Chomsky, "Three Models for the Description of Languages", *IEEE Trans. on Information Theory*, 2:3 , 1956.

[Chomsky '59] Noam Chomsky, "On Certain Formal Properties of Grammars", *Information and Control*, 2:2, 1959.

[Chomsky '63] Noam Chomsky, "Formal Properties of Grammars", *Handbook of Mathematical Psychology*, Wiley, New-York, 1963.

[Church '36] Alonzo Church: "An Unsolvable Problem of Elementary Number Theory", in *American Journal of Mathematics*, vol. 58, pag. 345-363, 1936.

[Clare '72] C. Clare: *Designing Logic Systems Using State Machines*, Mc Graw-Hill, Inc., 1972.

[Cormen '90] Thomas H. Cormen, Charles E. Leiserson, Donsld R. Rivest: *Introduction to Algorithms*, MIT Press, 1990.

[Dascălu '98] Monica Dascălu, Eduard Franţi, Gheorghe Ştefan: "Modeling Production with Artificial Societies: the Emergence of Social Structure", in S. Bandini, R. Serra, F. Suggi Liverani (Eds.): *Cellular Automata: Research Towars Industry. ACRI '98 - Proceedings of the Third Conference on Cellular Automata for Research and Industry*, Trieste, 7 - 9 October 1998, Springer Verlag, 1998. p 218 - 229.

[Dascălu '98a] Monica Dascălu, Eduard Franţi, Gheorghe Ştefan: "Artificial Societies: a New Paradigm for Complex Systems' Modeling", in *IFAC Conference on Supplemental Ways for Improving International Stability - SWIIIS '98*, May 14-16, Sinaia, 1998. p.62-67.

[Drăgănescu '84] Mihai Drăgănescu: "Information, Heuristics, Creation", in Plauder, I. (ed): *Artificial Inteligence and Information Control System of Robots*, Elsevier Publishers B. V. (North-Holland), 1984.

[Drăgănescu '91] Mihai Drăgănescu, Gheorghe Ştefan, Cornel Burileanu: *Electronica functională*, Ed. Tehnică, Bucureşti, 1991 (in Roumanian).

[Einspruch '86] N. G. Einspruch ed.: *VLSI Electronics. Microstructure Science. vol. 14 : VLSI Design*, Academic Press, Inc., 1986.

[Einspruch '91] N. G. Einspruch, J. L. Hilbert: *Application Specific Integrated Circuits (ASIC) Technology*, Academic Press, Inc., 1991.

[Ercegovac '04] Miloš D. Ercegovac, Tomás Lang: *Digital Arithmetic*, Morgan Kaufman, 2004.

[Flynn '72] Flynn, M.J.: "Some computer organization and their affectiveness", *IEEE Trans. Comp.* C21:9 (Sept. 1972), pp. 948-960.

[Gheolbanoiu '14] Alexandru Gheolbanoiu, Dan Mocanu, Radu Hobincu, Lucian Petrica: "Cellular Automaton pRNG with a Global Loop for Non-Uniform Rule Control", 18th International Conference on Ciruits, Systems, Communications and Computers (CSCC 2014), Santorini Island, Greece, July 17-21, 2014, vol. II, 415-420.[Online]. Available: `http://www.euromentor.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-15.pdf`

[Glushkov '66] V. M. Glushkov: *Introduction to Cybernetics*, Academic Press, 1966.

[Gödels '31] Kurt Gödel: "On Formally Decidable Propositions of Principia Mathematica and Related Systems I", reprinted in S. Fefermann et all.: *Collected Works I: Publications 1929 - 1936,* Oxford Univ. Press, New York, 1986.

[Hartley '95] Richard I. Hartley: *Digit-Serial Computation*, Kulwer Academic Pub., 1995.

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

[Hascsi '96] Zoltan Hascsi, Bogdan Mîţu, Mariana Petre, Gheorghe Ştefan, "High-Level Synthesis of an Enchanced Connex memory", in *Proceedings of the International Semiconductor Conference*, Sinaia, October 1996, p. 163-166.

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

[Helbing 89'] Walter A. Helbing, Veljko M. Milutinovic: "Architecture and Design of a 32-bit GaAs Microprocessor", in [Milutinovic 89'].

[Hennessy '07] John L. Hennessy, David A. Patterson: *Computer Architecture: A Quantitative Approach*, Fourth Edition, Morgan Kaufmann, 2007.

[Hennie '68] F. C. Hennie: *Finite-State Models for Logical Machine*, John Wiley & Sons, Inc., 1968.

[Hillis '85] W. D. Hillis: *The Connection Machine*, The MIT Press, Cambridge, Mass., 1985.

[Kaeslin '01] Hubert Kaeslin: *Digital Integrated Circuit Design*, Cambridge Univ. Press, 2008.

[Keeth '01] Brent Keeth, R. jacob Baker: *DRAM Circuit Design. A Tutorial*, IEEE Press, 2001.

[Kleene '36] Stephen C. Kleene: "General Recursive Functions of Natural Numbers", in *Math. Ann*., 112, 1936.

[Karim '08] Mohammad A. Karim, Xinghao Chen: *Digital Design*, CRC Press, 2008.

[Knuth '73] D. E. Knuth: *The Art of Programming. Sorting and Searching*, Addison-Wesley, 1973.

[Kolmogorov '65] A.A. Kolmogorov: "Three Approaches to the Definition of the Concept "Quantity of Information" ", in *Probl. Peredachi Inform.*, vol. 1, pag. 3-11, 1965.

[Kung '79] H. T. Kung, C. E. Leiserson: "Algorithms for VLSI processor arrays", in [Mead '79].

[Ladner '80] R. E. ladner, M. J. Fischer: "Parallel prefix computation", *J. ACM*, Oct. 1980.

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

[Maliţa '06] Mihaela Maliţa, Gheorghe Ştefan, Marius Stoian: "Complex vs. Intensive in Parallel Computation", in *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C - ICCGI*, 2006 Bucharest, Romania, August 1-3, 2006

[Maliţa '07] Mihaela Maliţa, Gheorghe Ştefan, Dominique Thiébaut: "Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" in *International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing* June 17, 2007 Seattle, WA, USA.

[Maliţa '13] Mihaela Maliţa, Gheorghe M. Ştefan: "Control Global Loops in Self-Organizing Systems", *ROMJIST*, Volume 16, Numbers 23, 2013, 177-191. [Online]. Available: `http://www.imt.ro/romjist/Volum16/Number16\_2/pdf/05-Malita-Stefan2.pdf`

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

[Mead '79] Carver Mead, Lynn Convay: *Introduction to VLSI Systems*, Addison-Wesley Pub, 1979.

[MicroBlaze] *** *MicroBlaze Processor. Reference Guide.* [Online]. Available: `http://www.xilinx.com/support/documentation/sw\_manuals/xilinx14\_1/mb\_ref\_guide.pdf`

[Milutinovic 89'] Veljko M. Milutinovic (ed.): *High-Level Language Computer Architecture*, Computer Science Press, 1989.

[Mindell '00] Arnold Mindell: *Quantum Mind. The Edge Between Physics and Psychology*, Lao Tse Press, 2000.

[Minsky '67] M. L. Minsky: *Computation: Finite and Infinite Machine*, Prentice - Hall, Inc., 1967.

[Mîţu '00] Bogdan Mîţu, Gheorghe Ştefan, "Low-Power Oriented Microcontroller Architecture", in *CAS 2000 Proceedings*, Oct. 2000, Sinaia, Romania

[Moto-Oka '82] T. Moto-Oka (ed.): *Fifth Generation Computer Systems*, North-HollandPub. Comp., 1982.

[Omondi '94] Amos R. Omondi: *Computer Arithmetic. Algorithm, Architecture and Implementation*, Prentice Hall, 1994.

[Palnitkar '96] Samir Palnitkar: *Verilog HDL. AGuide to Digital Design and Synthesis*, SunSoft Press, 1996.

[Parberry 87] Ian Parberry: *Parallel Complexity Theory*. Research Notes in Theoretical Computer science. Pitman Publishing, London, 1987.

[Parberry 94] Ian Parberry: *Circuit Complexity and Neural Networks*, The MIT Presss, 1994.

[Patterson '05] David A. Patterson, John L.Hennessy: *Computer Organization & Design. The Hardware / Software Interface*, Third Edition, Morgan Kaufmann, 2005.

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

[Păun '85] A. Păun, Gh. Ştefan, A. Birnbaum, V. Bistriceanu, "DIALISP - experiment de structurare neconventionala a unei masini LISP", in *Calculatoarele electronice ale generatiei a cincea*, Ed. Academiei RSR, Bucuresti 1985. p. 160 - 165.

[Post '36] Emil Post: "Finite Combinatory Processes. Formulation I", in*The Journal of Symbolic Logic*, vol. 1, p. 103 -105, 1936.

[Prince '99] Betty Prince: *High Performance Memories. New architecture DRAMs and SRAMs evolution anad function*, John Wiley & Sons, 1999.

[Rafiquzzaman '05] Mohamed Rafiquzzaman: *Fundamentals of Digital Logic and Microcomputer Design*, Fifth Edition, Wiley – Interscience, 2005.

[Salomaa '69] Arto Salomaa: *Theory of Automata*, Pergamon Press, 1969.

[Salomaa '73] Arto Salomaa: *Formal Languages*, Academic Press, Inc., 1973.

[Salomaa '81] Arto Salomaa: *Jewels of Formal Language Theory*, Computer Science Press, Inc., 1981.

[Savage '87] John Savage: *The Complexity of Computing*, Robert E. Krieger Pub. Comp., 1987.

[Shankar '89] R. Shankar, E. B. Fernandez: *VLSI Computer Architecture*, Academic Press, Inc., 1989.

[Shannon '38] C. E. Shannon: "A Symbolic Annalysis of Relay and Switching Circuits", *Trans. AIEE*, vol. 57, p.713-723, 1938.

[Shannon '48] C. E. Shannon: "A Mathematical Theory of Communication", *Bell System Tech. J.*, Vol. 27, 1948.

[Shannon '56] C. E. Shannon: "A Universal Turing Machine with Two Internal States", in *Annals of Mathematics Studies, No. 34: Automata Studies*, Princeton Univ. Press, pp 157-165, 1956.

[Sharma '97] Ashok K. Sharma: *Semiconductor Memories. Techology, Testing, and Reliability*, Wiley – Interscience, 1997.

[Sharma '03] Ashok K. Sharma: *Advanced Smiconductor Memories. Architectures, Designs, and Applications*, Whiley-Interscience, 2003.

[Solomonoff '64] R. J. Solomonoff: "A Formal Theory of Inductive Inference", in *Information and Control*, vol. 7, pag. 1- 22 , pag. 224-254, 1964.

[Spira '71] P. M. Spira: "On time-Hardware Complexity Tradeoff for Boolean Functions", in *Preceedings of Fourth Hawaii International Symposium on System Sciences*, pp. 525-527, 1971.

[Stoian '07] Marius Stoian, Gheorghe Ştefan: "Stacks or File-Registers in Cellular Computing?", in *CAS, Sinaia 2007*.

[Streinu '85] Ileana Streinu: "Explicit Computation of an Independent Gödel Sentence", in *Recursive Functions Theory Newsletter*, June 1985.

[Ştefan '97] Denisa Ştefan, Gheorghe Ştefan, "Bi-thread Microcontroller as Digital Signal Processor", in *CAS '97 Proceedings, 1997 International Semiconductor Conference*, October 7 -11, 1997, Sinaia, Romania.

[Ştefan '99] Denisa Ştefan, Gheorghe Ştefan: "A Procesor Network without Interconnectio Path", in *CAS 99 Proceedings, Oct., 1999*, Sinaia, Romania. p. 305-308.

[Ştefan '80] Gheorghe Ştefan: *LSI Circuits for Processors*, Ph.D. Thesis (in Roumanian), Politechnical Institute of Bucharest, 1980.

[Ştefan '83] Gheorghe Ştefan: "Structurari neechilibrate in sisteme de prelucrare a informatiei", in *Inteligenta artificiala si robotica*, Ed. Academiei RSR, Bucuresti, 1983. p. 129 - 140.

[Ştefan '83] Gheorghe Ştefan, et al.: *Circuite integrate digitale*, Ed. Did. si Ped., Bucuresti, 1983.

[Ştefan '84] Gheorghe Ştefan, et al.: "DIALISP - a LISP Machine", in *Proceedings of the ACM Symposium on LISP and Functional Programming*, Austin, Texas, Aug. 1984. p. 123 - 128.

[Ştefan '85] Gheorghe Ştefan, A. Păun, "Compatibilitatea functie - structura ca mecanism al evolutiei arhitecturale", in *Calculatoarele electronice ale generatiei a cincea*, Ed. Academiei RSR, Bucuresti, 1985. p. 113 - 135.

[Ştefan '85a] Gheorghe Ştefan, V. Bistriceanu, A. Păun, "Catre un mod natural de implementare a LISP-ului", in *Sisteme cu inteligenta artificiala*, Ed. Academiei Romane, Bucuresti, 1991 (paper at *Al doilea simpozion national de inteligenta artificiala*, Sept. 1985). p. 218 - 224.

[Ştefan '86] Gheorghe Stefan, M. Bodea, "Note de lectura la volumul lui T. Blakeslee: Proiectarea cu circuite MSI si LSI", in *T. Blakeslee: Prioectarea cu circuite integrate MSI si LSI*, Ed. Tehnica, Bucuresti, 1986 (translated from English by M. Bodea, M. Hancu, Gh. Stefan). p. 338 - 364.

[Ştefan '86a] Gheorghe Stefan, "Memorie conexa" in *CNETAC 1986* Vol. 2, IPB, Bucuresti, 1986, p. 79 - 81.

[Ştefan '91] Gheorghe Ştefan: *Functie si structura in sistemele digitale*, Ed. Academiei Romane, 1991.

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

[Ştefan '93] Gheorghe Ştefan: *Circuite integrate digitale*. Ed. Denix, 1993.

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

[Ştefan '96] Gheorghe Ştefan, Mihaela Maliţa: "Chaitin's Toy-Lisp on Connex Memory Machine", *Journal of Universal Computer Science*, vol. 2, no. 5, 1996, pp. 410-426.

[Ştefan '97] Gheorghe Ştefan, Mihaela Maliţa: "DNA Computing with the Connex Memory", in *RECOMB 97 First International Conference on Computational Molecular Biology*. January 20 - 23, Santa Fe, New Mexico, 1997. p. 97-98.

[Ştefan '97a] Gheorghe Ştefan, Mihaela Maliţa: " The Splicing Mechanism and the Connex Memory", *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, Indianapolis, April 13 - 16, 1997. p. 225-229.

[Ştefan '98] Gheorghe Ştefan, "Silicon or Molecules? What's the Best for Splicing", in Gheorghe Păun (ed.): *Computing with Bio-Molecules. Theory and Experiments.* Springer, 1998. p. 158-181

[Ştefan '98a] Gheorghe Ştefan, " "Looking for the Lost Noise" ", in *CAS '98 Proceedings, Oct. 6 - 10, 1998*, Sinaia, Romania. p.579 - 582.
`http://arh.pub.ro/gstefan/CAS98.pdf`

[Ştefan '98b] Gheorghe Ştefan, "The Connex Memory: A Physical Support for Tree / List Processing" in *The Roumanian Journal of Information Science and Technology*, Vol.1, Number 1, 1998, p. 85 - 104.

[Ştefan '98] Gheorghe Ştefan, Robrt Benea: "Connex Memories & Rewrieting Systems", in *MELECON '98*, Tel-Aviv, May 18 -20, 1998.

[Ştefan '99] Gheorghe Ştefan, Robert Benea: "Experimente in info cu acizi nucleici", in M. Drăgănescu, Ştefan Trăusan-Matu (eds): *Natura realitatii fizice si a informatiei*, Editura Tempus, 1999.

[Ştefan '99a] Gheorghe Ştefan: "A Multi-Thread Approach in Order to Avoid Pipeline Penalties", in *Proceedings of 12th International Conference on Control Systems and Computer Science*, Vol. II, May 26-29, 1999, Bucharest, Romania. p. 157-162.

[Ştefan '00] Gheorghe Ştefan: "Parallel Architecturing starting from Natural Computational Models", in *Proceedings of the Romanian Academy*, Series A: Mathematics, Physics, Technical Sciences, Information Science, vol. 1, no. 3 Sept-Dec 2000.

[Ştefan '01] Gheorghe Ştefan, Dominique Thiébaut, "Hardware-Assisted String-Matching Algorithms", in *WABI 2001, 1st Workshop on Algorithms in BioInformatics, BRICS*, University of Aarhaus, Danemark, August 28-31, 2001.

[Ştefan '04] Gheorghe Ştefan, Mihaela Maliţa: "Granularity and Complexity in Parallel Systems", in *Proceedings of the 15 IASTED International Conf, 2004*, Marina Del Rey, CA, ISBN 0-88986-391-1, pp.442-447.

[Ştefan '06] Gheorghe Ştefan: "Integral Parallel Computation", in *Proceedings of the Romanian Academy, Series A: Mathematics, Physics, Technical Sciences, Information Science*, vol. 7, no. 3 Sept-Dec 2006, p.233-240.

[Ştefan '06a] Gheorghe Ştefan: "A Universal Turing Machine with Zero Internal States", in *Romanian Journal of Information Science and Technology*, Vol. 9, no. 3, 2006, p. 227-243

[Ştefan '06b] Gheorghe Ştefan: "The CA1024: SoC with Integral Parallel Architecture for HDTV Processing", invited paper at *4th International System-on-Chip (SoC) Conference & Exhibit*, November 1 & 2, 2006, Radisson Hotel Newport Beach, CA

[Ştefan '06c] Gheorghe Ştefan, Anand Sheel, Bogdan Mîţu, Tom Thomson, Dan Tomescu: "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing", in *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University, August 20 to 22, 2006.

[Ştefan '06d] Gheorghe Ştefan: "The CA1024: A Massively Parallel Processor for Cost-Effective HDTV", in *SPRING PROCESSOR FORUM: Power-Efficient Design*, May 15-17, 2006, Doubletree Hotel, San Jose, CA.

[Ştefan '06e] Gheorghe Ştefan: "The CA1024: A Massively Parallel Processor for Cost-Effective HDTV", in *SPRING PROCESSOR FORUM JAPAN*, June 8-9, 2006, Tokyo.

[Ştefan '07] Gheorghe Ştefan: "Membrane Computing in Connex Environment", invited paper at *8th Workshop on Membrane Computing (WMC8)* June 25-28, 2007 Thessaloniki, Greece

[Ştefan '07a] Gheorghe Ştefan, Marius Stoian: "The efficiency of the register file based architectures in OOP languages era", in *SINTES13* Craiova, 2007.

[Ştefan '07b] Gheorghe Ştefan: "Chomsky's Hierarchy & a Loop-Based Taxonomy for Digital Systems", in *Romanian Journal of Information Science and Technology* vol. 10, no. 2, 2007.

[Ştefan '14] Gheorghe M. Stefan, Mihaela Malita: "Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation", *18th International Conference on Ciruits, Systems, Communications and Computers (CSCC 2014)*, Santorini Island, Greece, July 17-21, 2014, 582-597. [Online]. Available: `http://www.inase.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-42.pdf`

[Sutherland '02] Stuart Sutherland: *Verilog 2001. A Guide to the New Features of the Verilog Hardware Description Language*, Kluwer Academic Publishers, 2002.

[Tabak '91] D. Tabak: *Advanced Microprocessors*, McGrow- Hill, Inc., 1991.

[Tanenbaum '90] A. S. Tanenbaum: *Structured Computer Organisation* third edition, Prentice-Hall, 1990.

[Thiébaut '06] Dominique Thiébaut, Gheorghe Ştefan, Mihaela Maliţa: "DNA search and the Connex technology" in *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C - ICCGI*, 2006 Bucharest, Romania, August 1-3, 2006

[ThoughtSpot] ThoughtSpot: *SpotIQ AI-Driven Analytics. Architecting Automated Insights for the Masses*, White paper, [Online]. Available: `https://go.thoughtspot.com/rs/816-MBH-536/images/ThoughtSpot-SpotIQ-AI-Driven-Analytics-White-Paper.pdf`

[Tokheim '94] Roger L. Tokheim: *Digital Principles*, Third Edition, McGraw-Hill, 1994.

[Turing '36] Alan M. Turing: "On computable Numbers with an Application to the Eintscheidungsproblem", in *Proc. London Mathematical Society,* 42 (1936), 43 (1937).

[Vahid '06] Frank Vahid: *Digital Design*, Wiley, 2006.

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

[Uyemura '02] John P. Uyemura: *CMOS Logic Circuit Design*, Kluver Academic Publishers, 2002.

[Ward '90] S. A. Ward, R. H. Halstead: *Computation Structures*, The MIT Press, McGraw-Hill Book Company, 1990.

[Wedig '89] Robert G. Wedig: "Direct Correspondence Architectures: Principles, Architecture, and Design" in [Milutinovic '89].

[Waksman '68] Abraham Waksman, "A permutation network," in *J. Ass. Comput. Mach.*, vol. 15, pp. 159-163, Jan. 1968.

[webRef_1] [Online]. Available: `http://www.fpga-faq.com/FAQ_Pages/0017_Tell_me_about_metastables.htm`

[webRef_2] [Online]. Available: `http://www.fpga-faq.com/Images/meta_pic_1.jpg`

[webRef_3] [Online]. Available: `http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html#fsa_pfx`

[Weste '94] Neil H. E. Weste, Kamran Eshraghian: *Principle of CMOS VLSI Design. ASystem Perspective*, Second Edition, Addisson Wesley, 1994.

[Wolfram '02] Stephen Wolfram: *A New Kind of Science*, Wolfram Media, Inc., 2002.

[Zurada '95] Jacek M. Zurada: *Introductin to Artificial Neural network*, PWS Pub. Company, 1995.

[Yanushkevich '08] Svetlana N. Yanushkevich, Vlad P. Shmerko: *Introduction to Logic Design*, CRC Press, 2008.