

# The Massively Parallel Processor\*

A. Broggi, G. Conte  
Dip. di Ingegneria dell'Informazione  
Università di Parma, Italy

F. Gregoretti, C. Sansoè  
Dip. di Elettronica  
Politecnico di Torino, Italy

L.M. Reyneri  
Dip. di Ingegneria dell'Informazione  
Università di Pisa, Italy

## Abstract

*This paper describes a complete 6-year project, starting from its theoretical basis up to the hardware and software system implementation, and to the description of its future evolution.*

*The main goal of the project is to develop a subsystem that operates as a processing unit attached to a standard workstation and in perspective as a low-cost low-sized specialized embedded system devoted to low level image analysis and cellular neural networks emulation. The architecture has been extensively used for basic low level image analysis tasks up to optical flow computation and feature tracking, showing encouraging performances even in the first prototype version.*

## 1 Introduction

The PAPRICA system [5, 11] (an acronym for PARallel PProcessor for Image Checking and Analysis) described in this paper and shown in fig. 1 has the main characteristics of a conventional mesh-connected SIMD array but it has been specialized to the following objectives:

- to directly support a computational paradigm based on mathematical morphology [16] also on data structures larger than the physical size of the machine;
- to support hierarchical non-mesh data structures;
- to provide a low-cost experimental tool for research in the fields of image processing, VLSI design automation, and neural algorithms.

The kernel of the system is a bidimensional mesh of single-bit Processing Elements (PE) with a direct 8-neighborhood [7]. The instruction set can be described in terms of mathematical morphology operators [16], augmented with logical operations. Control flow operations are defined over the entire SIMD mesh, and

\*This work was partially supported by CNR Progetto Finalizzato Trasporti under contracts 93.01813.PF74, 93.04759.ST74 and 91.01034.PF93.

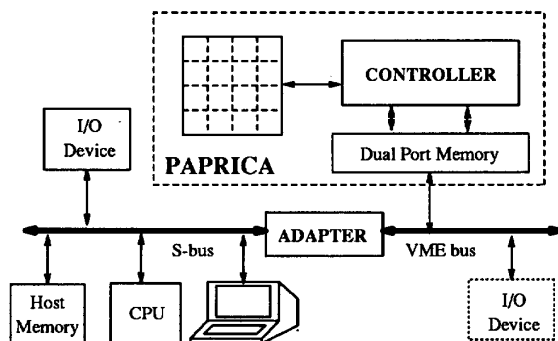


Figure 1: The complete system

hardware mechanisms are provided for the virtualization of the operation of the physically limited array over a large bidimensional data structure stored in a linearly addressable memory. An additional mechanism allows to map more complex hierarchical data structures on the same bidimensional processing grid.

The PAPRICA system has been designed to operate as a coprocessor of a general purpose host workstation. Data and instructions are transferred to and from the host through a dual port memory connected to a standard VME bus, while processing proceeds concurrently with the host activities.

The current PAPRICA implementation is based on a number of custom integrated circuits, which have been designed with a particular emphasis on global project management issues such as cost and development time. For this reason conservative design and engineering choices have often been taken leading to performance limitations both in terms of processing speed and in the number of available PEs.

The original target of the PAPRICA system is the acceleration of tasks related to the design and verification of IC layouts and masks. Nevertheless the generality of the mathematical morphology computational paradigm allows an efficient use of the architecture

in many tasks related to the processing of bidimensional data structures. The application fields range from image analysis for automotive applications, to the detection of faults in VLSI circuits, while perspectives applications are envisaged in artificial neural network emulation. The paper is organized as follows: sect. 2 presents the computational paradigm of PAPRICA system; sect. 3 and sect. 4 present the external and internal architecture of PAPRICA respectively; sect. 5 presents the software operating environment, while sect. 6 ends the paper with some concluding remarks.

## 2 Hierarchical Morphology

The computational model which has inspired the design of PAPRICA derives from the work of Serra [16], although it has been modified to extend its use to hierarchical architectures such as pyramids.

Mathematical morphology [16, 12] provides a bitmap approach to the processing of discrete images which is derived from the set theory. *Images* are defined as collections of *pixels*  $\mathcal{P}$ , which are  $N$ -tuples of integers:

$$\mathcal{P} \triangleq (\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_i, \dots, \mathcal{P}_N), \quad (1)$$

where  $\mathcal{P}_i \in Z$  is the  $i$ -th coordinate ( $Z$  is the set of integer numbers). The universe set of pixels is the discrete Euclidean  $N$ -space  $Z^N$ .

In practice images and similar types of data are always "size-limited", especially when they are stored in a computer memory. Therefore the concept of  $N$ -dimensional frame  $\mathcal{F}_N \subset Z^N$  is introduced and defined as a convex, size-limited, "rectangular" subset of  $Z^N$ :

$$\mathcal{F}_N \triangleq \{ \mathcal{P} \in Z^N \mid \mathcal{F}_{i,min} \leq \mathcal{P}_i \leq \mathcal{F}_{i,max}, \text{ for every } i \in [1 \dots N] \}, \quad (2)$$

where  $\mathcal{F}_{i,min}$  and  $\mathcal{F}_{i,max}$  are the bounds of the  $i$ -th coordinate.

The two pixels  $\mathcal{F}_{min} = (\mathcal{F}_{1,min}, \mathcal{F}_{2,min}, \dots, \mathcal{F}_{N,min})$  and  $\mathcal{F}_{max} = (\mathcal{F}_{1,max}, \mathcal{F}_{2,max}, \dots, \mathcal{F}_{N,max})$  can be viewed as the "corner" coordinates of  $\mathcal{F}_N$ , while the value  $f_i \triangleq (\mathcal{F}_{i,max} - \mathcal{F}_{i,min} + 1)$  is the *frame size* along the  $i$ -th coordinate.

A *framed image*  $\mathcal{I}$  is then defined as a subset of  $\mathcal{F}_N$ :

$$\mathcal{I} \triangleq \{ \mathcal{P} \mid \mathcal{P} \in \mathcal{F}_N \} \subseteq \mathcal{F}_N. \quad (3)$$

Consequently the operators of *dilation*  $\oplus$ , *erosion*  $\ominus$ , *complement*  $(\cdot)^c$ , *translation*  $(\cdot)_x$  and *transposition*  $(\cdot)$  in  $Z^N$ , introduced in [12], have been adapted to framed images. If  $A, B \subseteq \mathcal{F}_N$  and  $x \in \mathcal{F}_N$  are respectively two framed images and a pixel, the following definitions apply:

$$A \oplus B \triangleq \{ y \in \mathcal{F}_N \mid y = (a + b), \text{ for some } a \in A \text{ and } b \in B \} \quad (4)$$

$$A \ominus B \triangleq \{ y \in \mathcal{F}_N \mid (y + b) \in A, \text{ for every } b \in B \} \quad (5)$$

$$(A)^c \triangleq \{ y \in \mathcal{F}_N \mid y \notin A \} \quad (6)$$

$$(A)_x \triangleq \{ y \in \mathcal{F}_N \mid y = (a + x), \text{ for some } a \in A \} \quad (7)$$

$$(\check{A}) \triangleq \{ y \in \mathcal{F}_N \mid y = -a, \text{ for some } a \in A \}. \quad (8)$$

### 2.1 Data Structures and Hierarchies

Mathematical morphology as described above matches only computing architectures based on  $N$ -dimensional meshes [8, 9, 6, 13] and applies primarily to binary images. It cannot be used for other computing architectures having a more complex data organization, such as pyramids, hierarchical systems, multi-layered images and the PAPRICA system itself. In this section morphological operators are extended to such systems, without however entering too deeply into theory.

A *data structure*  $\mathcal{S}$  of *parallelism*  $\mathcal{C}$  over the frame  $\mathcal{F}_N$  is defined as a  $\mathcal{C}$ -tuple of framed images  $\mathcal{S}_j \subseteq \mathcal{F}_N$ , also called *layers*:

$$\mathcal{S} \triangleq (\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_j, \dots, \mathcal{S}_c). \quad (9)$$

Data structures can be used for: gray-scale images, color and multi-layered images and numerical matrices. An example of a simple data structure is the memory organization of a computer, where  $N = 1$  (linear organization) and  $\mathcal{C} = 16$  or 32 bits is the memory parallelism. Obviously frame limits are:  $\mathcal{F}_{1,min} = 0$  and  $\mathcal{F}_{1,max} = (\text{memory.size} - 1)$ .

Several independent data structures (with different dimensions, sizes, and parallelisms) can be combined together into what has been called a *data hierarchy*  $\mathcal{H}$  of *deepness*  $\mathcal{D}$ :

$$\mathcal{H} \triangleq (\mathcal{H}^1, \mathcal{H}^2, \dots, \mathcal{H}^\alpha, \dots, \mathcal{H}^{\mathcal{D}}). \quad (10)$$

This is a  $\mathcal{D}$ -tuple of possibly different data structures  $\mathcal{H}^\alpha$ , where the index  $\alpha \in [1 \dots \mathcal{D}]$  identifies the *hierarchy level* of parallelism  $\mathcal{C}^\alpha$  over a frame  $\mathcal{F}_{N_\alpha}^\alpha$ .

Data hierarchies  $\mathcal{H}$  can also be used to describe the "data storage organization"  $\mathcal{M}_{\mathcal{L}}$  of several computing architectures (e.g. disks, main memory, caches, registers, etc.). Note that the memory organization  $\mathcal{M}_{\mathcal{L}}$  of a computing system is not sufficient to completely define its interconnection topology. In fact this latter must be defined also in combination with the processor instruction set and the mechanisms devoted to the management of data interchanges inside the system. Some examples of the data hierarchy of existing parallel architectures are shortly described below [7, 2, 17]:

- **hypercubes (N-cubes):** their memory  $\mathcal{M}_{\mathcal{L}}$  is a flat data structure ( $\mathcal{D} = 1$ ). The dimension  $N_1 = N$  coincides with that of the cube. The parallelism  $\mathcal{C}^1$  defines the size (in bits) of each processor memory. Frame corners are  $\mathcal{F}_{i,min}^1 = 0$  and  $\mathcal{F}_{i,max}^1 = 1$ , for every  $i \in [1 \dots N]$ .

- **pyramids:** their memory  $\mathcal{M}_{\mathcal{L}}$  is a data hierarchy. The deepness  $\mathcal{D} > 1$  coincides with the number of pyramid levels. For each level  $\alpha$ , the dimension  $N_{\alpha} \geq 1$  defines the processor data organization (in most cases  $N_{\alpha} = 1, 2$  or  $3$ ). For each level, the parallelism  $\mathcal{C}^{\alpha}$  coincides with the size of each processor's memory.
- **2-D and 3-D meshes:** their memory  $\mathcal{M}_{\mathcal{L}}$  is a flat data structure ( $\mathcal{D} = 1$ ). Typical data dimensions are  $N_1 = 2$  or  $3$ , while  $\mathcal{C}^1$  is the number of bits per processor. Usually  $\mathcal{F}_{i,min}^1 = 0$  and  $\mathcal{F}_{i,max}^1$  defines the mesh width.
- **Golay hexagonal grids:** their memory  $\mathcal{M}_{\mathcal{L}}$  is a flat "lattice-type" subset of a bidimensional frame  $\mathcal{F}_2$  (not proven here):

$$\mathcal{M}_{\mathcal{L}} = \{\mathcal{P} \in \mathcal{F}_2 \mid (\mathcal{P}_1 + \mathcal{P}_2) \text{ is even}\} \quad (11)$$

- **PAPRICA:** the memory organization of PAPRICA is a two level data hierarchy ( $\mathcal{D} = 2$ ).

The first level ( $\alpha = 1$ ) is the bidimensional data structure  $\mathcal{K}$  of the "processor array" with parallelism  $\mathcal{C}^1 = 64$  and frame sizes  $f_1^1 = f_2^1 = 16$ . The sizes of this data structure can be easily extended to larger values by "virtualizing" the processor array, provided that the program satisfies a simple semantic constraint (see sect. 3.2). It then results a "virtual data structure"  $\mathcal{K}'$  with no limitations in size. This reconfigurable data structure is the key feature to map other computing architectures, by means of a specific *Update Block* construct, as described in sect. 3.2.2.

The second level of PAPRICA hierarchy ( $\alpha = 2$ ) is the *image memory*, which is organized as a data structure of dimension  $N_2 = 2$ . Parallelism  $\mathcal{C}^2$  and both frame sizes  $f_1^2$  and  $f_2^2$  are user-programmable. The product ( $\mathcal{C}^2 \cdot f_1^2 \cdot f_2^2$ ) is limited by the total memory size (4 to 8MB in the actual version).

A more complete description of PAPRICA memory organization is given in sect. 4.

## 2.2 Matching Operators and Hierarchical Morphology

The computational paradigm of PAPRICA is based on the concept of *matching operator*  $\mathcal{O}$ , which is derived from the *hit-miss transform* described in [16]. This is a rather general approach and includes the other morphological operators as special cases.

Matching operators in PAPRICA are defined as set transforms  $\mathcal{K} \mapsto \mathcal{K}$  where  $\mathcal{K}$  is the bidimensional data structure of PAPRICA processor array. A *simple N-dimensional matching template* is a couple  $\mathcal{Q} = (\mathcal{Q}_0, \mathcal{Q}_1)$ , where  $\mathcal{Q}_0, \mathcal{Q}_1 \subseteq \mathcal{F}_N$ , with the constraint that  $\mathcal{Q}_0 \cap \mathcal{Q}_1 = \emptyset$ . An *elementary matching* with a simple matching template is defined as:

$$A \mathcal{O} \mathcal{Q} \triangleq \{y \in \mathcal{F}_N \mid (y + q_1) \in A \text{ and } (y + q_0) \notin A, \text{ for every } q_1 \in \mathcal{Q}_1, q_0 \in \mathcal{Q}_0\} \quad (12)$$

or, in terms of the erosion operator:

$$A \mathcal{O} \mathcal{Q} = (A \ominus \mathcal{Q}_1) \cap (A^c \ominus \mathcal{Q}_0) \quad (13)$$

A *complemented matching*  $\mathcal{O}^c$  is also defined as

$$A \mathcal{O}^c \mathcal{Q} \triangleq (A \mathcal{O} \mathcal{Q})^c \quad (14)$$

A simple  $3 \times 3$  bidimensional matching template  $\mathcal{Q}$  can be sketched using the following notation:  $\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix}$ ,

where "x" is either 0, 1 or  $-$  iff the corresponding pixel of the matching template is an element of  $\mathcal{Q}_0, \mathcal{Q}_1$  or none of the two, respectively. The center of the matrix coincides with pixel  $(0, 0)$  of  $\mathcal{Q}_0$  and  $\mathcal{Q}_1$ .

A *composite matching* with a *matching list*  $\mathcal{Q}_{\mathcal{L}} = \{\mathcal{Q}^1, \dots, \mathcal{Q}^k, \dots\}$  is the union of elementary matchings:

$$A \mathcal{O} \mathcal{Q}_{\mathcal{L}} \triangleq \bigcup_k (A \mathcal{O} \mathcal{Q}^k) \quad (15)$$

A matching list can be sketched as a list of simple matching templates:

$$\mathcal{Q}_{\mathcal{L}} = \begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix}^c, \dots, K \begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix}, \dots \quad (16)$$

where  $K \in [2, 4, 8]$ . The superscript  $c$  declares that a complementary matching  $\mathcal{O}^c$  must be used in place of  $\mathcal{O}$  with that specific template, while the numeric constant  $K$  is a short form for the list of  $K$  possible rotations of the matching template by  $\frac{360}{K}$  degrees.

The concept of matching operators can be further extended to data structures and hierarchies. A *simple structure template* is a couple  $\mathcal{R} = (\mathcal{R}_0, \mathcal{R}_1)$ , where  $\mathcal{R}_0, \mathcal{R}_1$  are two structures which satisfy the constraint

$$\mathcal{R}_0_j \cap \mathcal{R}_1_j = \emptyset, \text{ for every } j \in [1 \dots C]. \quad (17)$$

If  $A$  is a data structure, an *elementary structure matching* with a simple structure template  $\mathcal{R}$  is a set transform given by:

$$A \mathcal{O} \mathcal{R} \triangleq \{y \in \mathcal{F}_N \mid (y + r_1_j) \in A_j \text{ and } (y + r_0_j) \notin A_j, \text{ for every } j \in [1 \dots C], r_1_j \in \mathcal{R}_1_j, r_0_j \in \mathcal{R}_0_j\} \quad (18)$$

while a *composite structure matching* is entirely defined by a *matching structures list*  $\mathcal{R}_{\mathcal{L}} = \{\mathcal{R}^1, \dots, \mathcal{R}^k, \dots\}$ :

$$A \mathcal{O} \mathcal{R}_{\mathcal{L}} \triangleq \bigcup_k (A \mathcal{O} \mathcal{R}^k). \quad (19)$$

The above definitions can be extended also to data hierarchies and to hierarchical matching operators. Therefore a *simple hierarchy template* is a couple  $\mathcal{T} = (\mathcal{T}_0, \mathcal{T}_1)$ , where both  $\mathcal{T}_0, \mathcal{T}_1 \in \mathcal{N}^{\mathcal{D}}$ , with the constraint that

$$\mathcal{T}_0_j \cap \mathcal{T}_1_j = \emptyset \quad (20)$$

NAME	DESCRIPTION	STRUCTURING ELEMENT	EQUIVALENT TO
NOP ( $L_1$ )	No OPeration	$\begin{matrix} - & - & - \\ - & 1 & - \\ - & - & - \end{matrix}$	$(\mathcal{K}_{L_1})$
INV ( $L_1$ )	INVersion	$\begin{matrix} - & - & - \\ - & 0 & - \\ - & - & - \end{matrix}$	$(\mathcal{K}_{L_1})^c$
NMOV ( $L_1$ )	North MOVE	$\begin{matrix} - & - & - \\ - & - & - \\ - & 1 & - \end{matrix}$	$(\mathcal{K}_{L_1})_{\{0,-1\}}$
SMOV ( $L_1$ )	South MOVE	$\begin{matrix} - & 1 & - \\ - & - & - \\ - & - & - \end{matrix}$	$(\mathcal{K}_{L_1})_{\{0,1\}}$
WMOV ( $L_1$ )	West MOVE	$\begin{matrix} - & - & - \\ - & - & 1 \\ - & - & - \end{matrix}$	$(\mathcal{K}_{L_1})_{\{-1,0\}}$
EMOV ( $L_1$ )	East MOVE	$\begin{matrix} - & - & - \\ 1 & - & - \\ - & - & - \end{matrix}$	$(\mathcal{K}_{L_1})_{\{1,0\}}$
EXP ( $L_1$ )	EXPansion	$\begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}^c$	$(\mathcal{K}_{L_1}) \oplus \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$
VEXP ( $L_1$ )	Vertical EXPansion	$\begin{matrix} - & 0 & - \\ - & 0 & - \\ - & 0 & - \end{matrix}^c$	$(\mathcal{K}_{L_1}) \oplus \begin{matrix} - & 1 & - \\ - & 1 & - \\ - & 1 & - \end{matrix}$
HEXP ( $L_1$ )	Horizontal EXPansion	$\begin{matrix} - & - & - \\ 0 & 0 & 0 \\ - & - & - \end{matrix}^c$	$(\mathcal{K}_{L_1}) \oplus \begin{matrix} - & 1 & 1 \\ - & 1 & 1 \\ - & 1 & 1 \end{matrix}$
NEEXP ( $L_1$ )	NorthEast EXPansion	$\begin{matrix} - & - & - \\ 0 & 0 & - \\ 0 & 0 & - \end{matrix}^c$	$(\mathcal{K}_{L_1}) \oplus \begin{matrix} - & 1 & 1 \\ - & 1 & 1 \\ - & 1 & 1 \end{matrix}$
ERS ( $L_1$ )	ERoSion	$\begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$	$(\mathcal{K}_{L_1}) \ominus \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$
VERS ( $L_1$ )	Vertical ERoSion	$\begin{matrix} - & 1 & - \\ - & 1 & - \\ - & 1 & - \end{matrix}$	$(\mathcal{K}_{L_1}) \ominus \begin{matrix} - & 1 & - \\ - & 1 & - \\ - & 1 & - \end{matrix}$
HERS ( $L_1$ )	Horizontal ERoSion	$\begin{matrix} - & - & - \\ 1 & 1 & 1 \\ - & - & - \end{matrix}$	$(\mathcal{K}_{L_1}) \ominus \begin{matrix} - & - & - \\ 1 & 1 & 1 \\ - & - & - \end{matrix}$
NEERS ( $L_1$ )	NorthEast ERoSion	$\begin{matrix} - & 1 & 1 \\ - & 1 & 1 \\ - & 1 & 1 \end{matrix}$	$(\mathcal{K}_{L_1}) \ominus \begin{matrix} - & 1 & 1 \\ - & 1 & 1 \\ - & 1 & 1 \end{matrix}$
BOR ( $L_1$ )	BORder	$\begin{matrix} - & - & - \\ 8 & 1 & 0 \\ - & - & - \end{matrix}$	-
LS2 ( $L_1$ )	LesS than 2	$\left( \begin{matrix} 1 & 1 & - \\ 4 & 1 & 1 \\ 1 & 1 & - \end{matrix}, \begin{matrix} 1 & 1 & - \\ 4 & 1 & 1 \\ - & - & 0 \end{matrix}, \begin{matrix} - & - & - \\ - & 0 & - \\ - & - & - \end{matrix} \right)^c$	-

Table 1: List of PAPRICA Graphic Operators  $G(L_1)$

for every  $\alpha \in [1 \dots \mathcal{D}]$  and  $j \in [1 \dots \mathcal{C}^\alpha]$ . If  $A \in \mathcal{N}^{\mathcal{D}}$  is a data hierarchy, an *elementary hierarchy matching* with a simple hierarchy template  $T$  is a  $\mathcal{N}^{\mathcal{D}} \mapsto \psi_{\mathcal{F}_N}$  set transform:

$$A \otimes T \triangleq \left\{ y \in \mathcal{F}_N \mid (y + t_{1_j}^\alpha) \in A_j^\alpha \text{ and } (y + t_{0_j}^\alpha) \notin A_j^\alpha, \right. \\ \left. \text{for every } \alpha \in [1 \dots \mathcal{D}], j \in [1 \dots \mathcal{C}^\alpha], t_{1_j}^\alpha \in \mathcal{T}_{1_j}^\alpha, t_{0_j}^\alpha \in \mathcal{T}_{0_j}^\alpha \right\}, \quad (21)$$

while a *composite hierarchy matching* is entirely defined by a *matching hierarchies list*  $\mathcal{T}_{\mathcal{C}} = \{T^1, \dots, T^k, \dots\}$ :

$$A \otimes \mathcal{T}_{\mathcal{C}} \triangleq \bigcup_k (A \otimes T^k). \quad (22)$$

Obviously morphological and matching operators are a subset of structure operators.

### 3 External Architecture

The instruction set of PAPRICA architecture can be partitioned into three different classes, namely:

- *Elementary instructions*, to perform logical and morphological operations; executed within a single clock cycle by every PE of the array.
- *Mapping statements* to implement the virtualization of the “processor array”  $\mathcal{K}$ .
- *Control flow* statements.

#### 3.1 Elementary instruction set

Each PAPRICA instruction is a structure matching with a structure template built from the cascade of a *graphic operator*  $G(\cdot)$  and a *logic operator*  $*$ . A PAPRICA instruction in assembly format is:

$$L_D = G(L_{S1}) * L_{S2} [\%A] \quad (23)$$

where  $L_D$ ,  $L_{S1}$  and  $L_{S2} \in [0 \dots 63]$  are three layer identifiers (Destination, Source1, and Source2). This specific *structure-matching* operates on  $\mathcal{K}_{L_{S1}}$  and  $\mathcal{K}_{L_{S2}}$  and stores the result into  $\mathcal{K}_{L_D}$ .

The graphic operator  $G$  is one of the sixteen composite matching operators listed in table 1, while the logical operator ‘\*’ is one of the eight unary, binary or ternary Boolean operators listed in table 2. The optional switch  $\%A$  introduces an additional operation

$$\mathcal{K}_0 = \mathcal{K}_0 \cup \mathcal{K}_{L_D} \quad (24)$$

which is useful in many circumstances (e.g. in CAD tools to accumulate errors). This operation will not be compatible with the constraint (26) introduced in next section.

### 3.2 Mapping Statements

PAPRICA uses a specific construct  $U$  called *Update Block* which is a tuple of either 16, 32, 48 or 64 composite structure matchings  $(\mathcal{R}_{L_0}, \mathcal{R}_{L_1}, \dots, \mathcal{R}_{L_{63}})$ . In practice an Update Block is a  $\mathcal{K} \mapsto \mathcal{K}$  set transform defined as

$$U(\mathcal{I}) \triangleq (\mathcal{I} \circ \mathcal{R}_{L_0}, \mathcal{I} \circ \mathcal{R}_{L_1}, \dots, \mathcal{I} \circ \mathcal{R}_{L_{63}}) \quad (25)$$

where  $\mathcal{I} \in \mathcal{K}$ . If an Update Block satisfies the following semantic constraint:

$$U(\mathcal{I}) = U(U(\mathcal{I})), \quad (26)$$

then operations on larger data structures  $\mathcal{K}'$  can be “virtualized” with the technique described in sect. 3.2.3, by splitting an image  $\mathcal{I}$  in a set of adjacent *windows*  $\mathcal{I}^k$  and by sequentially applying the same Update Block  $U(\cdot)$  to all the windows  $\mathcal{I}^k$ . Provided that constraint (26) is satisfied, the result is independent of the size of the Processor Array (PA)  $\mathcal{K}$  (not proven here).

Each matching list  $\mathcal{R}_L$ , of the Update Block  $U$  is made of a sequence of several simpler operators taken from the *instruction set*.

#### 3.2.1 Limit and Validity Areas

Because of the presence of “borders” in the frame  $\mathcal{F}_N$ , it is intuitive that the image result of a structure matching may be “undefined” along the borders. This paragraph defines more formally the area where results are valid. Some definition is necessary to better understand the formalism:

- an  $N$ -square  $\Gamma_m$  of half-size  $m$  is an image:

$$\Gamma_m \triangleq \{ \mathcal{P} \in \mathcal{F}_N \mid -m \leq \mathcal{P}_i \leq +m \text{ for every } i \in [1 \dots N] \} \quad (27)$$

- the *margin width*  $M(\mathcal{R}_L)$  of a matching structures list  $\mathcal{R}_L$  is the half-size of the smallest  $N$ -square

which fully “contains” all list elements:

$$M(\mathcal{R}_L) \triangleq \min \left\{ m \in Z \mid (\Gamma_m \supset (\mathcal{R}_{0_j}^k \cup \mathcal{R}_{1_j}^k)) \text{ for every } j \in [1 \dots C], k \geq 1 \right\} \quad (28)$$

- the *limit area*  $\mathcal{A}_L$  is the  $N$ -dimensional frame  $\mathcal{A}_L \subseteq \mathcal{F}_N$  in which the original image is defined.
- the *validity area*  $\mathcal{A}_V$  of an Update Block  $U$  is the  $N$ -dimensional frame  $\mathcal{A}_V \subseteq \mathcal{A}_L \subseteq \mathcal{F}_N$  in which the processed image is correct. It is the erosion of the limit area  $\mathcal{A}_L$  by  $\Gamma_{M(U)}$  as structuring element:

$$\mathcal{A}_V = \mathcal{A}_L \ominus \Gamma_{M(U)} \quad (29)$$

#### 3.2.2 Mapping Data Structures to Architectures

So far only the *logical organization*  $\mathcal{M}_L$  of data within a processing system has been considered. There is also a *physical data organization*  $\mathcal{M}_P$  which usually coincides with the traditional uni- (or bi- or tri-) dimensional organization of computer memories (i.e.  $M_X \times M_Y \times M_Z$  words by  $B$  bits).  $M_X$ ,  $M_Y$ ,  $M_Z$  and  $B$  are respectively the maximum memory addresses along the three coordinates and the memory parallelism (i.e. number of bits per word). The physical organization of PAPRICA image memory, as seen from the host computer, has  $B = 16$  bits, while both  $M_X = f_1^1$  and  $M_Y = f_2^1$  must be powers of two and  $M_Z = C_1/16$  is an integer number.

A 3-D *Structure Memory Map*  $\mathcal{O}$  of a  $N$ -dimensional frame  $\mathcal{F}_N$  is a one-to-one  $\mathcal{F}_N \mapsto [0 \dots M_X] \times [0 \dots M_Y] \times [0 \dots M_Z] \times [0 \dots B]$  mapping, such that  $\mathcal{O}(\mathcal{P})$  gives the physical address of pixel  $\mathcal{P}$  in the computer memory (namely, word “coordinates” and bit).

There is also an *Inverse Structure Memory Map*  $\mathcal{O}^{-1}$ , which is obviously a one-to-one  $[0 \dots M_X] \times [0 \dots M_Y] \times [0 \dots M_Z] \times [0 \dots B] \mapsto \mathcal{F}_N$  mapping.

Combining Memory Maps, Inverse Memory Maps and Update Blocks together it is possible to emulate several non-mesh computing architectures, such as pyramids. An example is illustrated in sect. 3.4.1. PAPRICA defines and uses only “simple” Memory Maps, as can be defined by six “skip” parameters. These parameters are modified by means of `SKIP.READ` and `SKIP.WRITE` control instructions. PAPRICA *mapping statements* are listed in table 3.

#### 3.2.3 Virtualization of Large Arrays

Since the size of most “real-world” images (often  $\gg 10^6$  pixels) is much larger than that of a PA of reasonable size (few thousands PEs max.), it is necessary to introduce a mechanism to virtualize the PEs for large images. This mechanism, which is called *UPDATE*, is based on splitting the input image in an array of smaller *windows*, as shown in fig. 2, and requires the presence of an Image Memory of a sufficient size to

NAME	DESCRIPTION	EQUIVALENT TO
(omitted)	No operation	$\mathcal{K}_{LD} = G(\mathcal{K}_{L_1})$
~	NOT	$\mathcal{K}_{LD} = G(\mathcal{K}_{L_1})^c$
&	AND	$\mathcal{K}_{LD} = G(\mathcal{K}_{L_1}) \cap \mathcal{K}_{L_2}$
&~	ANDNOT	$\mathcal{K}_{LD} = G(\mathcal{K}_{L_1}) \cap (\mathcal{K}_{L_2}^c)$
	OR	$\mathcal{K}_{LD} = G(\mathcal{K}_{L_1}) \cup \mathcal{K}_{L_2}$
~	ORNOT	$\mathcal{K}_{LD} = G(\mathcal{K}_{L_1}) \cup (\mathcal{K}_{L_2}^c)$
^	EXOR	$\mathcal{K}_{LD} = (G(\mathcal{K}_{L_1}) \cap (\mathcal{K}_{L_2}^c)) \cup (G(\mathcal{K}_{L_1})^c \cap \mathcal{K}_{L_2})$
+	PLUS (Arithm. Plus)	$\mathcal{K}_{LD} =$ "arithmetic sum" of $G(\mathcal{K}_{L_1}) + \mathcal{K}_{L_2} + \mathcal{K}_0$ $\mathcal{K}_0 =$ "carry out" of sum of $G(\mathcal{K}_{L_1}) + \mathcal{K}_{L_2} + \mathcal{K}_0$

Table 2: List of PAPRICA Logic Operators \*

STATEMENT	MEANING
MEMORG <SizX> <SizY> <Par>	Sets frame sizes $f_1^2$ and $f_2^2$ and parallelism $C_2$ of second level of PAPRICA hierarchy ( $\alpha = 2$ ).
SKIP.READ <SkX> <SkY> <SkL>	Sets 3 of the parameters which define memory mapping.
SKIP.WRITE <SkX> <SkY> <SkL>	Sets the other 3 parameters which define memory mapping.
VALIDITY <Area>	Sets <i>validity area</i> to <Area> for the following Update Blocks.
LIMIT <Area>	Sets <i>limit area</i> $\mathcal{A}_L$ to <Area> for the following Update Blocks.
MARGIN <M>	Sets <i>margin width</i> to <M> for the Update Blocks which follow.
SETDEF <Base>	Sets base address for <i>based</i> addressing mode.

Table 3: List of PAPRICA mapping statements.

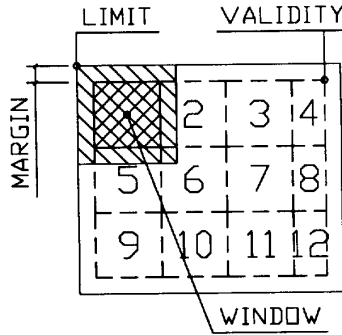


Figure 2: UPDATE mechanism and area of validity

store the whole image to be processed. A PAPRICA program must contain a list of UPDATE instructions (either *explicit* or *implicit*) placed at specific places, according to a set of simple semantic rules. The controller first loads into the actual PA the pixels from the first window (see fig. 2), then it executes the program up to the next UPDATE instruction and finally it stores the results back into the Image Memory. These operations are repeated for all the windows until the image is completely scanned, and for all the Update blocks.

The effect of these steps is to apply the list of instructions between two consecutive UPDATES to all the pixels of the image, as if they would all fit into a large "virtual" PA. The only limitation is obviously given by the total size of the Image Memory.

### 3.3 Flow Control Statement

PAPRICA *flow control statements* are listed in table 4. During processing, three *global flags* are computed for each Update Block, which are set according to the result of the last operator of that block. The *SET*, *RESET* and *NOCHANGE* flags are respectively set *iff* (for the last operator of that block)  $\mathcal{K}_{LD} = \mathcal{A}_V$ ,  $\mathcal{K}_{LD} = \emptyset$  or  $\mathcal{K}_{LD} = \mathcal{K}_{LD}^*$ , where  $\mathcal{K}_{LD}^*$  is the value of  $\mathcal{K}_{LD}$  "before" the operator has been computed.

A *PAPRICA Program*  $\mathcal{P}$  is a tuple of hierarchical operators implemented as the cascade of one or more Update Blocks  $\mathcal{U}_r$ , some *control instructions* and optionally one or more Mappings  $\mathcal{O}_i$ , which concur to perform a desired function. Although being made of Update Blocks, a PAPRICA Program usually does not satisfy the semantic constraint of the individual Update Blocks (see formula (26)).

### 3.4 Advantages of the Specific Virtualization Mechanism: Examples

The following subsections describe two of the main advantages of the window-based virtualization mechanism implemented on PAPRICA system.

#### 3.4.1 Mapping Pyramidal Architectures

Pyramidal architectures have shown several advantages in the field of image processing [17]. One major drawback of such architectures is that they are not easily scaled beyond a certain size. This is because interconnections among processors have an intrinsic 3-

STATEMENT	MEANING
UPDATE	Separates two Update Blocks. All the other mapping and flow control instructions are also <i>implicit updates</i> which separate Update Blocks.
FOR <iter> instructions	Repeats <i>instructions</i> <iter> times.
ENDFOR	
IF [NOT] <flag> instructions	Executes this program segment if <flag> is [NOT] set.
ELSE instructions	Otherwise executes this program segment.
ENDIF	
REPEAT instructions	Repeats <i>instructions</i> until <flag> is [NOT] set.
UNTIL [NOT] <flag>	
CALL <address>	Calls a subroutine at <address>.
RET	Returns from subroutine.
JUMP <address>	Jumps Program Counter to <address>.

Table 4: List of PAPRICA flow control instructions.

D structure (see fig. 3.a) which causes problems when mapped onto a 2-D silicon surface.

PAPRICA solves the problem of interconnections because its hardware allows dynamic mapping of limit and validity areas (see sect. 3.2.1). Using skip factors different from one, it is possible to map several pyramidal architectures such as for instance the “4 children per parent” topology shown in fig. 3.a. Fig. 3.b shows how it can be mapped on PAPRICA Image Memory.

Each pyramid processor has its own data memory, which is mapped onto a specific region of PAPRICA Image Memory. The bottom level of the pyramid (i.e. the collection of the leaves of the pyramidal tree) is composed of  $C_P^{L-1}$  children processors (where  $L$  and  $C_P$  are respectively the number of levels and the number of children per parent). Each pyramid processor has access to a total of  $N_{Px} \times N_{Py}$  image pixels, which are grouped close to each other in a rectangular area ( $A$ ) of PAPRICA Image Memory. The  $C_P^{L-2}$  processors of the second-last level are associated to a smaller rectangular area in the PAPRICA memory ( $B$ ). The processors of all the levels above, up to the root processor, are associated to smaller and smaller areas ( $C$ , etc.).

It is now clear that the only limitation to the size of the pyramid is given by the total amount of Image Memory available, while there is no limitation to the complexity of the interconnections among the pyramid processors, which can indifferently be connected to their brothers, their parents and their children.

Fig. 3.c represents the PAPRICA code necessary to emulate such a pyramidal structure. At the beginning, the program elaborates the lowest level of the pyramid, e.g. performing initial filtering of input data. This is done by having LIMIT and VALIDITY areas coincident with input data. Then, VALIDITY and SKIP are changed respectively to point to area  $B$  and to obtain the amount of decimation necessary (e.g.  $2 \times 2$ ), so that the output data of the lowest level are used as

input for the computations of the second-last one.

This process is iterated, each time changing LIMIT, VALIDITY and SKIP, up to the root level. It has to be noted that processing on different levels can be completely different in nature, and that merely changing the above mentioned parameters it is possible to change the type of logic connections mapped by the system. In any case, the overhead introduced to emulate pyramidal structures is very small because the host processor does not have to reorganize or move data in memory to process the different levels.

### 3.4.2 Implementing a Multiple Focus of Attention

Sometimes, after the loading of an image sub-window into the PA, the elaboration of the specific sub-window may be useless, and thus the operation of storing back the results into the image memory wouldn't be required. This fact allows to decrease the number of computations, skipping the sub-windows which won't produce significant results, and implementing a sort of a multiple “focus of attention” [18, 14].

The choice whether to perform the elaboration or not depends on the characteristics of the complete set of data loaded into the PA internal registers. The planned extension to PAPRICA controller is based on the consideration that each 16-bit element passes through the 16-bit bus that links the Image Memory and the PA, and can be captured by an additional circuit. Only a set of bits of the captured values are kept, thanks to a logical intersection operation (*AND*) with a programmable mask; then the result and the value contained into the register are sent into an ALU. The operation performed by the ALU can be selected within the following set: {*OR, AND, XOR, NOR, NAND, EQU, MAX, MIN*}, while the result is stored back again into the register. When a new image sub-window is loaded into the PA, the register is set or

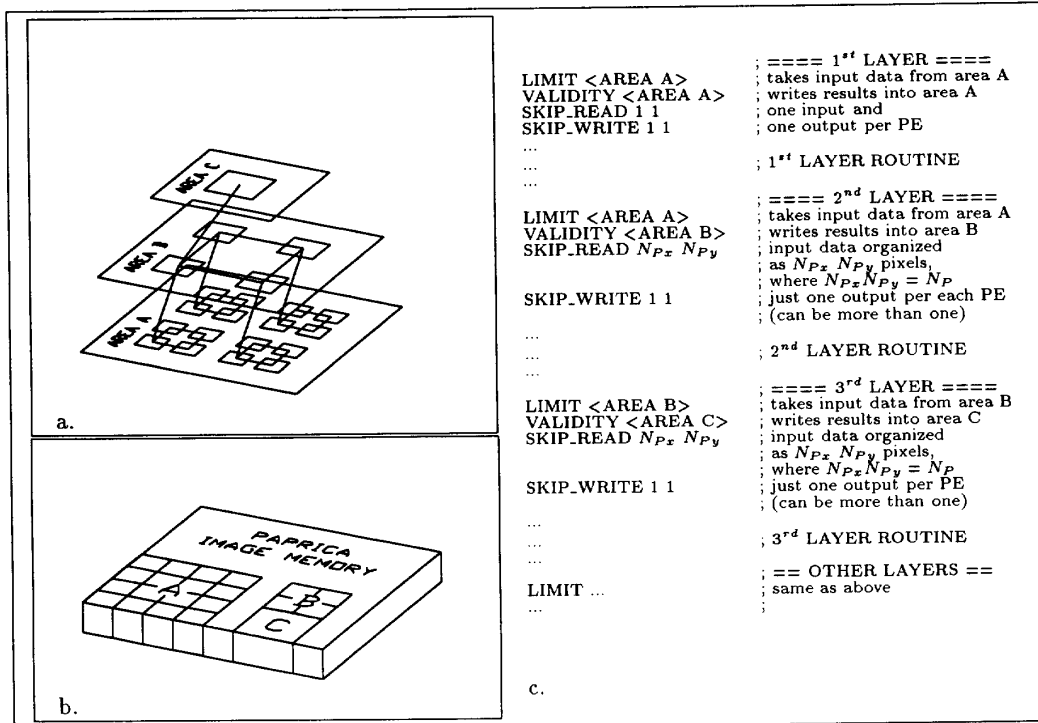


Figure 3: Mapping of a Pyramidal Architecture on PAPRICA

reset according to a specific programming input.

Finally, the elaboration and the consequent storing of the results can be conditioned to a particular bit (or set of bits) of the register.

The processing obtained in this case is substantially pyramidal: a single value is computed for each image sub-window loaded into the PA; the set of these values forms a coarse resolution mask. Then, the particular value associated to each sub-window selects the different fine-grain processing that will be performed by the PA. The extension of PAPRICA controller can be useful both to speed-up the elaboration, processing only the relevant areas of the image or implementing different elaborations with a high efficiency, and to implement functions that otherwise couldn't be implemented, such as the global computation of the maximum value in a  $n$ -bit image [4].

#### 4 Internal Architecture

The complete PAPRICA system, shown in fig. 4, is composed of a double-port memory board with a VME bus interface which contains the program and data memories (up to 8 Mbytes of static RAM), the bus arbitration logic and drivers and by a *piggy-back* board, shown in fig. 5, which hosts the PA and the

Controller.

The two boards interface using two buses, one connected to the program memory and another towards the data memory. The partitioning of the total memory in two different segments is software programmable. The memory board is rather conventional and is not described further; we will focus on the design of the processor board and on the implementation choices which have been made for the different blocks.

##### 4.1 The Processor Array

For the PA, composed of a number of identical functional units replicated over a bidimensional grid, the choice of a full custom implementation was straightforward in order to integrate the maximum number of PEs into a single die. The number of the PEs in a single chip has been chosen as a compromise between the feasible die area, the corresponding cost and yield requirements, the design complexity and the power supply requirements. Although with the current technology a design with  $8 \times 8$  PEs would have fit into a 1 cm by 1 cm die, it has been preferred to integrate into a chip only an array of  $4 \times 4$  PEs, whose photograph is shown in fig. 6. The chip has been designed and fabricated using a  $1.5 \mu\text{m}$  CMOS technology, with an area of approximately  $45 \text{ mm}^2$ . It is housed in a 84 pin PGA package and the measured yield is approxi-

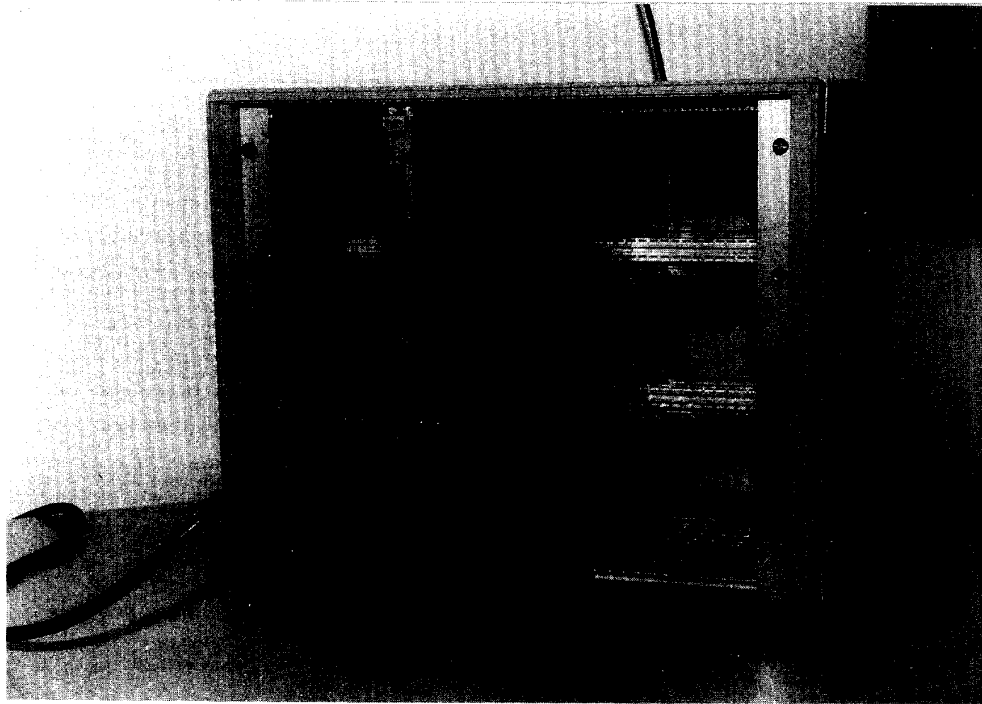


Figure 4: Photograph of the PAPRICA cabinet

mately 65% over a set of 120 samples. The complete PA is composed of a square matrix of  $4 \times 4$  chips for a total of 256 PEs (in a  $16 \times 16$  configuration).

The PEs are physically grouped into four columns, where the elements of the same column share a common data bus used either for memory operations (in memory mode) or for broadcasting instruction codes to all PEs (in program mode). The column busses are then connected together via a set of four blocks containing drivers, sense amplifiers and multiplexer circuits. This reduces the capacitive load on the interconnection lines and the delays associated to communications particularly for internal memory read operations during which the lines are driven by the minimum size transistor of the elementary cells. Multiplexing the internal busses between memory data and instruction codes, although requires multiplexer circuits, reduces the total I/O pin count (by 20 pins) and the total chip area required for routing (roughly 30% reduction).

In memory mode, the individual words of the RAM are selected by means of an ADDRESS BUS which is routed directly to all PEs in the array, while the data to be read or written passes through the BUS MUX blocks, which operate either as drivers or as sense amplifiers. In program mode, the instruction is broadcast to all the PEs by means of the vertical busses and the BUS MUX blocks, which operate as drivers.

#### 4.1.1 The Register File

The register file must be accessed during the execution of each instruction for operand fetching and storing of

the results with 1 bit parallelism. When data have to be transferred to/from the host memory a 16 bit parallelism is used as a compromise between the number of I/O pins, the internal routing, the interconnection complexity and the information transfer throughput. Therefore the register file of each PE is *internally* seen as 64 individual bits and from *outside* as four 16-bit words.

#### 4.1.2 The Execution Unit

Since the basic choice for PAPRICA has been to execute all instructions in a single cycle, microprogrammed solutions were discarded; instruction execution is performed by two combinatorial blocks corresponding respectively to the Graphic and Logic Operators. The inputs to these blocks are the two operands  $L_{S1}$  and  $L_{S2}$  from the PE register file, the 8 GOP operands from the neighbors and the instruction opcode. In order to reduce the area requirements these blocks have been implemented as semistatic PLAs with a timing optimized to reduce the power consumption. This choice has an obvious counterpart in the dynamic characteristics giving an increase in the delays and is currently being revised in the second version of the system.

#### 4.2 The Controller

The two main tasks of the Controller are rather independent and functionally have been assigned to two different blocks, namely the *Window Unit* (responsible for the UPDATE operation) and the *Instruction Unit*

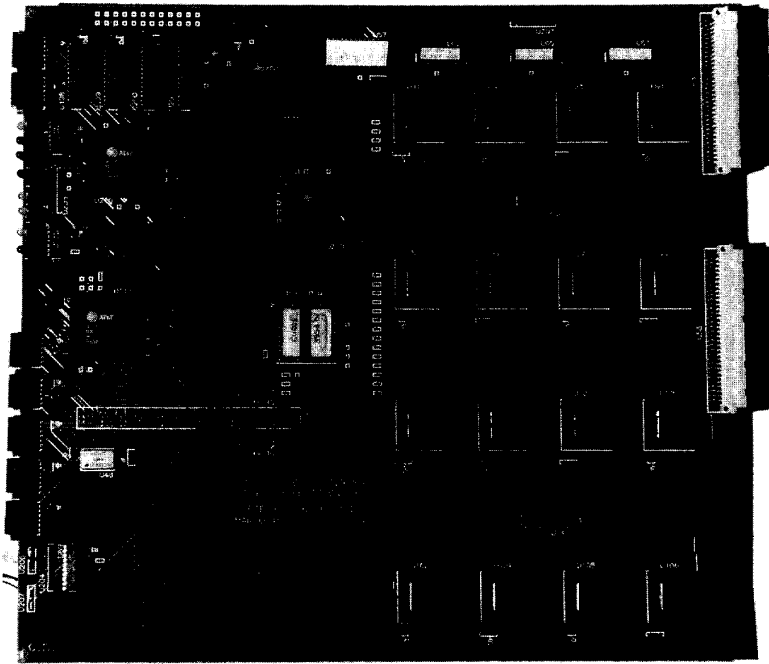


Figure 5: Photograph of the VME board hosting  $4 \times 4$  processors chips

(responsible for the execution of the Control Instructions).

#### 4.2.1 Window Unit

The main task of this unit is to sequentially generate the image memory addresses corresponding to a given subwindow and the PE addresses to/from which data are to be transferred. At the end of each subwindow the control is passed to the Instruction Unit to execute the instructions. From the functional point of view the task of the unit is simple and repetitive and its complexity arises from three facts:

- It has been decided to make this unit fully programmable with respect to X size, Y size, pixel depth of the image and the number of PEs.
- It is necessary to separate the address generation in two different sections, one for *write* operations (transfers from the array to the memory) and one for *read* operations, in order to modify the mapping of the image to the array with the SKIP instruction.
- The partitioning of the image into sub-windows does not depend only on the image and the PA size, but also on the MARGIN parameter, which may vary according to different programs or possibly also within the same program and must be programmable in the Window Unit.

Although composed of a number of counters, comparators and combinatorial logic, the Window Unit has come out as the most complex single component

of the system. It was therefore decided to implement it as a standard cell circuit in the same technology as the PEs giving a  $30 \text{ mm}^2$  chip with 110 I/O pins and a complexity of approximately 20.000 transistors.

#### 4.2.2 Instruction Unit

The Instruction Unit is a rather conventional sequencer; its main components are:

- a 32-bit INSTRUCTION REGISTER (IR) which contains the last fetched instruction. *Control* instructions are executed by the unit while *array* instructions are transferred to the PA.
- a 6-bit OFFSET REGISTER which contains the offset to be added to the operand addresses of an array instruction when relative addressing is used.
- a 24-bit PROGRAM COUNTER (PC) which contains the address in the Program Memory of the next instruction.
- an 8-bit STACK POINTER (SP)
- A 32-bit STACK to save the PC and internal parameters, such as the address of the instruction following the last UPDATE on a subroutine call or in nested FOR . . . NEXT loops.
- a 8-bit FOR COUNTER as loop counter for FOR loops.

From the functional point of view both the Window and the Instruction Units are operating units. Their control and timing, together with that of the array, is provided by a Finite State Machine (FSM), which may provide 4 different timing and control functions:

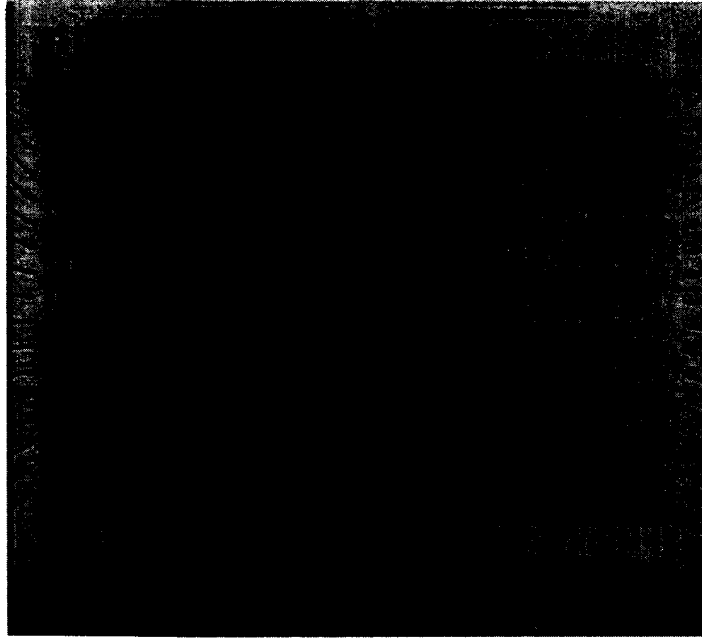


Figure 6: Microphotograph of the processor chip

- Transfer of one window to and from the array;
- Execution of a control or array instruction;

Since the Instruction Unit and the FSM are the two system components which *link* all the others, they are the most sensitive to changes, modifications and upgrades. It was therefore decided not to integrate them but to implement them with reprogrammable gate arrays. The complete unit fits into 4 XILINX 3042 components for an approximate equivalent of 12000 logic gates.

## 5 The Programming Environment

The programming environment designed for PAPRICA enables writing application programs starting from standard "C" language. The environment has been conceived in order to isolate applications from hardware implementation details. In fact, the same program can run on different platforms:

- the PAPRICA hardware;
- a software simulator running on either UNIX workstations or MS-DOS personal computers;
- a software simulator running on a Connection Machine CM2.

A block diagram showing the programming environment is depicted in fig.7. The availability of a complete software emulator at system level allowed to start developing applications before the completion of the hardware with only an obvious speed penalty, removed by the use of the Connection Machine.

Applications are written using "C" language with the help of two sets of functions:

- the *System Library*: a standard set of functions managing the interaction between the application and either the hardware or the emulator;
- the *Macro Library*: an open set of functions, that can be augmented by the user, for the automatic generation of PAPRICA Assembly code. These functions parametrically build segments of PAPRICA code to perform a specific task.

The *Debugger* is an example of application written with direct calls to the system library. This program provides a friendly environment to execute and debug code written in PAPRICA assembly language.

### 5.1 System Library

The system library contains functions to standardize two levels of interaction between applications and the system. Two sets of functions are defined: the first one, named *PAPRICA*, groups all the accesses to the coprocessor board, while the second one, named *PGI* (for PAPRICA Graphic Interface), provides an easy and standard way for application programs to present graphical image data on the workstation screen.

PAPRICA routines provide the same functionalities, independent of the environment on which the PAPRICA code is executed (the hardware or the emulators on workstations, CM2, MS-DOS). The same is also valid for the PGI set, in relation with the different hardware platforms (X-Window systems, MS-DOS).

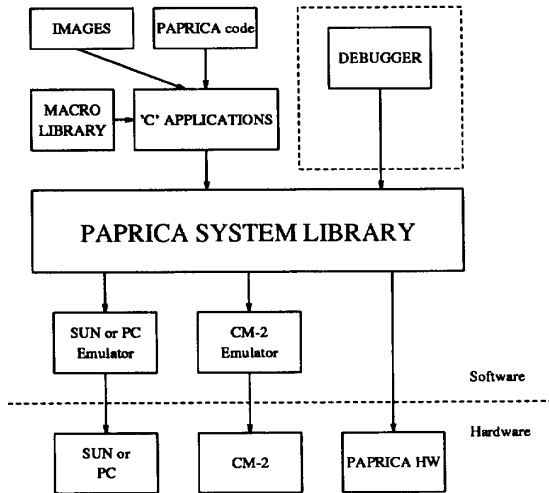


Figure 7: PAPRICA Operating Environment

### 5.1.1 PAPRICA Routines

PAPRICA routines can be subdivided in three different groups according to functionality.

- **Program Management**, including operations for the loading/unloading of segments of code between the host's memory and PAPRICA Program Memory. During host-to-PAPRICA transfers instructions are automatically translated by an embedded assembler into executable code. Conversely, in the opposite direction it is possible to have automatic translation to assembly language format (embedded disassembler) or transparent transfer of executable code.
- **Image Management**. In general, PAPRICA Image Memory is accessed by referring to an *image descriptor* which defines size, depth and position of the image within the Image Memory. These functions allow to handle several file formats, including GIF, PGM, JPEG, TIFF, CIF 2.0, MAGIC (from Berkeley University, used for VLSI CAD tools), internal format (a mere memory dump), and also to allocate and deallocate layers. This is very important when generating PAPRICA programs automatically from a higher level language, because it provides a kind of dynamic memory management.
- **System Control**, for the setup and initialization of the PAPRICA system. Moreover, they allow to set or reset the single step execution mode.

### 5.1.2 PGI Routines

The PGI set of routines, developed using the X11 windowing package, implements a graphical interface that can be conveniently used by application programs to

present in an easy and standard way images and textual information on the host display. Another version of these routines was written for MS-DOS to allow the use of the system on Personal Computers.

Due to the different meanings that a pixel can assume depending on the application, three solutions have been implemented:

- grey scale mapping: used in case of natural black and white images;
- absolute color mapping: used for VLSI CAD applications;
- numeric display: used for numeric applications.

## 5.2 Macro Library

A flexible solution to the problem of application development on PAPRICA system is the creation of a library of "C" routines that parametrically build segments of PAPRICA code to perform commonly used functions. The routines in the macro library directly add the generated code to the Program Memory at run time. This solution was preferred to the creation of assembly or executable code to be later loaded from a file, because the code can be generated conditionally to user inputs or run-time data.

If the library is wide enough, it is possible to write applications that, as far as the PAPRICA system is concerned, are limited to calls to the macro library and to the other interface functions. This library has already proved to be a very effective tool. It is anyway possible to call directly a PAPRICA program written in assembly language using a function defined in the system library, when writing a Macro function appears to be cumbersome.

Actually, the available routines cover, among the others, the following fields:

- setup of the PAPRICA system for a given memory organization (image descriptor).
- basic operations for VLSI design rule check (DRC), like minimum or maximum width, minimum distance or overlap, for any layer combination and dimension.
- filters for grey scale pictures processing.
- arithmetic operations on integers and floating point numbers (sum, subtraction, multiplication and division).

## 5.3 Program Example

Fig. 8 presents a small application that reads an image, processes it using a  $3 \times 3$  square low-pass filter, displays the results on a graphic window and saves them to an output file. It shows that the use of System and Macro libraries makes the writing of application programs an easy task, affordable by a normal "C" programmer.

```

#include "paprica.h"           /* header file for PAPRICA routines */
#include "pgi.h"              /* header file for PGI routines */
...
pap_image img_descr,result;  /* image descriptors used by the program */

main()
{
    pap_init(...);          /* PAPRICA hw or emulator is initialized */
    pap_imgsize(&img_descr); /* Image Memory configuration is saved to image descriptor */
    img_descr.size.lay=8;    /* image is 256 grey levels */
    pap_lalloc(&img_descr, PAP_NEWLAYER); /* 8 layers are allocated to image */
    pap_file2img("input.tif", PAP_TIFF, &img_descr); /* input image is read from file in TIFF format */
    sqlpfil(3,&img_descr, &result); /* filter program is created in PAPRICA Program Memory using
    a Macro Library function (3x3 square low pass filter) */
    pap_start();            /* PAPRICA program is executed */
    pap_waitend();         /* host waits for end of program */
    pgi_opengwin(...);    /* opens a graphic window */
    pgi_img2gwin(&result,...); /* output image is displayed in gr. window */
    pap_img2file("output.tif", PAP_TIFF, &result); /* and saved in TIFF format */
    pap_close();           /* PAPRICA hw or emulator is deallocated */
}

```

Figure 8: An example involving the use of System and Macro Libraries

## 6 Evolution and perspectives

Since the first prototype implementation, PAPRICA system has undergone a series of revision stages and a thorough investigation of a number of architectural modifications to overcome its limitations. Basically the feasibility studies have outlined the following three main possible solutions.

- The first one is to maintain the same architecture of the original prototype and, taking advantage of the technological evolution and of the possible optimizations in the chip layout, to increase the size of the array. If the size becomes of the same order of magnitude of the images to be processed, the loading/unloading overhead due to the overlapping of windows required by the use of morphological operators would be reduced.
- The second one is to increase the addressing space of each elementary processor by the use of external memory elements. This choice logically simplifies the PE virtualization mechanism but limits the array size and presents additional implementation problems due to morphological operators.
- The third one is to change the physical interconnection topology of the processing elements from a two-dimensional mesh to a linear array moving from a *window* based to a *scanline* based processor virtualization.

These solutions are presented and discussed in detail in [10]. The most promising solution, the third one, is presented in the following subsection. Table 5 shows the performance of the current prototype ( $16 \times 16 = 256$  PEs, 500 ns instruction cycle time, 250 ns read/write time access memory) with the aim of comparing them to the ones that can be achieved with the following architectural evolution proposal.

Operations	Cycles	Neighborhood	Processing Speed [Mpixel/s]
Plus, Minus	$\approx 10$	$1 \times 1$	1.92
Maximum, Minimum	$\approx 70$	$1 \times 1$	1.02
$x$ and $y$ Derivatives	$\approx 80$	$3 \times 3$	1.16
Average	$\approx 90$	$3 \times 3$	1.13
Kirsh Gradient	$\approx 450$	$3 \times 3$	0.55
Prewitt Gradient	$\approx 450$	$3 \times 3$	0.55
$8 \times 8$ Gradient	$\approx 490$	$3 \times 3$	0.52
Binary Thinning (1 iter.)	$\approx 50$	$17 \times 17$	0.27
Clustering (1 iter.)	$\approx 2000$	$5 \times 5$	0.12

Table 5: Performance of the current version of PAPRICA using 8-bit images

### 6.1 Architectural Evolution

The PE virtualization mechanism is simplified if one of the image dimensions can be entirely loaded into the array, because in this case it is necessary only to scan the image in the other dimension. This is the basic reason underlying the idea to investigate a new architecture based on a linear array of PEs, big enough to process a complete image line in one iteration [15]. The linear architecture also better matches applications in real-time vision systems, where processing speed and data latency are the most critical factors. The following characteristics must be considered to optimize the low-level processor:

- data come serially from a sensor which generally scans the image by rows, so data become available one row at a time;

- image dimensions are fixed and are dictated by the resolution of the sensor;
- the results of low-level processing must be further elaborated by other stages managing higher level tasks.

In this environment, a one-dimensional array is the best solution to minimize data latency, because a row can be processed as soon as the next few lines are acquired by the sensor. Furthermore, the image width in such systems normally range from 128 to 1024 pixels, making perfectly possible to match the width of the hardware array to that of the image.

The proposed architecture is depicted in Fig. 9. The array chip will contain from 64 up to 128 PEs and the memory necessary to keep 64 binary layers of five image lines to handle the neighborhood (which can be extended to 12 pixels). An external large memory will contain the rest of the image, addressed by line and binary layer. To overcome the limits on the Image Memory-to-PA communication bandwidth, the width of the image memory data bus will be the same as the number of PEs. This allows the transfer of a complete line of image data in one memory cycle. This is feasible because, due to the linear organization, very few pins are needed for neighbors propagation between the different array chips, so it is possible to assign a considerable amount of pins to the Image Memory-to-PA communications. With this architecture, the operations that are necessary to process image line  $n$  are the following:

- the data contained in the internal array line memory are shifted in south-north direction by one position, while image line  $n - 2$  is discarded;
- input binary layers for line  $n + 2$  are loaded from the external memory as south neighbors.
- the program block is executed for line  $n$ ;
- output binary layers for line  $n$  are saved to external memory.

For real-time vision systems, it is possible to include in the array chip an input shift register to be able to acquire directly the image lines from the sensor, without having to store them previously into the Image Memory via external devices (host computer or dedicated hardware).

As for the other architectures, the same instruction block will have to be iterated for all the image lines. One possibility that is currently under evaluation is the insertion in the array chip of a program cache, to avoid reloading of the same code from external (relatively slow) memory as many times as the image lines.

In summary, the advantage of this architecture over the one depicted in the previous section is the great simplification of the memory organization without performance degradation, at least when maximum sized images are processed: it should be noted in fact that, if it is necessary to deal with very small images, a lot of PEs will remain unused with a loss of performance compared with a square architecture with the same number of PEs.

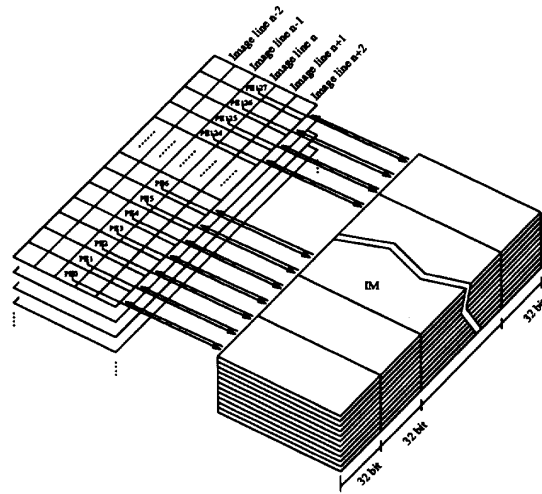


Figure 9: Linear array architecture

### 6.1.1 Performance Analysis

In this section we will compare the large array solution with internal memory, corresponding to the architecture of the current prototype, and the linear array.

For the linear array the expected performance figures for a number of basic image processing algorithms are reported in Table 6 assuming an array of 128 PEs with a 100 ns instruction cycle time, and a 100 ns read/write time access memory. The table reports separately the I/O figures and the basic algorithm timings, because a sequence of several operations can be executed on the same data.

I/O	Execution time
Read or Write - 1 bit matrix	$\approx 0.8\text{ns/pixel}$
Read or Write - 8 bit matrix	$\approx 7\text{ns/pixel}$
Algorithm	Execution time
Contouring of B/W image (1 instruction)	$\approx 0.8\text{ns/pixel}$
Skeleton of B/W image	$\approx 100 - 200\text{ns/pixel}$
Pattern matching ( $5 \times 5$ )	$\approx 1.6\text{ns/pixel}$
Matrix summation (8 bit, integer)	$\approx 7\text{ns/pixel}$
$3 \times 3$ pixel average (8 bit)	$\approx 30\text{ns/pixel}$
$5 \times 5$ pixel average (8 bit)	$\approx 70\text{ns/pixel}$
Motion flow (max. $\pm 8$ pixel per direction)	$\approx 0.5\mu\text{s/pixel}$

Table 6: Performances of a 128 PE linear array

If we extrapolate from the performances of the current prototype to a 16k PE implementation, assuming a speedup of 4 in the processor and memory cy-

cle time, we get the data of Table 7 which shows the comparison between a 128 linear array and a  $128 \times 128$  two-dimensional array without external memory (since the figures in the table are obtained assuming an image of  $128 \times 128$  pixels). The table clearly shows

Algorithm	128 PE linear array	16k PE 2D array
Skeleton of B/W image	10 Mpixel/s	64 Mpixel/s
Matrix summation (8 bit, integer)	38 Mpixel/s	488 Mpixel/s
$3 \times 3$ pixel average (8 bit)	24 Mpixel/s	289 Mpixel/s

Table 7: Comparison between a linear and a 2D array

that the performance ratio between the two solutions has a maximum of one order of magnitude for purely arithmetic tasks and a lower value when morphological operations are required. Considering that the ratio of the number of PEs is 128:1, this rough evaluations, which do not take into account the complexity of the external memory addressing and control, show clearly the advantage of the solution with external memory.

## 7 Conclusions

This paper has presented the result of a 6-year project, leading to the design and implementation of a prototype machine with 256 PEs and a comprehensive set of programming and development tools. The considered massively parallel architecture, devoted to the low-level analysis of images, has been described starting from the underlying computational model up to its physical implementation.

The performance limitations of the machine are related to architectural issues caused by budget constraints. Moreover the choice of well assessed technology, conservative engineering solutions such as the use of programmable arrays, and some timing errors in the chip design have led to an implementation which is slower by a factor of 4 than what achievable with the same architectural solution. Anyway, a lot of different image processing applications [1, 3] has already been written for PAPRICA system, showing its efficiency.

## References

- [1] G. Adorni, A. Broggi, G. Conte, and V. D'Andrea. A self-tuning system for real-time Optical Flow detection. In *Proc. IEEE System, Man, and Cybernetics Conference*, Le Toquet, France, October 17-20 1993.
- [2] K. Batcher. Design of a Massively parallel processor. *IEEE Trans. on Computers*, C-29:836-840, 1980.
- [3] A. Broggi. Parallel and Local Feature Extraction: a Real-Time Approach to Road Boundary Detection. *IEEE Trans. on Image Processing*, Feb.1995. In press.
- [4] A. Broggi, S. Mora, and C. Sansoè. Enhancement of a 2D Array Processor for an Efficient Implementation of Visual Perception Tasks. In *Proc. CAMP'93 - Computer Architectures for Machine Perception*, pages 172-178. New Orleans, December 15-17 1993.
- [5] G. Conte, F. Gregoretti, L. Reyneri, and C. Sansoè. PAPRICA: a Parallel Architecture for VLSI CAD. In A.P.Ambler, P.Agrawal, and W.R.Moore, editors, *CAD Accelerators*, pages 177-189. North Holland, Amsterdam, 1991.
- [6] M. Duff. Parallel Processors for Digital Image Processing. In P. Stucki, editor, *Advances in Digital Image Processing*, pages 265-279. Plenum Press, London, 1979.
- [7] T. Fountain. *Processor Arrays: Architectures and applications*. Academic-Press, London, 1987.
- [8] M. Golay. Hexagonal Parallel Pattern Transformations. *IEEE Trans. on Computers*, C-18:733-740, 1969.
- [9] D. Graham and P. Norgren. The Diff3 Analyzer: A Parallel/Serial Golay Image Processor. In M. Onoe, K. Preston, and A. Rosenfeld, editors, *Real Time Medical Image Processing*, pages 163-182. Plenum Press, London, 1980.
- [10] F. Gregoretti, L. M. Reyneri, C. Sansoè, A. Broggi, and G. Conte. The PAPRICA SIMD array: critical reviews and perspectives. In L. Dadda and B. Wah, editors, *Proc. IEEE ASAP'93*, pages 309-320, Venezia, Italy, October 25-27 1993.
- [11] F. Gregoretti, L. M. Reyneri, C. Sansoè, and L. Rigazio. A chip set implementation of a parallel cellular architecture. *Microprocessing and Microprogramming*, 35:417-425, 1992.
- [12] R. M. Haralick, S. R. Sternberg, and X. Zhuang. Image Analysis Using Mathematical Morphology. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 9(4):532-550, 1987.
- [13] B. Kruse. *Design and Implementation of a Picture Processor*. PhD thesis, Univ. Linköping, S, 1977.
- [14] H. Neumann and H. Stiehl. Toward a computational architecture for monocular preattentive segmentation. In R. G.Hartmann and G.Hauske, editors, *Parallel Processing in Neural Systems and Computers*. Elsevier (North Holland), 1990.
- [15] L. A. Schmitt and S. S. Wilson. The AIS-5000 Parallel Processor. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 10(3):320-330, May 1988.
- [16] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, London, 1982.
- [17] S. L. Tanimoto, T. J. Ligoeki, and R. Ling. A Prototype Pyramid Machine for Hierarchical Cellular Logic. In *Parallel Computer Vision*. Academic Press, London, 1987.
- [18] J. M. Wolfe and K. R. Cave. Deploying visual attention: the guided model. In *AI and the eye*, pages 79-103. A.Blake and T.Troscianko, 1990.