Design and Implementation of the Control Structure of the PAPRICA-3 Processor

F. Gregoretti, F. Intini, L. Lavagno, R. Passerone, L. M. Reyneri

Dipartimento di Elettronica
Politecnico di Torino
Torino, ITALY, 10129

Abstract

This paper describes the pipeline architecture designed to control the execution of instructions on the linear array processor PAPRICA-3, which is being developed at the Politecnico di Torino. The main applications of the array processor lay in the area of image processing, image recognition, embedded systems for guidance assistance and the like. Exploitation of this architecture is currently investigated in the area of real-time image processing, a very demanding task in terms of overall performance. Our design is aimed at improving the algorithmic efficiency by taking advantage of a multi-path queue structure which allows different instructions to run simultaneously, and by optimizing particular patterns of instructions which often appear in envisaged application programs.

1 Introduction

Real-time systems for image processing may benefit from Application-Specific Instruction Processors (ASIPs [6]). Such processors play an intermediate role between general-purpose programmable processors, which may not have enough processing power or memory access bandwidth for the application, and Application Specific Integrated Circuits, which may be too expensive and too rigid for rapidly evolving application needs. In this scenario, a processor is designed for a *class* of applications, and its instruction set and architecture are specifically tuned for that class. This allows to simultaneously achieve the required performance, as well as the required degree of programmability. The PAPRICA-3 system [2, 3] is a SIMD massively parallel processor designed for real time image processing applications. The system is composed of multiple instances of the same basic Processor Element (PE) which execute in parallel the same instructions on different pixels. This approach exploits the parallelism intrinsic to the data structure to boost the overall performance. The paper describes the implementation of the control of the processing array and is organized as follows. Section 2 provides a condensed description of the general architecture and Section 3 describes the instruction set. Section 4 and 5 present in detail the pipeline structure and section 6 analyzes its performance. Finally section 7 presents some conclusions.

2 General description

As shown in figure 1, the kernel of the PAPRICA-3 system is based on a linear array of Q identical 1-bit PEs connected to an image memory via a bidirectional Q-bit wise bus. The image memory is organized in

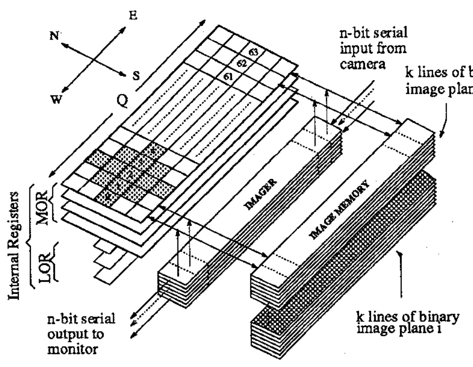
N
E
S
W
Q
n-bit serial input from camera
k lines of binary image plane 0
IMAGER
IMAGE MEMORY
Internal Registers
MOR
LOR
n-bit serial output to monitor
k lines of binary image plane i

Figure 1: **Block diagram of the processor Array**

addressable words whose length matches that of the processor array; each word contains data relative to one binary pixel plane (also called *layer*) of one line of an image, and a single operation is needed to load an entire line of data into the PE's internal registers. The rationale behind this system is that the size of the PE array equals (or is larger than) the height (or width) of the input image. This solution reduces the PE virtualization mechanism problem, which has been proven to be a critical design issue in bi-dimensional arrays [2].

Data are transferred into internal registers of each PE, processed and explicitly stored back into memory according to a RISC-like processing paradigm. The external memory can therefore be seen as a set of individually addressable Q bit wide lines. We will refer to the address of each line as an ***absolute line address.***

Each PE processes one pixel of each line and is composed of a Register File and a 1-bit Execution Unit. The instruction set core is based on morphological operators [1]. The result of an operation depends, for each processor, on the values of pixels in a given neighborhood (reduced 5 × 5, as shown in figure 1). Data from EAST and WEST directions may be obtained by direct connection with neighboring PEs while all other directions correspond to data of lines which have been previously processed (N, NE, NW) or which will be processed in the future (S, SE, SW). For this reason a number of processor registers have a structure which is more complex than that of a simple memory cell, and are actually composed of five 1-bit cells with a S→N shift register connection. When a load operation is performed, all data are shifted north by one position and the south-most position is taken by the new line from memory. In this way, data from a 5×5 neighborhood are available inside the array for each PE. This type of registers will be referred to in the following as *Morphological Registers (MOR)*. A second part of the instruction set comprises logical and algebraic operations which may be performed between the central pixels of MOR registers of the same PE, or between a number of single bit *Logical Registers (LOR)* which may be used for intermediate storage. The Execution Unit is responsible for applying morphological operators and for all the logical and algebraic operations. An Accumulator Register is used when a ternary algebraic operation, such as addition or subtraction, is executed to hold the value of the carry.

Two logical inter-processor communication mechanisms are available to exchange information among PEs which are not directly connected. The first one is the Status Evaluation Network (SEN) to which each processor sends the content of one of its registers under program control. The SEN provides two global flags, named SET and RESET, which are true when the specified register content is either all 1s or all 0s, and a status word which is set to the number of 1s in the specified register. This global information may be used to conditionally modify the program flow.

The second one is the Inter-processor Communication Network which allows global and multiple communication among components of different subsets of the PA (clusters of PEs). The topology of the communication network may be varied during program execution. In fact, each PE drives a switch that enables or disables the connection between it and the adjacent one. The PEs may thus be dynamically grouped into clusters, in which each PE can broadcast a register bit value to the whole cluster within a single instruction. This feature can be extremely useful in algorithms involving *seed-propagation* techniques, and in the emulation of pyramidal (hierarchical) processing.

An important characteristic of the system is the integration of a Serial-to-Parallel I/O Device, called *Video Interface* (VIF), which can be connected to a conventional imaging device (camera, linear CCD, etc). While a line is processed, the VIF automatically loads the following image line from the camera. At the end of this processing, the PA stores the results back into the VIF (on different bit-planes) and loads in parallel the following image line. During the data acquisition process, the VIF behaves like a shift-register, loading raw data serially from a camera, and outputting processed data serially to a display.

The system is designed to fit into a single integrated circuit, hence the control of all the processing elements is centralized at the chip level. Its design has been driven by two main considerations.

First many image analysis algorithms consist of sequences of low level steps, such as filters, convolutions, etc., to be performed over the whole image. Instruction fetching from an external memory is an important source of overhead. Hence we chose to pre-load each block of instructions (to be repeated for the whole image) into an internal memory, named Writable Control Store (WCS), and to fetch the instructions from there. In such a way it is possible to obtain the performance of a fast cache with a hit ratio close to 1, at a fraction of the cost and complexity.

The second consideration has been to avoid performance bottlenecks by matching the performance of the control section to that of the processing elements. To achieve this result, a sophisticated pipeline scheme has been developed which is expected to sustain an instruction execution flow of 100 Mops/s.

3 Instruction set

PAPRICA's instruction set has been designed to efficiently implement image processing algorithms. A dedicated hardware structure, named Image Descriptor, stores the base memory address of the image, the index of the line being processed at any time, and some other information regarding the way in which the image should be processed. This allows, for example, indirect memory access. To increase the flexibility of the processor, memory references may also be indexed using the contents of some special purpose counter registers, which are used as indices in loops. In addition to that, two instructions provide full synchronization and transfer capability to directly acquire data from a camera and send it after processing to a display. The instruction set fully supports the Mathematical Morphology approach to image processing, exploiting matching operations and logical operations. Bit-serial addition and subtraction with carry are also implemented. A branch instruction allows the programmer to take decisions on the flow of operations: considering the SIMD architecture, conditions must be set up on the base of global information, for which specific evaluation and accumulation instructions are provided.

PAPRICA's assembly language comprises the following classes of instructions:

- **General** and **Initialization** Instructions:

  NOP performs no operation.

  STOP performs soft and hard program stops.

  CONFIG configures neighborhood connection for sidemost processors.

  BASE, SKIP, RESET initialize image descriptor contents.

- **Memory Transfer** Instructions:

  LD loads data from memory into an internal register.

  ST stores data from an internal register to memory.

LDC loads data from the camera interface into internal registers.

STV stores data from internal registers to the camera interface.

ADD updates the memory address information of a specified image descriptor.

COMMOUT transfers internal register data among processor elements.

- **Morphological** Instructions:

LOP performs a logical or algebraic operation between two internal registers. Stores the result into an internal register.

MOP performs a match operation between an internal register (and its neighborhood) and the last set template. The result is then treated as the first operand of a logical or algebraic operation.

TEMPLATE sets the match template.

- **Status Evaluation** and **Flow Control** Instructions:

INIT initializes the status evaluation hardware.

EVAL performs status evaluation.

ACCUM accumulates status data on the dedicated status registers.

SET, INC initializes and increment counter register contents.

FOR repeats a block of instructions for a prespecified number of times. Enables indexed addressing.

ON sets up and evaluates a condition that influences the execution of the following instruction.

BRC unconditionally branches to a specified program memory address.

4 Pipeline description

In order to devise an efficient pipeline structure, instructions must be first divided into classes sharing the same kind of operations. For this purpose, only a coarse division of the instruction execution process is necessary. In our case the classical five phase pipeline [5] may be used:

IF Instruction fetch

ID Instruction decode

OR Operand read

EX Execution

OW Operand write back

Instructions	IF	ID	OR	EX	OW
NOP	x	x	-	x	-
STOP	x	x	-	x	-
CONFIG	x	x	-	x	-
BASE	x	x	-	x	-
SKIP	x	x	-	x	-
RESET	x	x	-	x	-
ADD	x	x	-	x	-
TEMPLATE	x	x	-	x	-
INIT	x	x	-	x	-
ACCUM	x	x	-	x	-
FOR	x	x	-	x	-
BRC	x	x	-	x	-
ST	x	x	x	x	mem
LD	x	x	mem	x	x
LDC	x	x	vif	x	x
STV	x	x	x	x	vif
COMMOUT	x	x	x	x	x
MOP	x	x	x	x	x
LOP	x	x	x	x	x
EVAL	x	x	x	x	x

Table 1: **Instruction phase requirements**

Only the IF, ID and EX phases must be present in all instructions. We can thus partition the instruction set on the basis of the required phases, as shown in table 4. The simplest pipeline structure would then comprise five stages corresponding to the five basic phases. This scheme, known in the literature as *monofunctional* ([5]), forces even unrelated instructions to wait if the preceding ones stall due to conflicts, thus degrading the overall performance. To avoid this effect, a multi-functional pipeline has been designed, which allows multiple instructions to be routed on different paths and execute simultaneously. The design objective was to reach peak performance for those patterns of instructions that most often are exploited in image processing: these comprise sequences of morphological instructions, loops, load-process-store, bit-serial computations. A particular attention has been dedicated to allowing potentially long instructions to run in parallel with shorter ones, by providing different paths for some of them. For example, flag evaluation, inter-processor communication, memory access and synchronization all share a wait mechanism that lets them "run in the background".

A comprehensive block-diagram of the proposed architecture is depicted in figure 2. A brief description of the functionality associated with each block follows:

IF Instruction Fetch. Updates the program counter content. Checks for active loops and handles their data management.

ID Instruction Decode. Decodes and routes instructions. Those which need operand read or write are routed to the conflict detection stage, after effective address calculation has been performed. The others are immediately executed. Branch instructions are executed at this stage, and require that IF be flushed to discard unwanted data. Flow control conflict detection and address evaluation are performed by this block. Due to the complexity associated to this stage, ID may be replicated many times in the implementation to meet the timing constraint.

CD Conflict Detection. Checks for data conflicts and handles the *booking table* for internal registers. Routes instructions to the proper block for execution.

OR Operand Read. Reads operands from internal registers. Checks for operand availability before execution. Routes instructions to the proper block for execution.

BP Bit Propagation. Delays MOP instruction execution for one cycle, to allow data propagation between neighboring processors.

EX Execution. Executes matching, logical and algebraic operators.

FE Flag Evaluation. Performs status evaluation and waits for completion. Stores the result in the status register.

CO Communication Out. Sends data to the communication network and waits for completion. Stores the result in the dedicated internal register.

MU Memory Unit. Manages all accesses to the image memory and waits for completion. Redirects instructions to the correct stage for operand write back (OW2 for logical registers, SM for morphological registers to perform the shift operation).

SY Synchronize. Waits for synchronization with the external camera and display.

OW1 Operand Write 1. Stores results back into internal registers. Checks for availability of the resource.

OW2 Operand Write 2. Stores results back into internal registers. Releases register occupancy flags and notifies the CD stage.

SM Shift MOR. Stores incoming data from image memory into internal morphological registers and shifts their content one line up. Releases the register's occupancy and notifies the CD stage.

SV Shift VIF. For LDC: load data from the camera interface into the internal morphological registers and shifts their content one line up. For STV: stores data from internal registers to the camera interface. Releases the register's occupancy and notifies the CD stage.

Instructions stall either when the desired operation cannot be executed because of resource contention, or when the block where the instruction must be issued

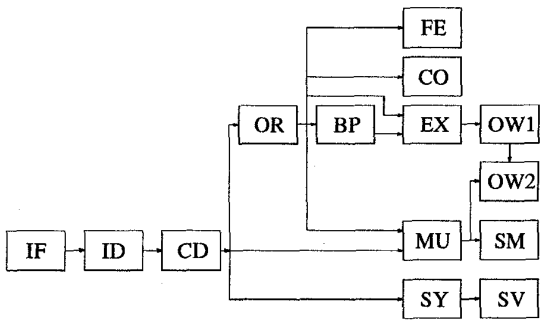

Figure 2: **Pipeline block-diagram**

is already occupied. The latter circumstance is minimized when different instructions are allowed to take different paths. This condition can be accomplished, as in our design, by multiple instances of the execution stage. On the other hand, this approach requires a more complex design of the conflict detection mechanism.

5 Conflict management

The concurrent execution of instructions in a pipeline structure causes contention conditions which must be detected and resolved in the most effective way. These conflicts between instructions may be classified as follows: ***structural conflicts*** which arise from the request of the same resource by two different instructions at the same time, ***data conflicts*** which arise because a pipeline structure allows the sequence of data accesses to be altered, and ***control conflicts*** which occur when the normal flow of operation is changed [5].

5.1 Structural conflicts

Given the structure of the pipeline, these conflicts are regarded as potential conflicts, since they occur only for particular sequences of instructions. Different structures are present in PAPRICA's hardware to perform different tasks:

- Register File read ports.
- Register File write port.
- Morphological Register shift.
- Video InterFace load/store.
- Execution unit (match, logical and algebraic).
- Flag Evaluation unit.
- Inter-processor Communication Network.
- Memory read/write access.

Each of these functional units corresponds to a different block in the pipeline block diagram, thus structural conflicts related to simultaneous requests of the same resource by two different blocks cannot occur. A conflict arises only when two instructions originating in the *execution* and *memory* units request for an operand write back. As seen in figure 2, block OW has been duplicated into two instances named OW1 and OW2. OW1 executes the write operation only when OW2 is empty, otherwise it behaves as a buffer booking OW2 for the following cycle and trasferring to it its content. In this way, while memory instructions may stall until a MOP/LOP sequence ends, only the latency of morphological instructions is increased without losing steady-state performance.

A conflict arises when a pipeline block may be accessed by different stages. This problem is caused by the MU and EX blocks:

- The MU stage may be accessed both by an LD in CD and by an ST in OR: in this case, the LD is stalled and the ST proceeds first, as in the natural order.

- The EX stage may be accessed by both a MOP in BP and a LOP in OR. Instead of stalling the LOP instruction in the operand read stage, the LOP is issued to the BP stage, which does nothing but delays it for one cycle. As for the OW stage, only latency is increased with no performance loss (in this case stalling is completely avoided).

5.2 Data conflicts

These conflicts occur when two instructions refer to the same data. The most common problem arises when the second instruction tries to read the data *before* the first one has changed it. To avoid this error, the second instruction must be stalled until the first has completed.

Different policies are implemented to avoid data conflicts for the different hardware structures.

- Image Memory data. This conflict never occurs because memory references are always performed by the same block (MU) *in order*.

- Status Registers data. This conflict occurs when an accumulation instruction is issued before the status evaluation from the previous one has completed. A dedicated flag in the pipeline informs the ID stage that an evaluation is in progress, stopping the accumulation until the evaluation completes.

- Internal Register data. This conflict originates from instructions which refer to the internal registers. They are: LD, ST, LDC, STV, MOP, LOP, COMMOUT, EVAL. Complexity arises from the different possible paths that these instructions may take, forcing the use of a sophisticated mechanism to detect contention. For each register in the Register File, a busy bit indicates whether a write operation for that register is in progress. The CD, OR and write back stages are in charge of handling such bits through the following procedure:

    1. For any instruction willing to write into an internal register, the register availability is checked in the CD stage against the state of the corresponding busy bit. If the busy bit is already set, the instruction waits in the CD stage until the register is released by one of the write back stages. Otherwise, if the busy bit is not set, the register is occupied by the instruction and control passes to one of the following stages. Only in the case of an STV, registers are booked as for a write operation, even though only a read operation will take place. This is necessary to avoid data being changed by other instructions during the VIF synchronization phase, which may take an unpredictable number of cycles.

    2. In the OR stage the source operands' busy bit is checked and the instruction is stalled if a write operation on such registers is in progress. Instructions referring to the same register as both the source and the destination operand do not execute the check phase (which had already been performed in the CD stage) to avoid deadlock.

    3. The write phases (OW1, OW2, SM, SV) inform the CD stage of the completion of their operation, and release the occupation of the internal register, making it available to the following instructions.

Correct operation is ensured by the CD stage, through which every instruction referring to internal registers must pass.

The CD stage handles such conflicts as *Write after Write* and *Write after Read*, while the *Read after Write* is managed by the OR stage. The *bypass* technique is often used in many pipeline structures to avoid stalling an instruction for a *Read after Write* conflict. This technique basically feeds the result of the execution stage back to the operand read stage, to make it immediately available. This technique is not applicable to our case, because:

- results are produced by both the EX and the MU stage, so a decision must be taken to get the data from the proper place.
- source operands come from different PEs, so a bit propagation delay must be taken into account before data are available to the OR stage.
- logical and morphological instructions may accumulate their result in AND or OR with the previous value stored in the destination register, so that the new value may not correspond to that computed by the execution stage.

However, this technique is applicable to the accumulator used for bit-serial computations, since its value is immediately available for the following instructions which need not be stalled.

Note that data stored in the image descriptors may be changed before an LD or ST completes: this does not matter because address calculation is performed immediately in the ID stage and the effective address in then delivered to the MU stage. On the other hand, the register which holds the template value must be preserved until every morphological instruction has terminated: for this reason, a TEMPLATE instruction waits in the ID stage if a MOP is still running in the pipeline.

5.3 Control conflicts

The program flow may be changed by the compound statements ON–BRC, which are executed in the ID stage. The first instruction is used to evaluate the condition, and the second to decide whether the jump must be taken or not. Since the ID stage is not released by the ON–BRC, only one other instruction is fetched before any decision is taken. Hence only the first stage of the pipeline need be flushed. Moreover, at least two more cycles are lost in the condition evaluation and branch execution. To avoid this loss of performance, a dedicated instruction has been introduced to handle fixed loops, which are a very common occurrence in the sort of application programs that we are targeting. The FOR instruction is executed in the ID stage, and simply sets some registers used by the IF stage to determine the value of the next program address. Since the number of instructions within the loop and the number of iterations are known *a-priori*, the IF stage always fetches the correct instruction and no cycles are lost for evaluating the end of the loop. Thanks to the indexed addressing mode, this mechanism highly increases performance and saves precious space in the program memory cache (WCS).

6 Performance evaluation

In this section, a few examples are examined based on the simulation of a behavioral model written in the Verilog Hardware Description Language [7]. A state diagram shows the instruction queue in the pipeline for a sequence of clock cycles.

The first example is shown in figure 3 and consists of a two-instruction loop (one morphological and one logical) *without* data dependency, and repeated 3 times. One instruction is issued for each cycle with no delay. Note how LOPs are routed to the BP stage to wait for a MOP bit propagation and leave space to the following instruction. Not considering the first (FOR) and the last (STOP) instruction, and assuming an infinite number of iterations, the performance gain over a non-pipelined solution is 5.5 (in real cases it is expected to be between 3.5 and 4.5).

The second example is shown in figure 4, and consists of a sequence of a MOP and a LOP instructions *with* data dependency. The second instruction waits in the OR stage for the MOP completion, and is then directly issued to the EX stage with no bit propagation. Note that the CD stage is free to manage other instructions to be delivered to different blocks. Peak speed up is 1.57 (in real cases it is expected to be between 1.3 and 1.45).

The last example is shown in figure 5 and consists of an LD and a MOP *without* data dependency. In this

FOR cycle : body of 2 instructions without data dependencies repeated 3 times

```
FOR 0 TO 2 DO 2
MOP
LOP
STOP
```

cycle no.	IF	ID	CD	OR	BP	EX	OW1
0	FOR						
1	MOP0	FOR					
2	LOP0	MOP0					
3	MOP1	LOP0	MOP0				
4	LOP1	MOP1	LOP0	MOP0			
5	MOP2	LOP1	MOP1	LOP0	MOP0		
6	LOP2	MOP2	LOP1	MOP1	LOP0	MOP0	
7	STOP	LOP2	MOP2	LOP1	MOP1	LOP0	MOP0
8		STOP (s)	LOP2	MOP2	LOP1	MOP1	LOP0
9		STOP (s)		LOP2	MOP2	LOP1	MOP1
10		STOP (s)			LOP2	MOP2	LOP1
11		STOP (s)				LOP2	MOP2
12		STOP (s)					LOP2
13		STOP					

Legend:
(s) - stall cycle

Figure 3: **For cycle: 2 instructions, 3 times, no data dependency**

LOP instruction waits for operand release by a previous MOP
Note that BP Stage free lets to bypass it by a LOP instruction

```
R4 = MOP R1 LOP R2;  // this MOP locks R4
R12 = R4 LOP R3;  // this LOP needs R4 as read operand
STOP
```

cycle no.	IF	ID	CD	OR	BP	EX	OW1
0	MOP						
1	LOP	MOP					
2	STOP	LOP	MOP				
3		STOP (s)	LOP	MOP			
4		STOP (s)		LOP (s)	MOP		
5		STOP (s)		LOP (s)		MOP	
6		STOP (s)		LOP (s)			MOP
7		STOP (s)		LOP			
8		STOP (s)				LOP	
9		STOP (s)					LOP
10		STOP					

Legend:
(s) - stall cycle

Figure 4: **MOP - LOP sequence with data dependency**

MOP instruction bypasses LD instruction waiting for memory read completion
No data dependencies
LD
MOP
STOP

cycle no.	IF	ID	CD	MU	SM	OR	BP	EX	OW1
0	LD								
1	MOP	LD							
2	STOP	MOP	LD						
3		STOP (s)	MOP	LD (w)					
4		STOP (s)		LD (w)		MOP			
5		STOP (s)		LD (w)			MOP		
6		STOP (s)		LD (w)				MOP	
7		STOP (s)		LD (w)					MOP
8		STOP (s)			LD				
9		STOP							

Legend:
(s) - stall cycle
(w) - wait cycle

Figure 5: **MOP overtakes a waiting LD**

case, a slow memory access time is assumed, so that the LD waits for 5 cycles in the MU stage. The MOP is free to overtake the waiting LD, and to complete its execution.

Obviously, it is not possible to evaluate exactly the speed up which may be obtained by a pipeline structure in the general case, because it essentially depends on the application program. Effective instruction scheduling by the programmer or by the compiler, aimed at avoiding data dependencies and at taking advantage of instruction concurrency may, in the best case, allow an overall speedup of 6 over the non-pipelined case. However, in the worst case only IF, ID and CD overlap giving a total speedup of 1.2. It has therefore been our goal to design a structure that would easily be exploited by image processing algorithms. An estimation of the number of 10ns cycles needed to filter one line of an 8-bit grey level image using a 100ns external memory and a 10ns internal memory is shown in the following table:

mem. cycle	no pipe	pipe	speed up
100ns mem.	554	241	2.3
10ns mem.	410	97	4.2

		imp. sch.	speed up
100ns mem.	554	131	4.2

The *no pipe* column refers to the number of cycles needed by a conventional sequential controller, while the *pipe* column refers to the pipelined case. The *speed up* colums reports the speed up achieved. The *imp. sch.* column refers to the case in which many cycles may be saved because part of the operation on the old line can be executed while loading the new data. Obviously, this possibility depends on the implementation of the algorithm.

7 Conclusions

A pipelined controller has been designed to execute instructions on a massively parallel processor. The main design objective was to boost performance for image analysis algorithms in order to be able to use the system in hard real time applications. Simulation shows that for a system comprising an internal fast image memory, the achieved speed up may reach a factor of 4-5 when little data dependency is involved. When using an external memory whose access time dominates the total processing time, intelligent instruction scheduling and concurrency exploitation are effective tools to gain further advantages. The design is currently being implemented using a mixed Macro module-based and Standard Cell-based methodology which relies on synthesis tools.

Acknowledgements

This work has been partially supported by Progetto Finalizzato Trasporti under contract number 94.01376.PF74, Progetto Speciale on VLSI Architecture for HDTV, and by MURST 40% on VLSI Architectures.

References

[1] J. Serra, "Image Analysis and Mathematical Morphology," *Academic Press*, London, 1982.

[2] F. Gregoretti, L. M. Reyneri, C. Sansoè, A. Broggi, and G. Conte, "The PAPRICA SIMD Array: Critical Reviews and Perspectives," *Proceedings ASAP'93 - IEEE Computer Society International Conference on Application Specific Array Processors*, Venezia, Italy, October 25 - 27 1993. IEEE Computer Society.

[3] A. Broggi, G. Conte, F. Gregoretti, L. Lavagno, L.M. Reyneri, C. Sansoè, G. Burzio, "PAPRICA-3 A Real Time Morphological Image Processor," *Proceedings of the 1st IEEE International Conference on Image Processing*, Austin, USA, November 1994.

[4] Yamashita et al, "A 3.84 GIPS Integrated Memory Array Processor with 64 Processing Elements and a 2-Mb SRAM," *IEEE Journal of Solid-State Circuits*, Vol. 29, No. 11, pp. 1336-1343, November 1994.

[5] D. A. Patterson, J. L. Hennessy, "Computer Architecture: a Quantitative Approach," *San Mateo: kaufmann*, 1990.

[6] P.G. Paulin, C. Liem, T.C. May, S. Sutarwala, "DSP Design Tool Requirements for Embedded Systems: a Telecommunications Industrial Perspective," *Journal of VLSI Signal Processing*, Vol. 9, No. 1-2, pp. 23-47, January 1995.

[7] D. E. Thomas, P. Moorby, "The Verilog Hardware Description Language," *Kluwer Academic Publishers*, 1991.