Dynamically Reconfigurable Architecture for Image Processor Applications

Alexandro M. S. Adário Eduardo L. Roehe Sergio Bampi

Institute for Informatics – Federal University at Porto Alegre
Av. Bento Gonçalves, 9500 – Porto Alegre, RS – Brazil
Cx. Postal 15064 – CEP: 91501-970

{adario, roehe, bampi}@inf.ufrgs.br

ABSTRACT

This work presents an overview of the principles that underlie the speed-up achievable by dynamic hardware reconfiguration, proposes a more precise taxonomy for the execution models for reconfigurable platforms, and demonstrates the advantage of dynamic reconfiguration in the new implementation of a neighborhood image processor, called DRIP. It achieves a real-time performance, which is 3 times faster than its pipelined non-reconfigurable version

Keywords

Reconfigurable architecture, image processing, FPGA

1. INTRODUCTION

Advanced RISC microprocessors can solve complex computing tasks through a programming paradigm, based on fixed hardware resources. For most computing tasks it is cheaper and faster to develop a program in general-purpose processors (GPPs) specifically to solve them. While GPPs are designed with this aim, focusing on performance and general functionality, total costs of designing and fabricating RISC GPPs are increasing fast. These costs involve three parts:

a) **Hardware costs**: GPPs are larger and more complex than necessary for the execution of a specific task. Developing application-specific processors for highly specialized algorithms is warranted only for large-volume applications that may require high power efficiency at expense of great hardware design cost;
b) **Design costs**: functional units that may be rarely used in a given application may be present in GPPs, and may consume substantial part of the design effort;
c) **Energy costs**: too much power is spent with functional units or blocks not used during a large fraction of the processing time.

For specific applications or demanding requirements in terms of power, speed or costs, one may rely on either dedicated processors or reused core processors, which may be well suited to the application or optimized for a given set of performance requirements. In the former case, only the necessary functional units highly optimized for a specific range of problems may be present, which will result in unsurpassed power and area efficiencies for the application-specific algorithm. Until recently, application-specific processors (ASPs) implemented in user-programmable devices were not feasible, but with increasing levels of FPGA integration (above 50K usable equivalent gates), as well as RISC cores and RAM merging into the reconfigurable arrays, the feasibility picture for user-configured ASPs has changed dramatically.

By tightly coupling a programmable device (e.g. FPGAs) to a GPP, we can exploit with higher efficiency the potential of the so called reconfigurable architecture. This structure can also aggregate some special on-chip macroblocks, like a shared memory. The dynamic reconfiguration of the hardware has become a competitive alternative in terms of performance against a GPP software implementation, and it offers significant time-to-market advantage over the conventional ASP approach.

Reconfigurable architectures allow the designer to create new functions and perform operations that would take too many cycles of the GPPs. With a reconfigurable architecture, a GPP does not need to include most of the complex functional units often designed-in, as well as considerable power and time can be saved.

These units may be implemented on the programmable device, when demanded by the application. Moreover, we can configure an application-specific subset of functional units from a larger set of units or a larger instruction set, and wire them up during execution time, increasing hardware density. Such density is the ratio between the sum of the resources used by all function units that can be mapped and the available resources on the programmable device.

This work is organized as follows. Section 2 presents a classification of reconfigurable architectures based on their execution models and reconfiguration scheme. Section 3 shows some examples of implemented architectures. The neighborhood processor is explained in section 4. DRIP Processor is introduced in section 5. Section 6 discusses the results of DRIP synthesis.

2. EXECUTION MODELS AND PROGRAMMABILITY

Page [10] reports five design strategies by which programs may be embedded in reconfigurable architectures: pure hardware, application-specific, sequential reuse, multiple simultaneous use, on-demand usage. Each model exploits a different part of the cost-performance spectrum of implementations and is well suited for a specific range of applications.

In a **pure hardware model**, a given algorithm is converted into a single hardware description, which is loaded into the FPGA. There is no relevant contribution of this model to reconfigurable architectures, since the configuration is fixed at design time. This model can be implemented using conventional HDLs and the currently available synthesis tools.

PRISM [3] is an example of **application-specific microprocessor (ASMP) model**. In this system, the algorithm is compiled into two parts (Figure 1.a): an abstract machine code and an abstract processor. In the next step, the two are optimized to produce a description of an ASP and the machine code level algorithm implementation.

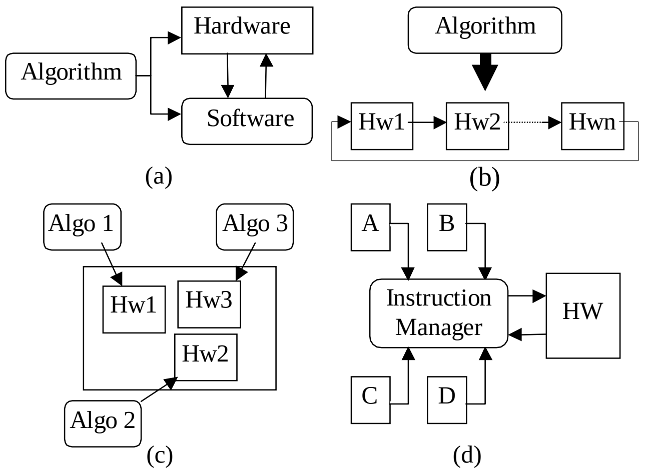

Figure 1. Example of execution models

Very often an algorithm is too large to be implemented on the available devices or the design is area constrained by engineering or economic reasons. To overcome this constraint, the design is splitted into several parts, which are moved in and out of the devices, increasing the hardware density and producing a set of reconfiguration steps (Figure 1.b). This model is called **sequential reuse**.

If there is a large availability of programmable devices, many algorithms can be resident and execute simultaneously, interacting with various degrees of coupling (tightened or loose) with the host processor. The **multiple simultaneous use model** (Figure 1.c) is less common, requires more area than the sequential reuse, but certainly is an interesting method to exploit reconfigurable computing.

The last model, **on demand usage** (Figure 1.d), is very interesting for reconfigurable computing, and can be applied to a wide range of applications. This model is suitable for real-time systems and systems that have a large number of functions or operations, which are not used concurrently, like the DISC (Dynamic Instruction Set Computer) implementation.

We generalize the execution models presented by Page, looking at reconfigurability from the point of view of the reconfigurable architecture design. This classification divides the design models in three programmability classes, considering the number of configurations and the time in which reconfiguration takes place:

a) **Static design (SD):** The circuit has a single configuration, which is never changed neither before nor after system reset. The programmable device is fully programmed to perform only one functionality that remains unchanged during system lifetime. This class does not exploit the reconfiguration flexibility, taking advantage only of the implementation/prototipation facilities.

b) **Statically reconfigurable design (SRD):** The circuit has several configurations (N) and the reconfiguration occurs only at the end of each processing task. This can be classified as run-time reconfiguration, depending on the granularity of the tasks performed between to successive reconfigurations. In this way, the programmable devices are better used and the circuit can be partitioned, aiming for resources reusability. This class of architecture is called SRA (statically reconfigurable architecture).

c) **Dynamically reconfigurable design (DRD):** The circuit also has N configurations, but the reconfiguration takes place at runtime (RTR, Run-Time Reconfiguration). This kind of design uses more efficiently the reconfigurable architectures. The timing overhead associated to this RTR procedure has to be well characterized within the domain of the possible set of run-time configurations. The overall performance will be determined by the overhead-to-computing ratio. The implementation may use partially programmable devices or a set of conventional programmable devices (when one process, the others are reconfigured). The resultant architecture is called DRA (dynamically reconfigurable architecture).

SRD and DRD run-time reconfiguration advantages depend largely on the specific algorithm and its partition in sizable grain tasks. The reconfiguration overhead is heavily dependent on FPGA microarchitecture, and it will be significantly decreased by FPGA + RISC core + SRAM integration within the same die, an area certainly open for recent innovations [7]. The SRD hardware will certainly show better performance when compared to GPP software implementation, given the large time overhead incurred for reconfiguration in current commercial FPGAs. The DRD hardware will benefit the most from innovations in the fast reconfiguration arena, while requiring significant more effort in developing compiler optimization. The main characteristics of all programmability classes are presented in Table 1.

Design	Number of Configurations	Reconfiguration at
SD	1	Design time
SRD	N	End of task
DRD	N	Execution checkpoint

Table 1. Summary of Programmability Classes

3. RECONFIGURABLE ARCHITECTURES IMPLEMENTATION

Several reconfigurable architectures were designed in the last decade, showing that this approach is feasible: DISC [13], PRISM [3], SPLASH [6], PAM [4] and Garp [7].

DISC is a processor that loads complex application-specific instructions as required by a program. It uses a National Semiconductor CLAy FPGA and is divided in two parts: a global controller and a custom-instruction space. Initially, a library of image processing instructions was created for DISC.

PRISM (Processor Reconfiguration through Instruction Set Metamorphosis) is a reconfigurable architecture for which specific tools have been developed such that, for each application, new processor instructions are synthesized. The tools for the PRISM environment use some concepts inherited from hardware/software codesign methods. Two prototypes, PRISM-I and PRISM-II, have been built using Xilinx XC3090 and XC4010 FPGAs, respectively.

SPLASH is a reconfigurable systolic array developed by Supercomputing Research Center in 1988. The basic computing engine of SPLASH is the Xilinx XC3090 FPGA. The second version of SPLASH, Splash 2, is a more general-purpose reconfigurable processor array based on XC4010 FPGA modules.

PAM (Programmable Active Memories) is a project developed by DEC PRL and consists of an array of Xilinx FPGAs. With dynamic reconfiguration, it has demonstrated the fastest implementation of RSA cryptography to that date [12].

Garp is a reconfigurable architecture that sets a trend to incorporate RISC cores with FPGA arrays. It incorporates a MIPS-II instruction-set compatible core with a reconfigurable array that may implement co-processor functions as a slave computational unit located on the same die of the processor. Garp simulation results have shown a 24X speed-up over a software implementation in a UltraSparc 1/170 of the DES encryption algorithm. In an image dithering algorithm for a 640x480 pixels frame the speed up obtained by Garp was 9.4 times.

4. NEIGHBORHOOD PROCESSOR 9

4.1 Array Processors

Array processors are a special class of parallel architecture, consisting of simpler processor called cells or processor elements (PEs). This class of processors has a large application on problems with spatially defined data structures such as Mathematical Modeling and Digital Image Processing. The PEs often have [5]:

a) a 2D matrix layout;
b) operation in a bit-serial mode;
c) access to a local memory;
d) connection to their nearest neighbors;
e) a synchronization scheme to execute the same instruction at any given cycle.

A neighborhood processor is a special device that simulates an array processor. It processes an input image, generating a new image, where each output pixel is an image function of its correspondent in the input image and the nearest neighbors. Using a standard neighborhood (e.g.: 3x3, 5x5 or 7x7 pixels), it scans the image line by line. NP9 [1] is based on the neighborhood processor architecture and is organized to process 3x3 (nine pixels) neighborhoods. Its architecture was first proposed by Leite [9].

4.2 The processor elements

The processor elements (PE) of NP9 are functionally simple and can execute just two basic operations: addition (ADD), representing the class of linear algorithms, and maximum (MAX), representing the class of non-linear algorithms. Each PE has two inputs (pixels X1 and X2), two weights (W1 and W2) associated to those inputs and one output S.

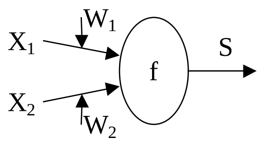

Figure 2. Model of NP9 Processor Element

4.3 Data Flow Graph

The PEs interconnection matrix of NP9 follows a data flow graph defined by a class of non-linear filters [11]. This class is widely used in digital image processing and the kernel of its data structure is represented by a sorting algorithm. The data flow graph (Figure 3) is based on the odd-even transposition sorting algorithm [8]. The hardware implementation of this algorithm is straightforward. The structure defined achieves a good trade-off between complexity, parallelism, area cost and execution time.

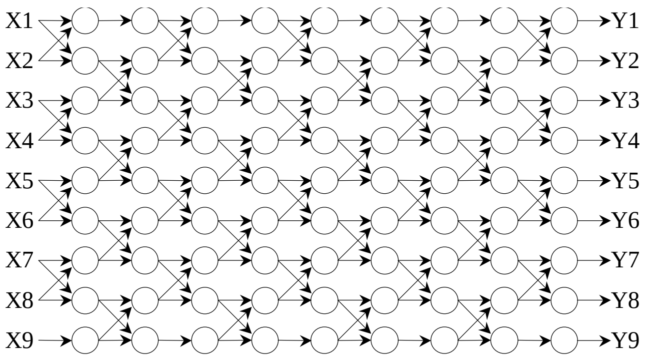

Figure 3. NP9 Data Flow Graph

4.4 General Structure of NP9

The general structure of NP9, at a high-level of abstraction shown in Figure 4, has three basic components: program register (PrgReg), execution pipeline and output mux. The execution pipeline corresponds to the data flow graph plus a stage register at each cell output. The external interface has nine input pixels (X1 to X9) and one serial output pixel (X_Out).

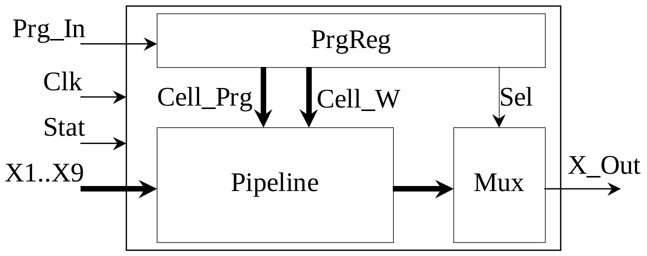

Figure 4. Structure of NP9

There are two possible operation states for NP9 indicated by the processor status signal (Stat): programming (PROG) or execution (EXEC). During the PROG stage, NP9 receives, through the programming channel, the data corresponding to the functions (Cell_Prg) to be executed by the cells, the input weights (Cell_W) and the output selector (Sel). The entire program is stored in a shift register (PrgReg). In the EXEC stage, the previously stored algorithm is executed, and the output (X_Out) is selected by a multiplexer.

4.5 Applications

Considering the primitive functions of NP9 and the programming flexibility associated to the NP9 data flow graph, one can configure a large number of low-level image processing algorithms onto its structure. Some algorithms that can be implemented on this processor are the following [9]: linear (convolution), non-linear and hybrid filters, binary and gray-level morphological operations (dilation, erosion, thinning and thickening algorithms, morphological edge detectors, gradient, "hit or miss" operator, "top hat" operators), binary and gray-level geodesic operations (geodesic dilation and erosion, image reconstruction), etc.

4.6 Implementation

NP9 was completely modeled in VHDL, compiled by QuickHDL under Mentor Graphics environment, and synthesized with AutoLogic II. After that, Max+Plus II processed AutoLogic netlists, generating NP9 architecture, implemented onto Flex10K FPGAs.

This final structure is a static design, and each algorithm generated for NP9 is implemented as a program. NP9 did not obtain the desired performance for real-time processing, considering 256 gray-level digital images of 1,024 x 1,024 pixels at a rate of 30 frames/s.

In addition, the resource utilization was also inefficient, and the processor used 6,526 logic elements in two FPGA devices (1 Flex10K70 and 1 Flex10K100). A dynamically reconfigurable design approach could reduce the resource utilization, allowing a cheaper final board, and a better performance was possible using reconfiguration. With this aim, DRIP was designed as a new reconfigurable architecture for digital image processing.

5. DRIP Architecture

DRIP (Dynamically Reconfigurable Image Processor) is a reconfigurable architecture based on NP9. DRIP design goal is to produce a digital image processing system using a dynamic reconfiguration (in a SRD scheme) approach. Based on previous NP9 design, we were expecting to obtain a minimum operation frequency of 32 MHz for real-time processing.

5.1 Customization of the Processor Elements

The first step in the definition of the DRIP architecture was the customization of its PEs. As mentioned above, each PE can implement two functions (ADD and MAX). The current model of the PE of NP9/DRIP operates with restricted weights, using only three values: -1, 0 and 1. Such parameters allow us to implement 8 distinct functions from a set of 18 possible configurations, not considering functions symmetrical on its inputs (e.g.: max(X1*0,X2*1) is equivalent to max(X1*1,X2*0) with inputs exchanged). These functions are summarized in Table 2.

With this information, we have designed optimized components, each one representing a distinct function. All functions together formed a basic function library, used for algorithm implementation during synthesis step (Figure 5). According to our design, the worst-case delay and resource usage in a function occurs for max(Xa, Xb) and defines DRIP maximum clock.

Original Operation	Function
add(0,0), max(0,0)	Zero
add(0,X2), add(X1,0)	X
add(0,-X2), add(-X1,0)	-X
add(X1,X2), add(-X1,-X2)	Xa + Xb
add(X1,-X2), add(-X1,X2)	Xa - Xb
max(0,X2), max(X1,0)	If positive(X) then X else 0
max(0,-X2), max(-X1,0)	If negative(X) then X else 0
other possibilities of max	Max(Xa, Xb)

Table 2. Customized functions of DRIP PE

5.2 Algorithm Mapping

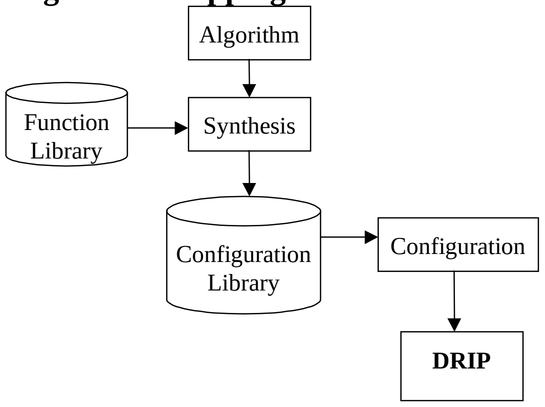

Figure 5. Design Flow of DRIP system

Typical design flow of the configuration of an algorithm onto DRIP is shown in Figure 5. First, an image processing algorithm is specified and simulated to verify its functionality. The specification can be done graphically, using an interface that represents the full DRIP/NP9 data flow graph. The algorithm can also be described using a high level language like C and translated to an intermediate representation which matches the DRIP architecture.

After specification, the algorithm is compiled/synthesized using the previously designed function library, fully optimized to achieve better performance. During algorithm synthesis, some optimizations are performed to reduce the complexity of the functions and to eliminate redundant or unused PEs. The configuration bitstream that customizes the FPGA for the optimized algorithm implementation is stored in a configuration library. The reuse of the modules of this library is essential for efficient implementation of several image processing functions.

Once the configuration bitstream data is stored, it can be used repeatedly and over several modules of the entire architecture. The synthesis and optimization steps for the configuration library elements may be slow, but this is counterbalanced by the reuse of the configuration library to implement more complex algorithms. Like in a software environment, the design and compilation times are sometimes large, but the massive use of the software can compensate its design costs and development time. This situation occurs in low-level image processing, where common algorithms are employed several times in distinct applications, and the size of the images and the number of frames require a great number of iterations.

5.3 Digital Image Processing System

In a complete digital image processing system (Figure 6), DRIP is connected to a visualization and acquisition system through a neighborhood generator. The generator receives the image pixels in serial form, line by line, and sends a complete neighborhood to be processed by DRIP. The configuration interface connects DRIP to the host computer and is responsible for the controlling of its configuration.

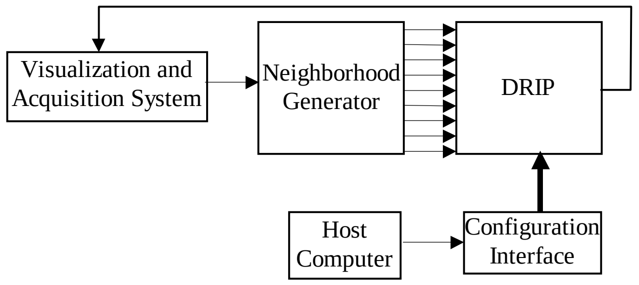

Figure 6. A Digital Image Processing System using DRIP

Currently, DRIP is being proposed as part of the entire system of Figure 6 in order to achieve a high performance image processor, relying on the dynamically reconfigurable features of DRIP. This goal is particularly challenging, since the neighborhood generator is a special memory with a very high bandwidth requirement. The best solution is to design a single chip containing DRIP and the generator, including substantial frame buffer memory, similar to the Garp implementation approach.

5.4 Potential Applications

A dynamic image processing system that relies on DRIP flexibility can take advantage of a web-based environment for remote processing. This possibility is not so far from the current state-of-the-art stage, given the increasing bandwidth available in the Internet. A Web-based framework can allow the distribution of the design/execution tasks: algorithm synthesis, program library storage, image acquisition, algorithm execution, image visualization. Each of these tasks can be performed on distinct sites throughout the web. The motivations for using the reconfigurable hardware herein proposed via Internet may be explained as an architecture on-demand.

A user may not have a machine with minimum requirements to execute compile/synthesis tasks or the software is not available for local use, only in a centralized algorithm server. A user system may have insufficient speed to perform more complex image transformations exclusively in software. A system including DRIP processor can be a server for digital image processing, and remote users can submit images and receive the visualization in their client browsers. The feasibility of such implementation depends on issues that are not addressed in this paper.

6. RESULTS

The customized functions of DRIP were compiled for Altera Flex10K FPGAs. We individually analyzed the performance of each function. After that, we choose the worst-case function in resource utilization and performance to implement a full DRIP data flow graph. The preliminary estimated performance (51.28 MHz) is 60% greater than the design target performance (32 MHz) and almost 200% faster than the fixed hardware (non-reconfigurable) NP9 implementation (17.3 MHz). The comparison between these performances is presented in Figure 7.

On the resource usage, DRIP also achieved much better results. It used only 1,113 logic elements of a Flex10k30, 83% less area than the NP9 pipelined implementation. The reconfiguration time was also reduced due to the FPGA device used, as the SRAM-based FPGA technology Flex10K used provides for much faster reconfiguration. Unfortunately, this family does not support partial reconfiguration yet, as does the Xilinx XC6200 family.

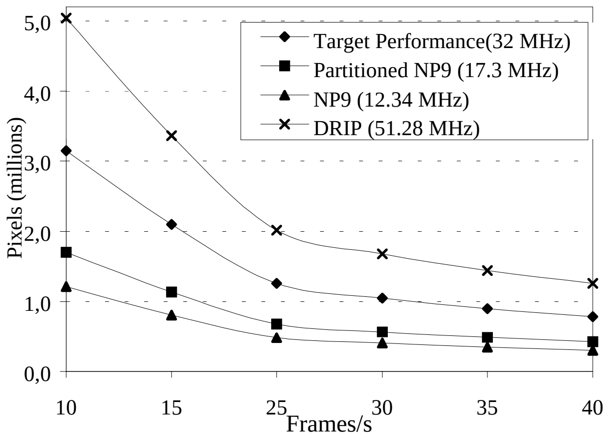

Figure 7. Graph of Processed Pixels x Frames/s

7. Conclusions and future work

Dynamic reconfiguration can obtain a considerable gain in area, performance, and cost for an application specific system. We demonstrate such advantages with the implementation of a reconfigurable dynamic image processor. Comparing with the equivalent statically configured (SD) design of NP9, an 80% area reduction and 3 times faster clock rate was achieved. The DRIP processor is then suitable for a high performance real-time image processing.

The design flow emphasizes a fundamental requirement in reconfigurable design: performance gain and large application life cycle must overcome the design costs and development time. A relevant trend in FPGAs and reconfigurable architecture is also bound to change considerably the picture in favor of DRD as the integration of ASP, GPP, and memory in one single chip becomes economically viable.

The future directions of the DRIP project will require an algorithm/program specification framework to allow a user-friendly interaction between high-level abstraction and processor architecture. The development of a system board with full support for dynamic reconfiguration of the processor and high-bandwidth memory availability is planned.

We are considering the implementation of a Web-based framework for remote image processing. This framework will allow to define a methodology for dynamic reconfiguration on a widely distributed environment. That could become a relevant market for reconfigurable computing, for efficient supply of an architecture on-demand. Hardware upgrading, maintenance, and adaptation could be performed from a remote host.

8. REFERENCES

[1] Adário, A. M. S.; Côrtes, M. L.; Leite, N. J. "A FPGA Implementation of a Neighborhood Processor for Digital Image Applications" In: 10 Brazilian Symposium on Integrated Circuit Design, Ago 1997. Proceedings..., 1997 p. 125-134.

[2] ALTERA. Data Book. Altera Corporation, San Jose, California, 1996.

[3] Athanas, P.; Silverman, H. F. "Processor Reconfiguration Through Instruction Set Metamorphosis". IEEE Computer, Mar 1993. p 11-18.

[4] Bertin, P. et al, Introduction to Programmable Active Memories. Paris: Digital Equipment Corp., Paris Research Lab, June 1989. (PRL Report 3).

[5] Fountain, T. J. Processor Arrays: Architecture and Applications. Academic Press, London, 1987.

[6] Gokhale, M. et al. "Building and Using a Highly Parallel Programmable Logic Array." Computer, vol. 24, no. 1, Jan 1991. p 81-89.

[7] Hauser, J. R.; Wawrzyneck J. "Garp: A MIPS Processor with a Reconfigurable Coprocessor". In: IEEE Symposium on FPGAs for Custom Computing Machines, 1997. Proceedings... p 24-33.

[8] Knuth, D. E. The Art of Computer Programming, Reading, Massachusetts: Addison-Wesley, 1973.

[9] Leite, N. J.; Barros, M. A.; "A Highly Reconfigurable Neighborhood Image Processor Based on Functional Programming". In: IEEE International Conference on Image Processing, Nov 1994. Proceedings... p 659-663.

[10] Page I. Reconfigurable Processor Architectures. Microprocessors and Microsystems, May 1996. (Special Issue on Codesign).

[11] Pitas, I.; Venetsanopulos, A. N. " A New Filter Structure for Implementation of Certain of Image Processing Operations". IEEE Trans. on Circuits and Systems, vol. 35, n. 6, June 1988. P 636-647.

[12] Shand, M.; Vuillemin, J. "Fast Implementations of RSA Cryptography". In: 11 Symposium on Computer Arithmetic, 1993, Los Alamitos, California. Proceedings... p 252-259.

[13] Wirthlin, M. J.; Hutchings, B. L. "A Dynamic Instruction Set Computer". In: IEEE Symposium on FPGAs for Custom Computing Machines, Apr. 1995. ***Proceedings...*** p 92-103