On Current Strategies for Hardware Acceleration of Digital Image Restoration Filters

ERIC GRANGER
Laboratoire d'imagerie, de vision et
d'intelligence artificielle
Dépt. de génie de la production automatisée
École de Technologie Supérieure
1100, rue Notre-Dame Ouest
Montreal, QC, H3C 1K3, CANADA

SERGE CATUDAL, ROBERT GROU,
MAME MARIA MBAYE and YVON SAVARIA
Groupe de recherche en microélectronique
Dépt. de génie électrique
École Polytechnique de Montréal
PO Box 6079, Station centre-ville
Montreal, QC, H3C 3A7, CANADA

Abstract: - Two advanced design methodologies for hardware acceleration of a standard digital image restoration algorithm are explored and compared. The first one is the custom-designed hardware approach, leading to an application-specific integrated circuit (ASIC) implementation. The second one consists of the configurable processor approach, yielding a mixed hardware/software implementation running on a Tensilica Xtensa V (T1050) microprocessor. Both implementations may be embedded as cores in System-on-Chip (SoC) designs. The two methodologies are compared from several standpoints, including implementation size, data throughput, customization, non-recurring engineering and production costs, and flexibility.

Key-Words: - Image restoration, adaptive Wiener filter, hardware acceleration, SoC design, ASIC, configurable processor, Tensilica Xtensa.

1 Introduction

The range of technologies currently available for hardware acceleration of digital filters that are used in time-critical image processing applications is broader than ever. The choice of one over the others is not always straightforward, especially when considering the fast pace at which technologies, and the associated design tools are changing. For instance, mixed hardware/software and reconfigurable platforms are some of the recent alternatives to conventional custom-designed hardware implementations, *i.e.*, application-specific integrated circuits (ASIC) and field-programmable gate array (FPGA) circuits, and to conventional software implementations running on, *i.e.*, standard microprocessors and digital signal processors (DSP). Selecting a technology that can meet the throughput, latency, ease of change, reliability, supportability and size specifications within the development and unit cost budgets can be challenging.

As a research experiment, an adaptive Wiener filter [1] [3] [6] has recently been implemented according to different strategies for hardware acceleration. Wiener filters may be considered as representative of digital filters used in image restoration and noise reduction applications. In fact, one of the first methods developed for restoring images degraded by additive random noise is based on Wiener filtering, and it has since influenced the development of several other image restoration systems [3]. Pixel-by-pixel variants of adaptive Wiener filters are computationally intensive, and require hardware acceleration for time-critical applications, since processing is adapted at each pixel of an image.

Two different design strategies for hardware acceleration of these filters are presented and compared in this paper. The first one is a custom-designed hardware strategy, whereas the second one is a configurable processor strategy. Both approaches result in an implementation of a dedicated image restoration core that may be embedded into System-on-Chip designs. While a design according to each strategy would be valuable in itself, the comparison of the results obtained and the lessons learned may also provide useful insight. The two methodologies are compared to each other in terms of processing rate, implementation size, and cost.

In the next section, adaptive Wiener filtering is briefly reviewed. Then, implementations of the

Wiener filter according to two design strategies are described in Section 3. They are compared and discussed in Section 4.

2 Adaptive Wiener filtering

The Wiener filter is a linear estimator minimizing the mean-squared error (MSE) between the estimated and the original signal, knowing some statistic about the original one. The essential idea behind this filter is to exploit information contained in the image at hand, as well as the imaging system being used. It has been used extensively as a solution to image restoration[1] problems, to reduce or eliminate additive random noise degradation [4]. In this context, the Wiener filter is a method of image recovery that is designed to minimize the MSE criterion.

Some adaptive versions of the Wiener filter perform pixel-by-pixel processing, where filtering is based on the changing local characteristics of the image, degradation, and any other relevant information in the neighborhood centered around the pixel. The mean and power spectrum of the signal and noise are estimated locally instead of being fixed. Although adapting processing at each pixel is computationally intensive, these filters have the advantage of reducing additive random noise without significantly blurring the image.

Lee's Minimum Mean Squared-Error (MMSE) algorithm [3] is a pixel-by-pixel variant of the adaptive Wiener filter that is very popular in image processing. Additive noise is assumed to be zero mean and white Gaussian with a variance of σ_n^2. The algorithm progresses through each pixel (x, y) of a noisy digital image, I, and produces a filtered digital image, I^*, according to the following equation [6]:

$$I^*(x,y) = \mu(x,y) + \frac{\sigma^2(x,y) - \sigma_n^2}{\sigma^2(x,y)}[I(x,y) - \mu(x,y)] \tag{1}$$

where σ_n^2 is the variance of additive Gaussian white noise, and $\mu(x,y)$ and $\sigma^2(x,y)$ are estimates of the local mean and variance, respectively, associated with the pixel of coordinate (x,y). The variance of additive white noise σ_n^2 is assumed to be known *a priori*, or is estimated from the local variances $\sigma^2(x,y)$ of current and/or previous images in a sequence. The local mean and variance are calculated based on η, which represents an N-by-M window of pixels centered over the local neighborhood of pixel (x,y). They may be defined by:

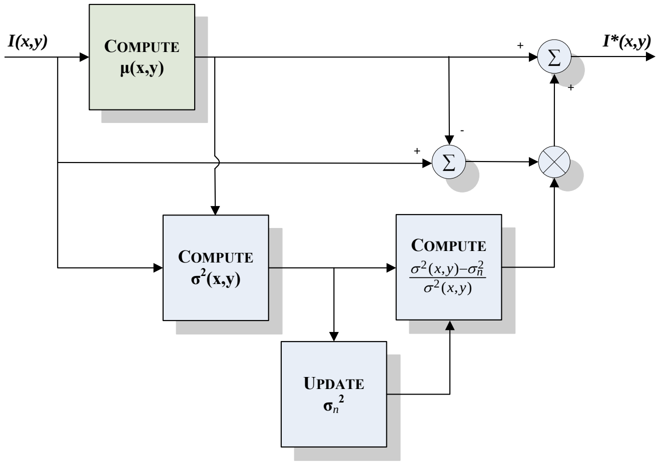

Fig. 1. Block diagram of the MMSE algorithm.

$$\mu(x,y) = \frac{1}{NM}\sum_{x,y\in\eta} I(x,y) \tag{2}$$

$$\sigma^2(x,y) = \frac{1}{NM}\sum_{x,y\in\eta}[I(x,y)^2 - \mu^2(x,y)] \tag{3}$$

A window size is defined by odd numbers, typically $N = M = 3$ or 5. Contour pixels are usually handled by using a mirror of defined pixels in the corresponding window.

Figure 1 shows the bloc diagram of the MMSE algorithm applied to each pixel of an image, for a given M and N value. It is assumed that σ_n^2 is obtained by progressively updating an average $\sigma^2(x,y)$ value from the current and previous images in a sequence. Several computational aspects of this algorithm are demanding, and may require hardware acceleration for time-critical application. They include the multiplications and accumululations to compute the local mean and variances in Eqs. (2) and (3), and the division by the local variance, and, potentially, the estimation of σ_n^2 needed to perform filtering in Eq. (1).

3 Methodologies for hardware acceleration

Figure 2 presents the architecture of a dedicated image restoration system to implement the adaptive Wiener filter described in Section 2. It consists of a pixel receiver and transmitter, input/output devices, a direct memory access (DMA) controller, an internal line buffer, a generic high speed bus, and a main controller.

[1] Image restoration refers to the process of recovering an image that has been contaminated by noise, and blurred by the image system involved [4].

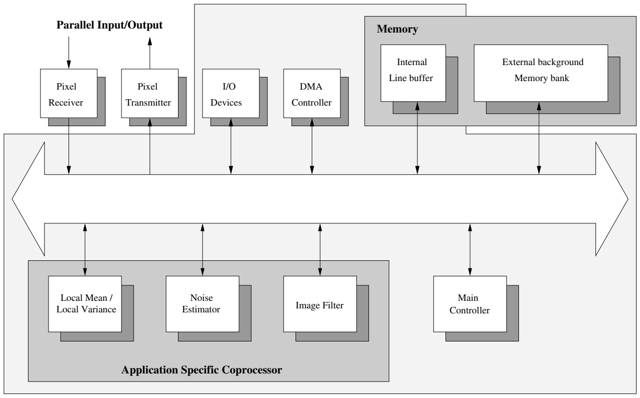

Fig. 2. Architecture of a system to reduce additive white Gaussian noise degradation based on the MMSE algorithm.

In order to accelerate filtering for time-critical applications, the architecture also contains an application-specific coprocessor that computes the local mean and variance, the variance of noise, and image filtering in the place of software running on the main controller. The architecture also interfaces with an external memory bank via an external memory controller to expand storage capabilities.

To filter a noisy image, successive pixels are received by the pixel receiver, and transferred to external memory. The DMA controller ensures that relevant lines of pixels are progressively transferred between the external memory and the line buffer without intervention by the main controller. The line buffer stores M lines of pixels that are as long as the image being processed, and transfers successive $M \times N$ pixel windows to the application-specific coprocessor. Pixels inside the line buffer are therefore used by the coprocessor to either (1) estimate local mean and variance values, (2) estimate the variance of additive noise, or (3) filter the noisy pixels. As pixels are processed, results are stored in external memory, where they may be transferred to the pixel transmitter.

From this point on, this paper focuses on currently-available strategies to implement the application-specific coprocessor for filtering a set of pixels stored in an internal line buffer. Although the division in Eq. (1) is an issue for any implementation, more emphasis is placed on accelerating implementations of other computationally demanding parts of the MMSE algorithm, as similar computations are more commonly found in digital image processing filters. In order to eliminate another potential bottleneck, the estimate of σ_n^2 is obtained progressively, by updating an average $\sigma^2(x, y)$ value from the stream of images. (Otherwise, a frame buffer and significant processing would be required for each new image.) It is assumed that the system's controller is a simple finite state machine (FSM) dedicated to orchestrating this task. It is also assumed that the window size is of $M \times N = 3 \times 3 = 9$ pixels, and that each pixels is stored with 8 bits.

The options available to implement the application-specific coprocessor for time-critical image processing applications range from software running on a standard microprocessor to custom-designed hardware. For instance, given C or C++ software corresponding to the application-specific processing, the code can be quickly compiled for, and then run on, a DSP, or alternately, it can be mapped to high-performance FPGA or ASIC designs. Between these two extremes, several mixed hardware/software and reconfigurable platforms offer interesting performance-cost trade-offs. In either case, these implementations can be viewed as cores to be embedded into SoC designs.

The rest of this section presents the main steps required to design the application-specific coprocessor, using currently-available design tools, according to two advanced design methodologies.

3.1 Custom-designed hardware approach:

Designing custom hardware represents a conventional approach to accelerating functions needed in embedded systems. This approach is traditionally employed to achieve a very high level of performance for large production. It offers low-level control over the circuitry that is generated, and therefore allows optimize the circuit size, clock frequency, and power consumption to suit the application needs. Although this paper focuses on ASIC implementation, some or all of the general approach described in this section may also lead to FPGA and SoC designs.

With this approach, a designer would usually begin his design with a specification, and a software model in, *e.g.*, C or C++. The following is a standard ASIC design methodology for custom hardware.

Architecture modeling. At first, the software model is translated to a dedicated hardware architecture. This architecture is analyzed to ensure that estimates of performance meet design constraints. SystemC and related design tools such as CoCentic Studio may be used to develop an executable model or virtual prototype of the architecture, and thus perform architectural

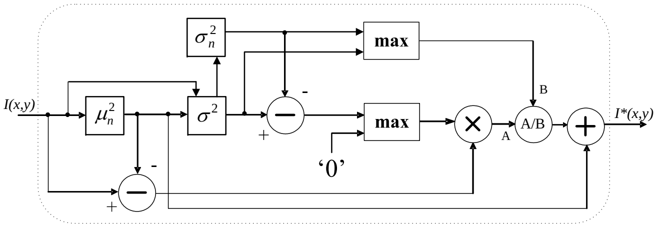

Fig. 3. Data path for the ASIC implementation of the application-specific coprocessor.

exploration, asses performance, etc. [11]

Design entry and analysis. Functional blocks of the architecture are coded using a hardware description language (HDL), such as Verilog or VHDL, and then interconnected to form the the coprocessor’s data path. For more standard functional blocks, design time may be reduced by purchasing IP cores, and embedding it into the design. Internal operation of all the functional blocks is controller by a FSM. This design step also involves behavioral simulation using a pre-defined testbench, and tools such as Synopsys VCS or Cadence NC Verilog, and yields an RTL-level description of the application-specific coprocessor.

The block diagram of Figure 3 shows the data path described in RTL for the application-specific coprocessor. As shown in the figure, successive pixels are pipelined through a chain of functional blocks. A pipelined fixed-point divider (a non-restoring division array [2]) was implemented with 17 pipeline to increase throughput. In addition, previously-calculated values were stored in delay buffers and reused to streamline performance. After an initial latency of 41 clock cycles to fill the pipeline, the data path filters pixels of an image at a rate of 1 pixel per clock cycle. Since this design is cascadable, the application-specific coprocessor is implemented with 4 such data paths, to filter 4 pixels in parallel. In this case, however, the line buffer must store 6 lines of pixels.

Technology optimization. This step in the design flow involves logic and physical synthesis of the RTL-level description using tools such as Synopsys Design Compiler and Physical Compiler, and Cadence RTL compiler. Synthesis calls on standard-cell hardware technology libraries, wire load models, etc., to map the RTL-level description to the gate-level or netlist description of the application-specific coprocessor. Physical synthesis approaches may be employed to address the shortcomings of traditional flows by concurrently optimizing the logical and physical design, rather than relying on statistically-based wire-length models.

The RTL-level description of the data path presented in Figure 3 was synthesized using Synopsys Design Compiler, and the TSMC 0.18μm worst-case technology. The delay between divider pipeline registers constitute the design’s critical path, and sets the maximum clock frequency at 120 MHz. In this case, circuit size of the data path is about 18.5k gates.

Design verification. The netlist description is subjected to static timing analysis (STA), gate-level simulation, formal verification, power estimation, and pre-layout technology checking (*e.g.*, timing convergence) using well-known design tools. Critical paths are estimated from statistical models and timing violations are fixed by re-synthesizing with new timing constraints or by restructuring the logic.

Layout. Layout of the verified netlist involves floor planning to arrange cores from a hard macro library and I/Os, placement of synthesized gates, clock tree synthesis, post-layout technology checks, and automatic test pattern generation. From the layout, critical paths from the placed design are extracted, and back-annotated to the STA tool. However, when actual wirelengths do not match predicted pre-placement statistical based wire lengths, this can cause a timing problem and can lead to costly design iterations.

Finally, once this ASIC design flow is completed, the resulting ASIC core may be embedded into a SoC design. For addition information on this approach to hardware acceleration, the reader is referred to [9].

3.2 Configurable processor approach:

Configurable processors represent newer alternatives to embedded systems design, where current technologies allow to generate application specific instruction-set processors (ASIP). This approach allows to connect a specialized hardware coprocessor to the processor core[2], with the advantage of reducing the communication costs (typically associated with multiprocessor approaches to hardware acceleration), and the design effort [8] to mostly crafting specialized instructions (SIs).

Although Altera offers similar technology with the NIOS processor [7], which is targeted for FPGA designs, this papers focuses on Tensilica’s Xtensa V (T1050) processor technology [10], which is targeted

[2] The processing core implements base instructions, and optional modules, such as a floating point coprocessor.

for ASIC designs. This technology effectively yields a mixed hardware/software implementation running on a Tensilica Xtensa V (T1050) microprocessor.

A designer would usually begin his design of a configurable Xtensa processor with a specification, and the initial executable software code in, *e.g.*, C or C++. The following is the basic design methodology for the Xtensa configurable processor.

Initial code profiling. For comparison purposes, it is important that the design cycle begin by measuring the performance of the initial code before optimizing the processor. Code profiling allows to isolate performance bottlenecks or areas of the code that may be accelerated.

Code cleaning. For code that is destined for embedded systems, it is important to verify that relevant programming rules [5] are respected.

Specialized instruction design. At this point, the designer begins iteratively optimizing, where each iteration consists in designing a SI, and profiling the resulting code until a timing performance target has been reached. Tensilica's Instruction Set Simulator (ISS) allows to estimate the number of cycles associated with a SI.

Several SIs were implemented optimize the speed of the application-specific coprocessor. The first SI allows to compute the local mean and variance for one pixel, whereas the second one allows to compute the same values for 4 pixels at a time. These computations constitute a performance bottleneck for the MMSE algorithm. Finally, a third SI allows to apply adaptive filtering to 4 pixels simultaneously.

For the first instruction, internal registers that store the local mean and variance are created. These registers are initialized to zero prior to pixel processing, and assigned intermediate results as processing progresses through a pixel window. The final register values are read by the code. An acceleration factor of almost 30 was achieved by implementing the local mean and variance with a SI, instead of in software.

For the second instruction, 3 words, each consisting of 4 pixels, are stored in 32 bit internal registers. These words represent a total of 12 pixels, which provide 3 processing windows. The local mean and variance may be computed directly with these 3 words. Since some pixels are missing from its processing window, the computation of the 4th pixel cannot be computed directly. To compute this last pixel, 3 registers were defined inside the processor core, which implement a stack, and which store the last pixels of a window. The local mean and variance of the last pixel are thereby computed while processing the next set of 4 pixels. Overall, this SI processes 4 pixels at a time, with the exception of the 1st pixel of a line, where only 3 pixel can be processed.

Finally, one last SI was implemented to perform filtering on 4 pixels simultaneously, based on the local mean and variance of 4 pixels, as well as the variance of the additive white noise. Using SIs to process 4 pixels in parallel only yields a speedup factor of about 2. Indeed, our parallel processing approach requires loading data into the internal registers of the processor core, which tends to decrease performance gains. The line buffer must also store 6 lines of pixels.

Specialized instruction synthesis. Once the timing performance targets have been reached through the inclusion of specialized instructions, the Xtensa Processor Generator allows to generate a RTL-level description for a specialized coprocessor that implements the set of SIs. Synthesis of this description for a specific hardware technology yields a specialized coprocessor containing the circuitry required to implement the SIs. This coprocessor also contains additional circuitry, such as a decoder, for integration into the Xtensa processor.

Based on synthesis reports for the resulting coprocessor, and the design constraints, a SI may or may not be acceptable. Other instructions may also be designed to further optimize performance. For instance, the parallel computation of the local mean and variance for 4 pixels creates a long data path with two stages of multipliers and adders. Pipelining the critical path can be achieved by splitting the second SI into two separate SIs, each one associated with a with a single stage of operations. As a result, the maximum clock frequency almost doubles, although the specialized coprocessor increases in size by about 10% (since additional internal registers are needed to store intermediate results between the two stages).

Finally, once the performance targets have been reached in terms of both speed and area, the resulting processor – core processor connected to the specialized coprocessor – may be generated, and then embedded into a SoC design. For addition information on this approach to hardware acceleration, the reader is referred to [10].

4 Comparison of design strategies

The purpose of this section is to compare the two methodologies and corresponding implementations to each other, for the same hardware technology and application, from the standpoint of their processing rate, implementation size, and cost.

The custom-designed RTL-level description of the application-specific coprocessor was synthesized using Synopsys Design Compiler, and the TSMC 0.18μm worst-case technology. The maximum clock frequency of this ASIC design is 120 MHz, and the circuit size is about 74.0k gates.

An Xtensa V (T1050) processor has been generated to implement the application-specific coprocessor. From the performance estimate provided by Tensilica's tool suite, and with TSMC 0.18μm worst-case technology, the processing core has a maximum clock frequency that ranges from 100MHz to 215MHz, and a circuit size that rages from about 66.9k to 80.6k gates. Finally, the RTL-level description corresponding to the specialized coprocessor was synthesized using Synopsys Design Compiler, and the same hardware technology as the processor core. The coprocessor has a maximum clock frequency of 211MHz and a circuit size of about 48.5k gates.

Table 1 presents a summary of performance estimates obtained by using the custom-designed hardware and the configurable processor strategies to implement the application-specific coprocessor. The processing rate estimates are given for images of size 256×256 pixels, and 352×288 pixels (used for Motion JPEG). The processing time is defined by the clock frequency, and the number of clock cycles needed by the coprocessor to filter all pixels of an image. This time includes the number of cycles to compute $\mu(x, y)$ and $\sigma^2(x, y)$, to progressively update σ_n^2 and to perform filtering for each pixel of the images. The time required to move successive pixels to and from the line buffer is excluded, since it is identical for both implementations. The gate count consists of the sum of NAND gates required to implement the application-specific coprocessor.

As shown on the table, the ASIC implementation achieves a processing rate that is two orders of magnitude faster that that of the Xtensa processor, yet requires a much smaller number of gates. In fact, the specialized coprocessor generated by the Xtensa processor incurs significant overhead (decoder, MUXs, etc.) to allow for operation with the core processor. The custom-designed hardware strategy, on the other hand, offers greater control over the circuit that is generated from its RTL-level description.

The non-recurrent engineering (NRE) cost of an implementation is a function of the length of the design flow, the required software tools, and the experience of the designers [9].

Table 1. Estimated performance resulting from the two different hardware acceleration strategies used to implement the application-specific coprocessor with image sizes of 256×256, and 352×288 pixels.

Performance measures	**ASIC design (@120 MHz)**	**Tensilica Xtensa T1050 (@211 MHz)**
Processing rate:		
- 256×256 **pixels:**		
# clock cycles	16,578	2,982,853
processing time	0.138 msec	14.14 msec
- 352×288 **pixels:**		
# clock cycles	22,578	4,299,954
processing time	0.213 msec	20.38 msec
Total gate count:	74.0k	129.1k

The custom-designed hardware approach involves the highest NRE cost. It involves expensive software tools, by companies such as Synopsys, Cadence and Mentor Graphics, and a relatively complex design flow, entailing a long learning curve and highly specialized designers. Moreover, this implementation is the most difficult and expensive to modify once the devices have been produced. On the plus side, for a large production, the ASIC implementation offers an economical unit cost.

Even though the design flow is less complex, the unit cost associated with the configurable processor approach is moderate, and it requires a specialized designer, and complex software tools. Significant knowledge of the software and architecture of embedded processors, and of digital VLSI circuit design is required to generate an optimized Xtensa processor. For instance, reordering the operation sequence of an SI requires considerable expertise in order to assess its impact on the area and clock frequency of the resulting coprocessor. Otherwise, the circuitry created by the SI quickly becomes the performance bottleneck of the processor. The designer must also keep track of data dependencies between SIs, and of the processor's pipeline. Finally, a significant amount of design effort must be invested in verifying that an SI functions in the same way as the initial code sequence. The design effort may be alleviated with Tensilica's trend towards the automated proposal of SIs and corresponding speedup factors.

On the plus side, the software component of the processor is easy and economical to modify. Another advantage of the configurable processor approach is that hardware/software codesign analysis is performed directly in the development environment. Hardware components of the application-specific coprocessor are generated automatically from performance bottlenecks found during code profiling.

Although outside the scope of this paper, the division in Eq. (1) represents a potential bottleneck to hardware acceleration. A basic pipelined fixed-point divider was implemented for the ASIC design. There are several alternative designs to accelerate the division, each one having a different impact on the circuit size. For instance, one could alternate use of 2 basic dividers operating in parallel, such that the clock frequency is effectively doubled, or one could use a Taylor series expansion to estimate the division, and a look-up table to store common factors. A software division was performed with the Xtensa processor. Since this operation requires about 100 clock cycles, crafting SIs to accelerate this operation would have a significant impact on overall performance.

5 Conclusions

In this paper, two advanced design strategies for hardware acceleration of an adaptive Wiener filter are explored and compared. The first one is a custom-designed hardware strategy, leading to an ASIC implementation, an the second one is a configurable processor strategy, yielding a mixed hardware/software implementation running on a Tensilica Xtensa V (T1050) microprocessor. Both approaches result in an implementation of an application-specific coprocessor that may be embedded into SoC designs.

Performance estimates indicate that the ASIC implementation can process images at a rate that is two orders of magnitude greater that that of the Xtensa processor. This level of performance is attained with a much smaller gate count. The ASIC implementation, however, has higher NRE costs, and is difficult to modify once it has been fabricated. Nonetheless, it offers an economical solution for high volume production.

The Xtensa processor implementation presents an interesting alternative in that the design flow is less complex, and software components of the implementation can easily be modified, and hardware/software codesign analysis is performed on the fly. However, this approach still required a very specialized designer, and a potentially long design cycle to produce an optimized Xtensa processor, especially for larger, complex application code. Even then, specialized coprocessors generated for the Xtensa processor incur a significant overhead in terms of circuit size.

Acknowledgements: -

This research was supported in part by Gennum Corporation, the Canadian Microelectronics Corporation, Micronet R&D, and the Natural Sciences and Engineering Research Council of Canada.

References: -

[1] H. C. Andrews and B. R. Hunts, *Digital Image Restoration*, (Englewood Cliffs, NJ: Prentice Hall, 1977).

[2] J. F. Cavanagh, *Digital Computer Arithmetic: Design and Implementation*, (McGraw-Hill, 1984).

[3] J.-S. Lee, Digital Image Enhancement and Noise Filtering by Use of Local Statistics, *IEEE Trans. Pattern Analysis and Machine Intelligence*, 1980, 165-168.

[4] C. M. Leung and W.-S. Lu, A Modified Wiener Filter for the Restoration of Blurred Images, *IEEE Pacific Rim'93*, 1993, 166-169.

[5] R. Leupers, Code Generation for Embedded Processors, *Proc. 13th Annual Int'l Synposium on System Synthsis*, September 2000, 173-179.

[6] J. S. Lim, *Two-Dimensional Signal and Image Processing*, (Englewood Cliffs, NJ: Prentice Hall, 1990).

[7] Altera Inc., *NIOS 3.0 CPU Data Sheet*, 2003.

[8] A. Peymandoust, L. Pozzi, P. Ienne and G. De Micheli, Automatic Instruction Set Extension and Utilization for Embedded Processors, *IEEE Computer Society*, 2003.

[9] M. J. S. Smith, Application-Specific Integrated Circuits, (Addison-Wesley, 1997).

[10] Tensilica Inc., *Xtensa Microprocessor Data Book – for Xtensa V (T1050) Processor Cores*, 2002.

[11] Open SystemC Initiative, *SystemC 2.0.1 Language Reference Manual*, 2003.