ASIP Design and Synthesis for Non Linear Filtering in Image Processing

L. Fanucci, M. Cassiano, S. Saponara
DIIEIT, University of Pisa,
via G. Caruso, I-56122, Pisa, Italy
l.fanucci @iet.unipi.it

D. Kammler, E. M. Witte, O. Schliebusch,
G. Ascheid, R. Leupers, H. Meyr
ISS, RWTH Aachen University,
Templergraben 55, D-52056, Aachen, Germany
kammler@iss.rwth-aachen.de

Abstract

This paper presents an Application Specific Instruction Set Processor (ASIP) design for the implementation of a class of nonlinear image processing algorithms, the Retinex-like filters. Starting from high level descriptions, first algorithmic optimization is accomplished. Then a processor architecture and an instruction set are customized with special respect to the algorithmic computations in order to achieve the specified timing at reasonable complexity. Taking advantage of the programmability of processor architectures, the flexibility of the system is increased, involving e.g. dynamic parameter adjustment and color treatment. ASIP implementation results in 0.13 μm CMOS technology are presented.

1. Introduction

In the most recent panorama [1] of digital system design, ASIPs are gaining ground to fill the gap between the highly optimized platforms (ASICs) and the more flexible solutions offered by DSPs. Since ASIPs are optimized towards certain applications, they combine the high performance and efficiency of a dedicated solution with the flexibility of a programmable solution.

ASIPs are flexible within an application domain, being able to accomplish a group of functionalities using a set of common operations. Therefore, it makes them more useful than ASICs in case of applications requiring a certain degree of programmability. Moreover, since the customization of the design is focused on the addressed application domain, they are more specialized and therefore more optimized than DSPs, being able to provide the right features in terms of timing performance, energy consumption and required area. Architecture Description Languages (ADLs) [2,3,4] offer the ASIP designer a quick and optimal design convergence by automatically generating the software tool-suite as well as the Register Transfer Level (RTL) description of the processor [5]. Recently, optimization techniques have been introduced, which make the generated RTL-code quality comparable to handwritten code [6]. Of course, while designing an ASIP, it is the designer's duty to do the trade-off of performance vs. flexibility in the most suitable way. Depending on the application, a more specialized or a more flexible ASIP may be desired.

ASIPs can be used in various kinds of scenarios like signal or video processing and in all applications involving some elaboration kernels that are regularly repeated, so that it is easier to spot an effective structure for an instruction set. In particular, in the video field, both in the en/decoding blocks and in the post-processing filtering, some innovative algorithms are spreading involving highly non linear operators [7,8]. This makes the hardware design harder than for the classical linear counterparts (FIR, IIR). By changing some algorithmic parameters, the same class of filters can be used for different application scenarios. For this goal, dedicated ASIC solutions are not suitable since they provide only very limited flexibility. In addition to that, video algorithms usually foresee a repetition of the same slice of operations for a high number of frames each composed of large pixel matrixes. Thus, DSP solutions are not acceptable, because high computational performance, low energy consumption and low silicon area are very important specifications in handheld and mobile scenarios. For the above reasons, the ASIP concept applies very well to video applications.

In this paper the implementation of the Retinex class of algorithms [7,8,9] by using an ASIP is presented as case study. The identification of the operation kernels and their mapping onto an instruction set are described. Then some special processor concepts that were used to achieve a good trade-off between performance and flexibility are described in more detail, highlighting the power of ASIPs. In particular, this case study will be used as a test bench for the ASIP implementation of non linear operators. The architecture has been developed using the ADL LISA [2].

Section 2 describes Retinex-like non linear operators and their implementation and verification. Section 3 presents the ASIP design referring to the features that make it an application specific design. In Section 4 the synthesis results and the performance of the system are shown. We conclude with a summary and an outlook.

2. Retinex-like non linear algorithms

In the Retinex theory, first proposed in [10], an image is expressed as the pixel-by-pixel product of the ambient illumination y and the reflectance r of the scene object. Based on the Retinex theory several non linear operators have been proposed in literature for image contrast enhancement, correction of images acquired in bad lighting conditions, control of dynamic in logarithmic sensors [7,8,11,12]. All these filters exploit a

3-9810801-0-6/DATE06 © 2006 EDAA

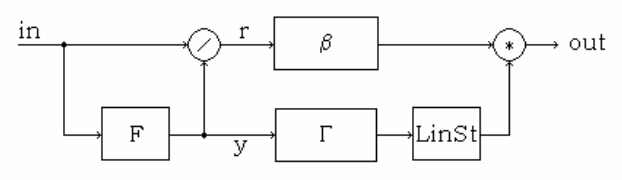

Figure 1. Block diagram of Retinex-based operators

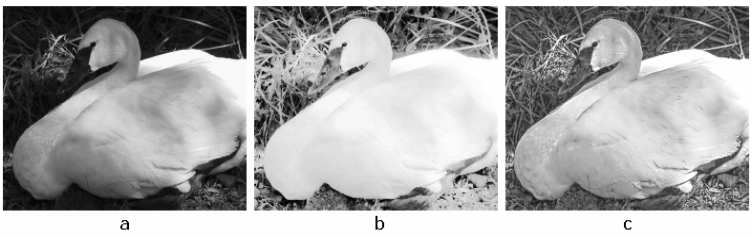

Figure 2. a) Original image, b) Histogram equalization, c) Retinex algorithm

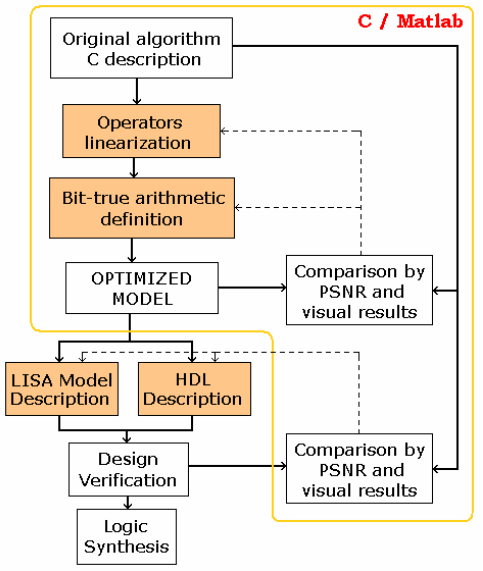

Figure 3. Optimization flow

similar structure sketched in Fig. 1: a non linear edge-preserving low-pass filter F is used to estimate the illumination y. Then the reflectance information r is obtained by division. These components are split to different elaboration chains which operate non linear point-to-point transformations, e.g. luminance correction Γ and reflectance enhancement β. After Γ transformation the illumination component is linearly stretched to perfectly cover the whole input range (0÷255 in case of 8-bit input pixel). Eventually the two components are recombined by multiplication. As an example, Fig. 2 shows a portion of an image acquired in bad lighting conditions (2a). The application of the classical histogram equalization brings to the result visualized in Fig. 2b. While trying to get the image clearer, a detail blurring comes up. The Retinex algorithm, instead, permits to obtain the effect in Fig. 2c solving the problems of image contrast and brightness together.

The description of F, Γ and β blocks in Fig. 1 involves the use of non linear operators reproducing the non linear behaviour of the human visual system. Here below, the equations for the Γ and β operators, depending on the shape parameters γ and b, are reported:

$$\Gamma(y) = 255 \cdot \left(\frac{y}{255}\right)^{\gamma\left(1+\frac{y}{255}\right)}, \quad \beta(r) = \frac{1}{1+e^{-b\cdot\log r}} + \frac{1}{2} \qquad (1)$$

The low-pass filter F is based on a recursive configuration whose coefficients are non linear functions of the input pixel in(n,m) and its neighbours within 3x3-sample masks. It needs four passes through the whole image to accomplish to its function.

$$y(n,m) = \frac{S_o \cdot f_o + S_v \cdot f_v + in(n,m)}{S_o + S_v + 1} \qquad (2)$$

being:
$$f_o = \alpha \cdot y(n-1,m) + (1-\alpha)\cdot in(n,m) \qquad (3)$$

$$f_v = \alpha \cdot y(n,m-1) + (1-\alpha)\cdot in(n,m) \qquad (4)$$

$$S_o = \frac{10^{-2}}{10^{-5} + \left(\log_{10}\left(\frac{in(n-1,m)+1}{in(n+1,m)+1}\right)\right)^2} \qquad (5)$$

A similar expression is available for S_v, evaluating the gradient through the m-direction. The α parameter controls the cut-off frequency of the filter. Non linearity plays its role both in Γ and β transformations and in the expressions of filter coefficients.

Dealing with such non linear operators requires to tailor a design flow to achieve the area/performance/energy specifications. First, an optimization stage should be addressed in a C/Matlab environment, as sketched in Fig. 3. For this purpose some effective methodologies for bit-true arithmetic definition and linearization of non linear operators have been developed requiring some pre-fixed optimization schemes based on piecewise linear and piecewise constant. In this stage two criteria are used to minimize the degree of the approximation: the objective criterion based on PSNR evaluation and the subjective one based on visual perception. These optimizations carried out at algorithmic level have been presented in [9] and are therefore omitted in this paper. Then the implementation can be addressed using the favourite approach among the different implementation methodologies. In Fig. 3 the ASIP and ASIC approaches are depicted, referring to their respective class of description languages: ADL and HDL. In case of ASIP design the HDL netlist is automatically generated from the LISA description. Hereafter, we present the implementation steps, choosing the ASIP path indicated in Fig. 3.

3. ASIP implementation for the algorithm domain based on Retinex

The Retinex algorithm presented in Section 2 is a flexible framework that can be applied to a large number of operations in the multimedia scenario [7,8,11,12]. The filter F in Fig. 1 can be programmed by tuning its

bandwidth or varying the number of passes through the image the filter will execute. The shape of the Γ and β transformations in equation (1) can be adjusted to achieve different image processing effects. Furthermore, the application can be switched between a linear and a logarithmic domain in which the multiplication/division operators are replaced with addition/subtraction operators [12]. The treatment of colored image by using the Retinex algorithm is still an open issue, but some intuitive solutions have been proposed according to the colour space. For instance, a possible procedure for RGB images is the parallel elaboration of the three components. For the YCrCb or YUV spaces, the Y component is processed while letting the others pass unchanged. Also the colour space conversion should be pursued as a capability of the system to be designed in order to cope with different input video sources.

These flexibility requirements make a programmable solution indispensable. But the demand for low elaboration time, low silicon area and high energy efficiency, as required in many applications like mobile devices or single-chip embedded systems in multimedia scenarios, forces to keep the advantages of a dedicated design. For these reasons, ASIPs offer an excellent trade-off, since the most repeated application kernels can be grouped in optimized hardware units, while keeping the activation of those hardware accelerators at a software level by the definition of a suitable instruction set for the ASIP programming.

3.1. A memory-dominated design

Like most multimedia applications, the design of video filtering architectures is dominated by the memory size and data transfer rate [13]. For example, an 8-bit per pixel VGA frame (640x480) needs 300 Kbytes. In case of video processing, it is often necessary to store more than one image since the previous frames are needed in elaboration (e.g. temporal filtering). Moreover, 8 bits are not sufficient to correctly represent the intermediate processing results: Extra bits are needed also for the fractional part [9]. Eventually, it can be the case that the image processing is split over several pipeline stages in order to increase the information throughput (see Section 3.2). To determine the required memory size to store the intermediate images, the memory size has to be multiplied with the number of used pipeline stages. Referring to VGA format, a worst case evaluation leads to a memory requirement of 10 Mbytes [14], unacceptable for systems designed for a single die. To reduce the memory amount, one way was pursued in the C/Matlab optimization step [9], by playing on the number of precision bits the trade-off between algorithmic performance and required memory. But lowering the number of fractional bits below the found optimum value can lead to a great worsening of the algorithmic performance. An effective way is to remove the pipelining at a frame level. This solution is based on a re-utilization of the same memory to store the intermediate data concerning the partially elaborated frames. The main drawback is, of course, the worsening of the throughput of the information, which is a critical specification item. Actually, real-time applications require a high throughput. Because of the trade-off between memory resources and data throughput, we decided to use two frame memories. This solution allows to keep a slight parallelism in the elaboration, since it is possible, for instance, performing the Γ and β transformations at the same time, without increasing the memory requirements too much compared to the simplest solution involving a single frame memory. In the case study, we used 8 integer bits and 6 fractional bits for data representation. Therefore, the total required memory for VGA format processing is 1.03 Mbytes. Moreover, there is a highly effective methodology to improve timing performance keeping the benefits of this memory organization. This is achieved by re-introducing a pipelining of the elaboration moving it from the frame level to the pixel level, which is more efficient in terms of memory usage. That allows for parallel elaboration of several pixels making the architecture timing efficient as well. Entering in more details about memory architecture implementation, Synchronous SRAM memories have been used for data storage. The two RAMs have been named X RAM and Y RAM. They are read scanning the whole image in order to produce the illumination component (y) according to the F filter functionality. This process requires four passes of the whole image and the intermediate results are stored in the Y RAM, while the X RAM contains the input image. After that both RAMs are further scanned and the reflectance component (r) evaluation is performed by division. Also the Γ and β transformations are performed. Then the Γ output is stored in the Y RAM, while the β output is stored in X RAM. In the end, a further scan of the two RAMs is required for the component recombination leading to the output image, which is finally stored in the X RAM. In all frame processing stages, a pipelining of subsequent pixels is used to speed up the architecture. Both, the particular memory organization and the data pipelining are important hardware customizations applying to the case study application. These sorts of customizations of memory and pipeline architecture are major advantages of ASIPs. Other resources utilized in the processor arithmetic can be customized according to the application needs, too. In the case study, 16 general purpose 32-bit registers have been instantiated. Some additional dedicated registers have been used for the storage of processing parameters which can be easily used during the elaboration. A fixed point arithmetic has been used for data representation and instructions have been coded using 32-bit instruction words.

3.2 Task concurrency: a pipelined architecture

As mentioned in the previous subsection, the pixel elaboration has been split over a pipelined architecture. This choice has the benefit of increasing the architecture parallelism and to shorten the critical path. This property of pipelined systems leads to an increased data throughput, which is highly desirable in our case. However, this strategy can have some drawbacks due to

increased latencies, silicon area overhead, e.g. by pipeline control and registers, and dependencies in the pipeline. Data dependencies can exist between neighbored instructions, that is, a result produced by an instruction may be used as an operand by the following instructions. Mechanisms to cope with data dependency problems are discussed in the next section.

Using ADLs, the design space is fully explorable with no restriction given by pre-designed parts or templates. Nevertheless, templates can be used as a first starting point, but the designer is not limited by that. Customizations of pipeline structure and memory architecture are presented in the following.

In the case study, seven pipeline stages have been introduced. After design space exploration this pipeline organization resulted as the best trade-off between throughput increase and complexity considering that the higher is the number of pipeline stages, the higher is the throughput but also the higher is the complexity and the risk of inter-dependencies and hence of pipeline stalls. Particularly, to understand why such a pipeline structure has been used we have to refer to a repeated optimization technique used all over the design: the piecewise approximation of non linear operators. Since this is a widely utilized functional kernel in the optimized application, some particular attention was paid to its implementation. As example, let us consider the piecewise linear technique used to approximate the Γ transformation in Fig. 4.

Since the throughput is a pressing specification, it is desirable having an instruction able to load an operand from the data memory, to perform the Γ transformation in the piecewise linear form and to store the result back to the data memory. The designed pipeline allows for the processing of such an instruction, using the following stages (Fig. 5):

- FE: the fetch stage in which the instruction is fetched from the program memory.
- DC: the decode stage in which the instruction is decoded, producing the control signals for the operating part.
- LD: the load stage in which the operand is loaded from the data memory.
- CMP: the comparison stage where the loaded operand is compared to the edges on the abscissa axis in order to identify the correct approximation interval.
- ROM: the ROM stage in which the result of the previous comparison is used to address a ROM from which to fetch the parameters (offset Q and slope K) of the correct piecewise segment.
- ARITH: the arithmetical stage in which the fetched parameters are used to calculate the output according to piecewise segment expression $K \times IN + Q$.
- WB: the write-back stage in which the output is stored back to the data memory.

The names assigned to each stage are mnemonical names applying to the presented particular case. Depending on the instruction, different operations can be executed in the stages, meaning e.g. the ROM stage is not used for ROM accesses exclusively.

The piecewise linear approach allows also for the implementation of the division operation with a throughput of one division per cycle, which is a great advantage for the system performance. Obviously, the performed division is a customized operation leading to acceptable results only operating on inputs in the working range. Otherwise the approximation introduced by our procedure would compromise the result. Nevertheless the LUT technique used for our customized division shows satisfactory results. In the division instruction, the LD stage is used to load the denominator from the data memory, the CMP stage is used to load the numerator and the WB stage to write the computed ratio back to the data memory (see memory accesses in Fig. 5).

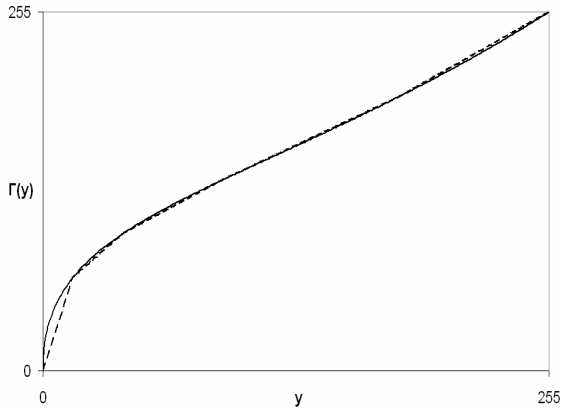

Figure 4. Γ transformation (straight line) and its piecewise linear approximation (dashed line)

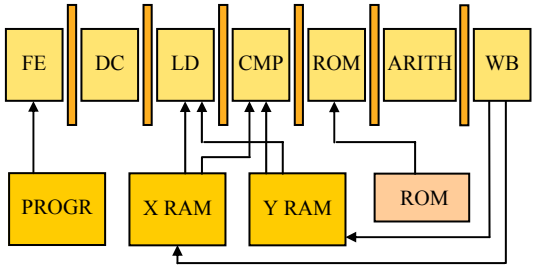

Figure 5. Pipeline structure and memory accesses

3.3 Data dependencies: the bypass mechanisms

Data dependencies are a problem related to the pipeline architecture. A 7-stage pipeline obviously leads to the following disadvantage of data hazards: in the LD stage an instruction ("consumer") may read from a shared storage (a general purpose register or a memory location), which is expected to be written by a previous instruction ("producer"). If the producer instruction has not yet reached the WB stage in which the final result is stored in the shared storage, the consumer instruction will load an outdated value. That will cause a completely wrong result. There are two standard solutions for this issue: pipeline interlocking and bypassing. Using interlocking, the instructions trying to access data, that has not yet been written back, causes the pipeline to be stalled partially. This causes unacceptable throughput degradation, especially in performance critical loops.

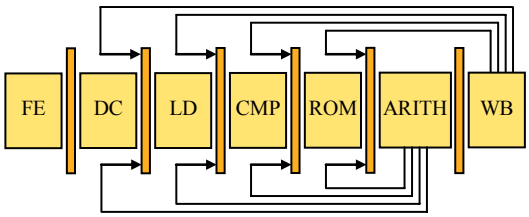

Figure 6. Implemented bypasses

This drawback can be solved by instruction rescheduling – either by the processor or by the compiler. This approach is usually strongly limited by the data and control flow. A more efficient way of resolving the data dependencies is to implement bypasses. Bypasses forward data immediately from a pipeline stage back to a previous stage. In the case study the majority of the instructions can provide the final result not before the ARITH stage. Therefore, two kinds of bypasses were implemented depending on the starting point of the bypass path: bypasses from the ARITH stage or bypasses from the WB stage. In both cases, more than one path was implemented depending on the end point of the bypass. They include: bypasses to the LD stage, the CMP stage, the ROM stage and the ARITH stage (Fig. 6). Most of the implemented bypasses are extensively used e.g. in the non linear filter F (Fig. 1), which implements one of the key elaboration steps of our case application.

3.4. Customized Instruction Set

One of the most important advantages of ASIPs is the fact that the instruction set can be customized according to the requirements of the application. This enables a trade-off between computational performance, silicon area and energy consumption. In order to increase the architecture efficiency it can be beneficial to implement complex pipelined instructions. This shortens the length of the final assembly program that is in our case study strictly related to the number of clock cycles needed for the complete elaboration.

Since the specific scenario is image/video processing, it is important to notice that there will be a portion of the assembly program (referred to as main loop) that has to be repeated a large number of times according to the image size (one iteration per pixel), typically in the order of hundreds of thousands of times. That means that a particular attention has to be paid to the number of program lines setting up the main loop, in order to avoid any waste of cycles and to maximize the throughput. In particular we show this for our case study in the following after giving a list of the most important instruction set customizations:

- Single instruction non linear transformations
- Automatic address calculation
- Zero overhead loops

For example, considering the address generation for the data memory, from the algorithmic specifications it can be noticed that some pre-fixed patterns are established iterating over the image. Thus an Address Generation Unit (AGU) calculating the next address for the data memory by incrementing the pixel pointer can be implemented in hardware. This is reflected in the syntax of several instructions by a short extension. Thus the address update is performed in parallel without the need of wasting cycles just to do the data address update.

Another observation is, that in conventional loop implementations comparisons and conditional branches create a significant instruction overhead and, even worse, cause pipeline control hazards. They lead to pipeline stalls and flushes. These problems can be avoided by implementing a loop mechanism in hardware. This is possible for loops being executed a pre-calculated number of times (equal to the image size). In this case it is enough to have a loop-parameter initialization before entering the loop and to manage the loop jumps by the hardware. This technique is known as zero-overhead loop implementation. With these implementation strategies, the programming is made easier and pipeline stalls and flushes resulting from control hazards can be eliminated.

The designed Instruction Set includes 42 instructions. They can be categorized in the following groups: non linear transformations (9), arithmetical computations (11), space colour conversions (6), memory accesses (9), processor initialization (6) and loop control (1).

4. Synthesis and performance

After the ASIP design has been fully carried out and its behaviour has been verified, a synthesis has been performed by means of Synopsys Design Compiler using a standard-cell CMOS 0.13 µm technology library with 1.2 V supply voltage. The requirements concerning the minimum clock frequency of the system are listed in Table 1 for the real time elaboration of YUV video sequences with a frame frequency of 24 Hz. The synthesis results show a 6.5 ns critical path located in the ARITH stage. That means that the maximum ASIP clock frequency is 154 MHz and this matches with real-time processing of CIF video formats up to 28 Hz. This is a satisfactory result considering that we moved to a programmable implementation approach (opposed to an ASIC) and that we were able to make the processor flexible but also efficient enough to allow for the outer control of elaboration parameters, output dynamic and for processing of coloured images represented in RGB, HLS, YCrCb or YUV spaces. The matching degree to the original algorithm is demonstrated by a PSNR of 30.7 dB. A speedup of a factor 1.8 can be achieved by performing only two passes of the filter F in Fig. 1 instead of four. This way real-time processing of 50 Hz CIF and 18 Hz VGA videos is allowed while the visual quality reduction is limited (e.g. 0.1 dB PSNR reduction for the image in Fig. 2). As far as the circuit complexity of the ASIP processing core is concerned, the synthesis results led to a complexity of 109 Kgates. A power simulation has been performed on gate level resulting in an average power consumption of 0.32 mW/MHz which corresponds to roughly 17 nJ/pixel. This number varies slightly with image size and aspect ratio. The ASIP

Format	f_{clock} - MHz	RAM - KB
QCIF (176×144)	32	86
CIF (352×288)	129	347
VGA (640×480)	391	1050

Table 1. Clock frequency and RAM to process different video formats (at 24 frames/s) in real time

performs well comparing the synthesis results with state-of-art implementations of similar non linear video filtering algorithms on DSP [15] or dedicated VLSI cells [16]. DSP-based implementations have been proposed in the literature for the real time elaboration of up to CIF videos but their power cost is in the order of watts, more than one order of magnitude higher than the ASIP power consumption. With respect to dedicated VLSI macrocells the ASIP stands for its higher flexibility while synthesis results are comparable.

The design has been also mapped on a DN6000K10s prototyping board equipped with a Xilinx Virtex-II Pro FPGA. Since the speed of the FPGA emulation is much higher than the speed of any RTL simulation we were able to process more test data (pictures) in a short time than we would have been using RTL simulations. This enabled us to carry out a complete life demonstration of the effects introduced by the algorithm on still images.

5. Conclusion

A complete ASIP design has been presented in this paper. The design process has been split over two steps: an algorithmic optimization step referred in Section 2 and a processor design detailed in Section 3. The main considerations that led to the designed architecture have been listed, leading from the memory organization to the architecture pipelining and, eventually, to the further customization of the architecture by the addition of some hardware features like bypasses, AGU and special structures for hardware looping. During the whole design the basic idea of the Instruction Set has been kept in mind as a guide for the hardware design. The synthesis performed on CMOS 0.13 μm technology showed that the ASIP performances are better than the results that could be obtained by a DSP implementation. Moreover, the processor architecture allows for a certain degree of flexibility compared to ASICs, involving the setting up of several parameters and the colour treatment.

Acknowledgements

This work has been partially funded by the European Commission through the Network of Excellence in Wireless COMmunications (Newcom).

References

[1] H. Peters, R. Sethuraman, A. Beric, P. Meuwissen, S. Balakrishnan, C. Pinto, W. Kruijtzer, F. Ernst, G. Alkadi, J. van Meerbergen, G. de Haan, “Application specific instruction-set processor template for motion estimation in video applications”, IEEE Tran. on Circuits and System for Video Tech., vol. 15, April 2005, pp. 508-527

[2] A. Hoffmann, H. Meyr, R. Leupers, *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002

[3] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, A. Nicolau, “EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability”. Proc DATE’99, Mar. 1999

[4] A. Fauth et al., “Describing Instruction Set Processors Using nML”. Proc. of the European Design and Test Conference (ED&TC), 1995

[5] O. Schliebusch, A. Chattopadhyay, D. Kammler, G. Ascheid, R. Leupers, H. Meyr, T. Kogel, "A Framework for Automated and Optimized ASIP Implementation Supporting Multiple Hardware Description Languages". Proc. ASP-DAC, Shanghai, China, Jan 2005

[6] O. Schliebusch, A. Chattopadhyay, E. Witte, D. Kammler, G. Ascheid, R. Leupers, H. Meyr, "Optimization Techniques for ADL-driven RTL Processor Synthesis", Proc. IEEE Workshop on Rapid System Prototyping (RSP), Montreal, Canada, June 2005

[7] G. Orsini, G. Ramponi, P. Carrai, R. Di Federico, “A modified retinex for image contrast enhancement and dynamic control”, Proc.IEEE ICIP03, Sept.03, pp.393-396

[8] S. Marsi, G. Ramponi, S. Carrato, “Image contrast enhancement using a recursive rational filter”, Proc. IEEE IST04, Stresa, Italy, May 2004, pp. 29-34

[9] S. Saponara, M. Cassiano, L. Fanucci, “Cost-effective VLSI Design of Non Linear Image Processing Filters”, Proc. DSD Euromicro 2005, Porto, Sept. 2005

[10] E. Land, J. McCann, “Lightness and retinex theory” Journ. of the Opt. Soc. of America, vol. 61, pp. 1-11, 1971

[11] M. Ogata, T. Tsuchiya, T. Kubozono, K. Ueda,“Dynamic range compression based on illumination compensation”, IEEE Trans. Cons. Electr., vol. 47, n.3, pp. 548-558, 2001

[12] S. Marsi, S. Carrato, G. Ramponi, B. Crespi, “Video dynamic range compression for logarithmic CMOS imagers”, Proc. COST 276 Workshop, Thessaloniki, Greece, May 2004

[13] F. Chatthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecapelle, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia Design*, Kluwer Academic, Boston, Mass, USA, 1998

[14] M. Cassiano, “Design of VLSI architectures for image quality improvements”, Master Thesis, University of Pisa, Italy, Dec. 2004

[15] L. Tenze, S. Carrato, C. Alessandretti, S. Olivieri, “Design and real-time implementation of a low cost noise reduction video system”, Proc. COST 254 Workshop, Neuchatel, CH, pp. 36-40, May 1999

[16] S. Saponara, L. Fanucci, P. Terreni, “Design of a low-power VLSI macrocells for nonlinear adaptive video noise reduction”, Journal on Applied Signal Processing, n. 12, Sept. 2004, pp.1921-1930