Adaptive Hardware Implementation for the Deblocking Filter Used in H.264/AVC Using System Generator

Kamel Messaoudi Amira Yahi Nawfel Messaoudi
Electrical engineers department
Mohemed Cherif Mssaadia University
Souk-Ahras, Algéria
kamel.messaoud@univ-annaba.org

El-Bay Bourennane Salah Toumi
LE2I Laboratory - Burgundy University, France
LERICA Laboratory – Badji Mokhtar University, Algeria
ebourenn@u-bourgogne.fr
salah.toumi@univ-annaba.org

***Abstract*—Xilinx System Generator is a Matlab/Simulink high-level based design tool especially for the development of complex digital circuits using Hardware Description Language (HDL). In this paper we propose a high level model for the deblocking filter used in H.264/AVC using System Generator of Matlab/Simulink. Synthesis results are compared with implementations realized using RTL level. The proposed model allows for rapid edits of the architectures and permits the implementation of filters used in other standards and norms (HEVC for example). The proposed implementations are verified using Xilinx-Virtex5 platforms.**

***Keywords*—Deblocking Filter; H.264/AVC codec; Hardware IPs; System-Generator; System-on-Chip.**

I. Introduction

The deblocking filter is one of new tools used in the H.264/AVC. This adaptive filter is used both in the encoder and in the decoder to eliminate the artifacts on the block boundaries [1]. In fact, the original frames used in H.264/AVC are partitioned into blocks and macroblocs of pixels and all processing are based on them which introduces artifacts on the block boundaries. This filter is used to increase the coding efficiency and to improve the decoded video quality [2].

Recently, several software and hardware implementations are proposed for the deblocking filter. The reconfigurable computing and FPGAs have been successfully used in recent years to implement complex algorithms into hardware, obtaining astonishing results compared to processor-based solutions in terms of performances and reusability [3]. The hardware implementations for the deblocking filter are based on various filtering orders to limit the access to memories in FPGA and to give the possibility of parallel processing [4][5][6][7]. Generally, these implementations used the hardware description languages (HDL) level which is still visible only by specialists with various difficulties encountered when switching between architectures. Edit the HDL codes remains a very difficult stage even for specialists who realize such codes. The use of graphical tools, such as Matlab/Simulink, has become a necessity in most implementations.

The System Generator of MATLAB/Simulink is a fully-featured tool for simulation programs for the FPGA. It is also possible to provide system modeling and automatic code generation from MATLAB/Simulink [8]. Using the Xilinx-specific blocksets, System Generator gives the ability to simulate a Simulink model and generate a synthesizable HDL. System Generator integrates RTL, embedded IP, MATLAB and hardware components. Using System Generator for DSP, developers with little FPGA design experience can quickly create and implemented FPGA designs.

The main idea of this work is to realize a new model for the deblocking filter using MATLAB/Simulink. This model is based on the Xilinx System Generator tool and we used the same hardware architectures proposed in [9][10] using the VHDL. System Generator allows self-generation of VHDL code for the deblocking filter from the initial Simulink model. The new model is beneficial where the objective is to implement the filter without requiring detailed knowledge of hardware design and HDL. It's also beneficial especially when we seek to edit the same implementation or when switching from one implementation to another. The advantage of this approach is also highlighted in terms of reducing concept-to-Silicon design time and effort.

The rest of this paper is organized as follows: in Section 2 we give an overview of the deblocking filter used in the H.264/AVC. In Section 3 we give the related works. In Section 4 we introduce the System Generation tool integrated in Matlab/Simulink. In Section 5 we describe firstly the hardware implementations used for the deblocking filter. Secondly, we detailed the new model for the deblocking filter based on System Generator. Simulation and synthesis results are given in this section. Finally, in Section 6 we present the conclusions and outline future work.

II. The deblocking filter used in H.264/AVC

The H.264/AVC was jointly developed in an open standard process by the world leading experts of the ITU-T Video Coding Experts Group and the ISO/IEC Moving Pictures Experts Group [11]. Actually, the H.264/AVC is well suited for various types of video services including mobile phone applications, broadcast SDTV and HDTV services via satellite, cable or terrestrial transmission, HD-DVD and Digital Cinema, etc. in-fact, the H.264/AVC achieves a significant improvement in coding efficiency when compared to other coding methods. It can save as much as 25% to 45% and 50% to 70% of bitrate when compared to MPEG-4 and MPEG-2 respectively [12].

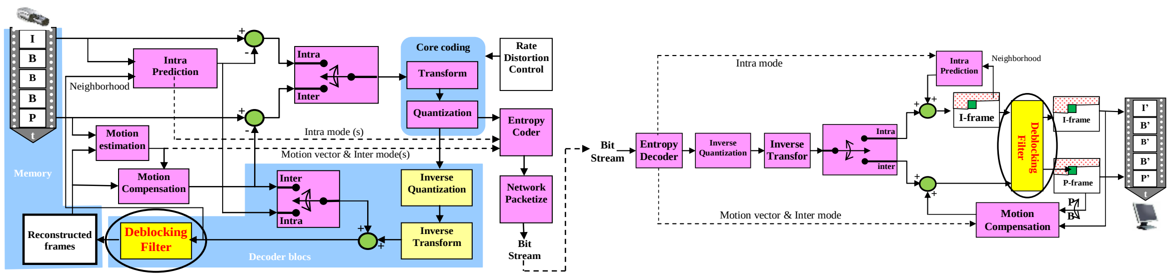


Fig. 1. The deblocking filter location in the H.264/AVC.

As shown in figure 1, the encoding system is composed of the forward path (encoding) and the inverse path (decoding). The forward path predicts each macroblock using intra-prediction or inter-prediction; it also transforms and quantizes the residual, then it forwards the result to the entropy encoder module. Finally, it generates the output packets in the NAL module. The inverse path involves the reconstruction of the macroblock from the previously transformed data by utilizing the inverse transform and quantization, the reconstruction module and the deblocking filter [13]. The decoder is composed practically by the same modules in the inverse path.

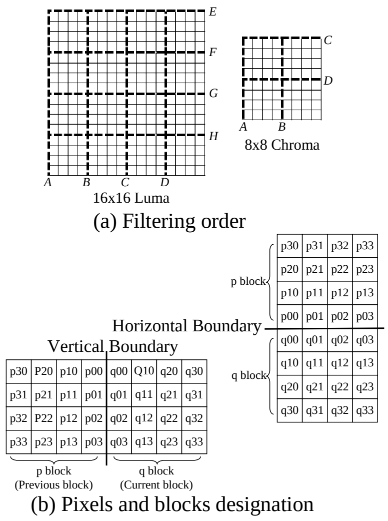
A. *The Deblocking filter*

H.264/AVC was proposed to take advantage of the temporal and spatial redundancy occurring in successive visual images [14]. In a video sequence, the video compression efficiency achieved by this standard is not the result of any single feature but rather a combination of a number of encoding tools and algorithms, one of these tools being the adaptive deblocking filter [1]. In-fact, H.264/AVC is a block-based coding system: the original frame is partitioned into blocks of pixels, and the algorithm performs the prediction, transformation and quantization based on them [4]. However, the use of block-based processing often introduces artifacts on the block boundaries [15]. For this reason the deblocking filter is used to decrease these artifacts, which increases the coding efficiency and improves the decoded video quality [2].

Previous video codecs utilize a post-filter only in the decoder to improve visual quality at the output. At the encoder, specifically at the motion-compensation module, unfiltered decoded frames are used as reference to reconstruct further frames [11]. In H.264, the filter is used in the encoder and in the decoder (Figure 1) in order to manipulate the same reference images [5]. However, the deblocking algorithm used in the H.264/AVC is more complex than the filter used in previous video compression standards [6]. Some of the complexities of this filter explained as follows. First of all, the H.264 deblocking filter is highly adaptive and applied to each boundary of all the 4×4 luma and chroma blocks of pixels in a 16×16 macroblock of pixels. Second, it can update 3 pixels in each direction, in which the filtering takes place. Third, in order to decide whether the deblocking filter will be applied to a boundary, the related pixels in the current and the neighborhood blocks must be read from memory. Because of these complexities, the deblocking filter can easily account for one-third of the computational complexity of an H.264/AVC [1].

B. *Software architecture of the DBF*

According to the H.264/AVC software reference, the deblocking filter module receives as input the reconstructed macroblocs of pixels, from the Inverse Transform and Quantization modules. These modules generate the reconstructed macroblock, one 4×4 block at a time. In H.264/AVC, the filtering stage is applied to each 4×4 block boundary, in a specific order, as shown in Figure 2.a. Vertical boundary edges (A, B, C and D) are filtered first, followed by the horizontal ones (E, F, G and H) [16]. All filtering steps take place from left to right and from top to bottom. Moreover, macroblocks are processed in a raster-scan order over the frame. The deblocking filtering process consists of modifying pixels at the four block edges by an adaptive filtering process. This process is performed using one of the five different standardized filters, selected through the means of a Boundary Strength (BS) calculation [11]. Figure 2.b defines graphically some notions employed in the deblocking Filter.



(b) Pixels and blocks designation

Fig. 2. Edge filtering order in macrobloc and pixels adjacent to boundaries.

- Boundary Strength

The boundary strength is obtained from the block type and some pixel arithmetic calculation are used to determine if the existing pixel differences along the block border are a natural image edge or an artifact [2]. Through this process, it is decided whether or not the filtering is necessary, and how much strength has to be applied. The filtering outcome depends on the BS and on the gradient of the image samples across the boundary [4].

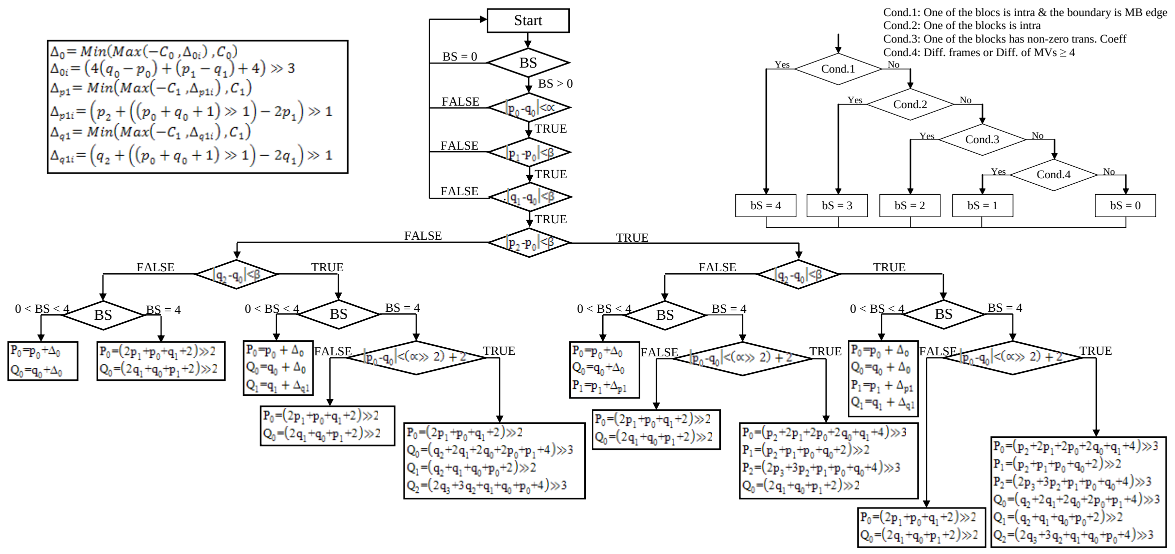

Start
BS
BS = 0
BS > 0
FALSE
TRUE
Cond.1: One of the blocs is intra & the boundary is MB edge
Cond.2: One of the blocks is intra
Cond.3: One of the blocks has non-zero trans. Coeff
Cond.4: Diff. frames or Diff. of MVs ≥ 4
Yes
No
bS = 4
bS = 3
bS = 2
bS = 1
bS = 0
0 < BS < 4
BS = 4

Fig. 3. The Deblocking filter algorithm used in H.264/AVC.

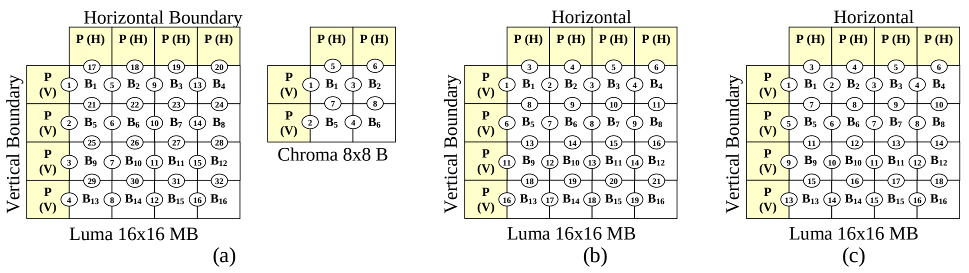


Fig. 4. Filtering order proposed by the H.264/AVC standard, by Li et al and by Chen et al.

The BS parameter is chosen according to the rules given by the standard, as shown in Figure 3. The result of applying these rules is the fact that the filter is stronger at places where blocking distortion is significant, such as the boundary of an intra coded MB or a boundary between blocks that contain coded coefficients [17]. When filtering a block boundary, eight pixels are involved and some of them may be modified according to the BS value. In H.264/AVC, BS is set to five different levels (0 to 4) and the bigger BS is, the stronger the filtering will be. When BS=0, no filtering is applied and none of the pixels are changed; when BS=4, the strongest filtering may modify six pixels in the process. When BS lies between 1 and 3 means some weaker filtering, modifying four pixels only [15].

- Filter Selection

Once the BS has been calculated in the block, the filtering of boundary samples is determined by analyzing each pixel on the block boundary. A Group of samples from the set (p2, p1, p0, q0, q1, q2) are filtered only if BS > 0 and |p0-q0| < α and |p1-p0| < β and |q1-q0|≤ β [1]. α and β are thresholds defined in the standard [17]. They increase with the average Quantize Parameter (QP) of the two blocks p and q. The effect of the filter decision is to 'switch off' the filter when there is a significant change across the block boundary in the original image. When QP is small, a very small gradient across the boundary is likely to be applied to image features. In contrast, block effects should be preserved and so the thresholds α and β are low. When QP is larger, blocking distortion is likely to be more significant and α, β are higher so that boundary samples are filtered [11].

Figure 3 shows the overall algorithm of the highly adaptive deblocking filter used in H.264/AVC. There are several conditions that determine whether a 4×4 block boundary will be filtered or not. There are additional conditions that determine the strength of the filtering for each 4×4 block boundary. This filter can change the values of up to 3 pixels on both sides of a block boundary depending on the outcomes of these conditions [1].

III. Related work

The most important restriction imposed by the deblocking filter used in H.264 CODECs is the filtering order of pixels. The sequential filtering order proposed by H.264/AVC [17] is shown in Figure 4.a. The restriction imposed is that if a pixel is involved in vertical and horizontal filtering, then the horizontal filtering should precede the vertical. This is rather loose and offers opportunities for implementation optimization by exploring different filtering schedules, aiming faster operation through the use of parallelism or at solutions that consume less memory. Several authors have proposed different filtering orders to limit the access to memory and to give the possibility of parallel processing. Khurana et al. [18]

proposed different filtering orders where horizontal and vertical filtering alternate. Sheng et al. [19] proposed an order where a higher frequency of vertical-horizontal filtering direction changes is observed. Li et al. [7] proposed another solution involving a degree of parallelism with vertical and horizontal filtering occurring at the same time, speeding up the filtering at the cost of having to use two filtering units. The scheduling for this solution is presented in Figure 4.b.

Chen et al. [15] proposed a new strategy with only 18 steps instead of 21 steps for the luminance (Figure 4.c). The authors used 4×4 pixel register, one transpose array, one 16×32Bit SRAM and two 1-D filter units (one for filtering the vertical boundary and the other for filtering the horizontal boundary). The use of two filters in pipeline architecture reduces the number of clock cycles required to process a macroblock of pixels to 120. Messaoudi et al. [9][10] proposed new hardware architectures for the deblocking filter used in H.264/AVC. These architectures use the same filtering order used in [15] where the two elementary filters (horizontal and vertical filters) are decomposed into four directional filters, each for one direction (left, right, top and bottom). An additional directional filter (vertical right) is used specifically to filter the left neighborhood blocks [9]. This technique also eliminates the need for the transpose circuit, simplifies the control unit and allows for pipelined architectures [10] particularly when using a 128-bit data bus [9]. In these implementations for the deblocking filter a new strategy for memory management is used. Several on-chip memories are employed to support efficient parallel access in order to speed up the entire filtering process.

IV. High-level modeling tools for system-on-chips

Computation intensive multidimensional data applications are more and more present in several domains such as image and video processing. These systematic applications are characterized by a very large amount of data-parallelism and the processing of multidimensional data arrays. Generally, real time and critical conditions should be ensured in these applications [3]. The modeling of highly repetitive structures in graphical form poses a particular challenge if a hierarchical approach is not adopted [20]. Currently, several high level modeling tools are used to perform embedded systems, namely: MDSDF (Multi-dimensional Synchronous Dataflow), Daedalus system-level design [21], GASPARD2 [22] and MATLAB/Simulink [23][24]. Using these high level modeling environments, several examples are given as to how the structure described is subsequently mapped into VHDL code [25]. These tools are beneficial where the objective is to implement the algorithms without requiring detailed knowledge of hardware design and hardware description languages.

MATLAB is interactive software proposed by MathWorks for numerical computations that simplifies the implementation of linear algebra routines. Powerful and matrix operations can be performed by using MATLAB commands. Simulink [23] is an additional MATLAB toolbox that provides for modeling, simulating and analyzing dynamic systems within a graphical environment [8]. Recently, MathWorks and Xilinx engineers have finalized tools specifically to generate HDL code from Simulink models containing both native Simulink blocks and Xilinx-specific blocks. In-fact, Xilinx System Generator is a MATLAB Simulink blockset that facilitates FPGA hardware design [23]. It extends Simulink in many ways to provide a modeling environment that is well suited to hardware design. System Generator provides access to underlying FPGA resources through low-level abstractions, allowing the construction of highly efficient FPGA designs. It integrates RTL, embedded, IP, MATLAB and hardware components. With System Generator for DSP, developers with less FPGA knowledge can create promptly FPGA implementation in a very short time compared to traditional RTL development one.

System Generator complements HDL design tasks by providing an easily configured test bench for both functional simulation and hardware verification. System Generator uses the built-in interface to HDL simulators like ModelSim to simulate the HDL codes within MATLAB environment. It also permits the Real-time hardware verification. The design can be tested in hardware at the targeted input rate and clocking frequency. The output of the hardware is captured into MATLAB and compared with the output test vectors. In-fact, we can used Xilinx-specific blocks for simulation, for code generation and for post-simulation. System Generator has a varied blocksets which can be automatically compiled to an FPGA target. For developers already familiar with HDL, System Generator provides additional advantages with even the possibility of incorporated already developed HDL modules using the Simulink Black-Box.

Figure 5 shows the System Generator design flow. System Generator works within the Simulink model-based design methodology [24]. Firstly, the application or the algorithm can be developed and implemented using the standard Simulink blocksets. Matlab/Simulink uses floating-point numerical precision and without hardware detail. This software implementation can be verified using Simulink simulation results. System Generator can be used to specify the hardware

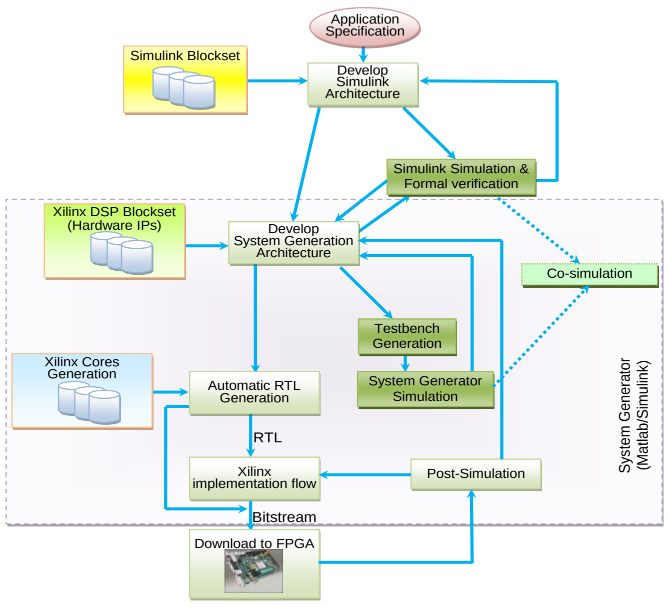

Application Specification
Simulink Blockset
Develop Simulink Architecture
Simulink Simulation & Formal verification
Xilinx DSP Blockset (Hardware IPs)
Develop System Generation Architecture
Co-simulation
Testbench Generation
Xilinx Cores Generation
Automatic RTL Generation
System Generator Simulation
RTL
Xilinx implementation flow
Post-Simulation
Bitstream
Download to FPGA
System Generator (Matlab/Simulink)

Fig. 5. System Generator design flow.

implementation details for the FPGA devices using Xilinx DSP blockset. The hardware implementation can be verified both using Simulink simulation and System generator simulation. Simulation results can be compared in order to propose necessary modifications. System Generator invokes Xilinx Core Generator to generate the vhdl code, the NGC file or the Bitstream for the specified FPGA device. As mentioned in [23] [24], several steps are required before and after the generation of synthesizable codes.

The authors in [25] present a good example concerning the methodology for implementing real-time applications on reconfigurable logic platform using Xilinx System Generator. A new architecture is detailed using System Generator for Color Space Conversion for video processing. The proposed methodology aims to improve the design verification efficiency for such complex system.

V. HARDWARE IMPLEMENTATION FOR THE FILTER

A. The used implementation

Firstly, we use hardware implementations for the deblocking filter proposed in [9] and [10]. We used RTL-level of these implementations. These implementations are realized based on four and five directional filters respectively. In [15] the main idea is the use of two elementary filters in pipeline architecture instead of a single filter used in other implementations. In [9][10], each elementary filter is divided into two directional elementary filters (left and right for the vertical filter and high and low for the horizontal filter). A fifth directional filter is added in [9] specifically to filter the left neighborhood blocks of the current macroblock of pixels (Figure 6). This subdivision allows us to better manage the internal memory; it also clearly separates the data input/output for each elementary filter. In addition, the two implementations utilize a 32-bit or 128-bit data width. The 128-bit data width is used in order to avoid the transposition circuits.

- Implementation strategy

As mentioned in Figure 7, the used hardware implementations for the deblocking filter are based on three steps:

– Defining a strategy for loading and storing blocks in order to limit the access to external memories.
– Defining a processing strategy using the elementary modules (elementary filter).
– Defining a processing strategy in each elementary filter in 3 stages: BS selection, filtering decisions and filtering implementation.

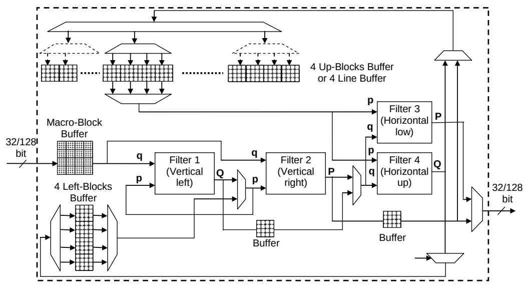
For the first step, two approaches are possible to save blocks of pixels being processed and their neighboring blocks of pixels: either through the use of internal memories or external memories. In the used implementations, we utilize internal memories to record the current macrobloc (16 blocks of pixels) being processed (macroblock Buffer in Figure 6), the four neighborhood blocks at the left (4 Left-Block Buffer in Figure 6) and all the neighborhood blocks in the same row at the top of the current macroblock (4 Up-locks Buffer or 4 Line Buffer in Figure 6). At the end of treatment of a macrobloc, the four blocks of pixels at the right are stored in the neighborhood left memory to serve as left neighborhood for the next macrobloc. The four blocks of pixels at bottom in the treated macrobloc are stored in the neighborhood top memory. These blocks are used to serve as top neighborhood of the next row of macrobloc s. This strategy has the following advantages:

– It limits the number of accesses to external memories, which means a gain from the perspective of processing time and reduction in power consumption.
– Memory reuse to save different successive blocks.
– Provides several reading strategies of data for the processing modules to improve data-parallelism.
– Uses memories with inputs/outputs of 128-bits, instead of registers. It does so by consuming only 36 Kbits memory, which represents a BRAM block in the used devices.

In Figure 7, BS values for each block of pixels are assumed to be provided by the upstream processing modules; these values are calculated based on the used processing method, the position and type of the processed block of pixels, and the selected partitioning strategy in the prediction modules of the encoder.

- The processing strategy used in deblocking filter

To implement the deblocking filter in the first architecture, we use four cascaded directional filters in order to obtain a parallel processing of blocks as indicated in figure 8.a. In the second architecture, an additional vertical-right filter is used in parallel with the vertical-left filter specifically to process the left neighborhood blocks of the current macrobloc (Figure 8.b). In this implementation, the two vertical right-filters run alternatively (in ping-pong manner), if the first works the



The implementation based on 4 filters

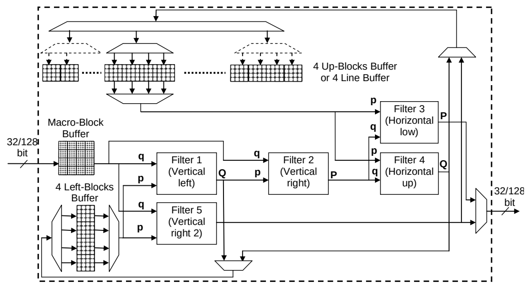

4 Up-Blocks Buffer or 4 Line Buffer
Macro-Block Buffer
32/128 bit
4 Left-Blocks Buffer
Filter 1 (Vertical left)
Filter 2 (Vertical right)
Filter 3 (Horizontal low)
Filter 4 (Horizontal up)
Filter 5 (Vertical right 2)
32/128 bit
p
q
P
Q
The implementation based on 5 filters

Fig. 6. The used hardware implementations for the deblocking filter [9][10].

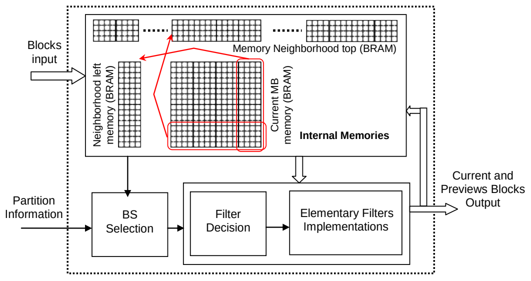

Blocks input
Memory Neighborhood top (BRAM)
Neighborhood left memory (BRAM)
Current MB memory (BRAM)
Internal Memories
Partition Information
BS Selection
Filter Decision
Elementary Filters Implementations
Current and Previews Blocks Output

Fig. 7. Deblocking filter implementation strategy.

other is set to idle by the selection command line. If the second works, the first is set to idle by forcing the value of BS to zero (no filtering). This is true for the blocks B4, B8, B12 and B16 (Figure 8.c), where the edges on the right are not filtered. Using an additional filter has the following advantages:

- We eliminated multiplexers that are between the elementary modules in figure 6.a. The selection conditions will be solely used on the control inputs.
- Intermediate buffers (which lie between the elementary filters) are also deleted.

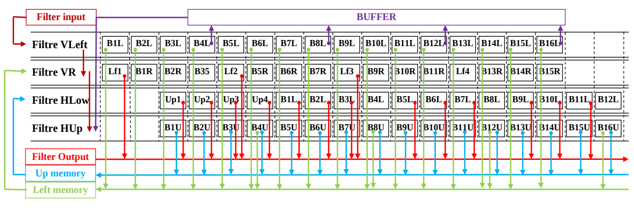
In Figure 8 (a and b) are depicted the filter ordering of different blocks of pixels in each elementary filter. We also show the order and timing of output data and intermediate blocks of pixels that will be recorded in the neighborhood memories. At the deblocking filter output, the following orders of the processed blocks are observed: V1 (Lf1), H1 (Up1), H2-4, V2, B1-3, V3, B5-11 (Figure 8.c). At the end of treatment, the other blocks will be recorded as follows:

- The blocks B4, B8, B12 and B16 are stored in the neighborhood-left memory to serve as neighborhood-left of the next macroblock.
- The Blocks V4, B13, B14, B15 and B16 are stored in neighborhood-top memory to serve as neighborhood-up of the next row of macroblocks.

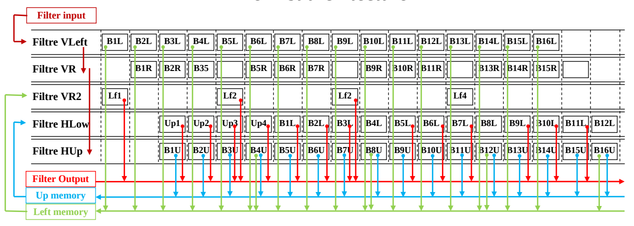
At the data output of the deblocking filter, it is necessary to add an addressing system to rearrange the processed blocks in their correct positions in macroblocks stored in external memories. Blocks at image boundaries are not filtered, because there are no neighbors. This is not the case for other blocks with neighbors in four directions. This causes an irregularity and, consequently, increases the complexity of the control unit of the filter. In order to avoid this inconvenience, we propose to apply the filter for all blocks and to just change the values of BS.

- Elementary filter hardware architecture

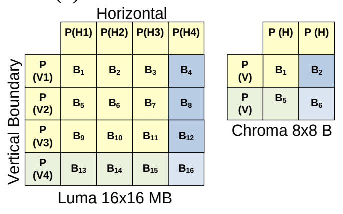
Each directional filter is used to process the current blocks of an input in one direction, according to the input values (BS, Alpha, Beta and tc0) calculated in the main module. As shown in Figure 2.b, for each filter operation, eight pixels (p3-0, q0-3) on both sides of the edge act as the input of the deblocking filter (Figure 9). By using the chart in Figure 3, the internal architecture is almost the same for the five directional filters. The filtering equations are the same; the only differences are theirs outputs which depend on filter direction.



The first architecture



(b) The second architecture



(c) Deblocking filter outputs.

Fig. 8. Timing and order of filtered blocks of pixels and filter outputs.

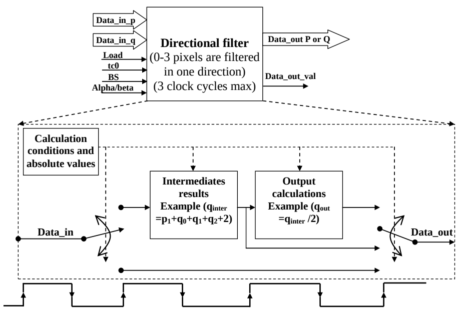


Fig. 9. Elementary filter inputs/outputs.

In each elementary filter, we propose to use registers for each condition to avoid the use of multiplexers. This increases the number of internal registers, but offers the possibility of pipelining the elementary filters. According to the proposed architectures, the pipeline is necessary in both horizontal filters to avoid overlapping blocks.

- Synthesis results

The two architectures decrypted in Figure 6 are implemented in vhdl and synthesized to Xilinx Virtex5 XUPV5 platform. We used ModelSim6.1 for simulation and ISE12.2 tool for project design and synthesis. The first architecture proposed in this work takes a maximum number of clock cycles equal to 75 cycles to process one macroblock (16x16 pixels). The second one takes a maximum number of clock cycles equal to 71, which is about 40% less than the best of the competing proposals. A comparison is given in Table I concerning the number of clock cycles necessary to process one macrobloc, the number of used elementary filters, the area occupied by the required amount of memory and the type of used memories to store neighborhood blocs.

TABLE I. VARIOUS IMPLEMENTATIONS PROPOSED FOR THE DEBLOCKING FILTER USED IN H.264/AVC.

	Cycles per MB	Filter Cores	Memory for current MB	Memory for neighb. blocks
H.264/AVC	192	1	512	Off-chip
Khurana et al. [18]	192	1	128	Off-chip
Sheng et al. [19]	192	1	80	Off-chip
Li et al. [7]	140	1	112	Off-chip
Chen et al. [15]	120	2	/	On-chip
Messaoudi et al [10]	59-75	4	256	On-chip
Messaoudi et al [9]	55-71	5	256	On-chip

Table II shows the synthesis results of the implemented filters. The filter in the first implementation consumes 9,274 LUTs and was able to run at 165.44 MHz. In the second implementation, the filter occupied 7,506 LUTs and was able to run at 170.95 MHz. Both proposals produce the same quality of filtered images. However, the addition of a fifth directional filter in the second implementation provides the following benefits:

– The elementary filters are connected directly without need for multiplexers. Therefore, the number of Luts used in the second proposal is decreased despite the use of a fifth elementary filter. In-fact, according to the results of synthesis, the number of Luts is reduced by 24%.
– The number of clock cycles required for processing a MB is reduced to 71 cycles in the second implementation; which is about 40% less than the best of the competing proposals.

TABLE II. SYNTHESIS RESULTS OF THE TWO IMPLEMENTATIONS.

	Four filters based implementation		Five filters based implementation	
	Used	%	Used	%
Number of Slice Registers	5,812	8	6,286	9
Number of Slice LUTs	9,274	13	7,506	10
Number of Block RAM/FIFO	10	7	12	8
Maximum operating frequency	165.44MHz		170.95MHz	
Clock cycles per macrobloc	59 – 75		55 – 71	

- The High-level model using system generator

Firstly, a black-box of each elementary filter is created as mentioned in figure 10. We felt that these elementary filters are already optimized using the vhdl and it is not necessary to reproduce them using the Simulink blockset. This is one of the highlights of System Generator that enables the use of black-box to insert hdl codes in Simulink. The simulation of the overall pattern will be carried out according to these codes. During generation of the overall code System Generator uses the hdl codes of each black-box and seeks other codes of Simulink blocks in the Xilinx blocksets.

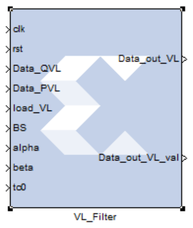


Fig. 10. Black box for one elementary filter (Vertical-Left Filter).

The next step is to connect the elementary filters to realize the two architectures presented in figure 6. We use memories for recording pixels of blocks being processed and neighboring pixels of neighboring blocks. We also use ROM for recording constant, multiplexers and de-multiplexers and counters for addressing the used memories, and comparators, concatenation operations, etc.

- Self-generation of HDL codes

After the creation and simulation of Simulink description, several steps are required before self-generation of hdl code. We define firstly the system 'clocking'. In this dialog the icon "FPGA clock period (ns)" defines the period in nanoseconds of the clock used. Secondly, we define the location of input / output on the FPGA plan (Allocation of pins). Knowledge of reserved addresses is needed from the datasheet each FPGA platform. Table III shows the synthesis results of the vhdl self-generated using System generator for the two implementations based on four and five elementary filters.

TABLE III. SYNTHESIS RESULTS OF THE TWO IMPLEMENTATIONS SELF-GENERATED USING SYSTEM-GENERATOR.

	Four filters based implementation		Five filters based implementation	
	Used	%	Used	%
Number of Slice Registers	6,214	9	6,084	9
Number of Slice LUTs	9,724	14	8,122	10
Number of Block RAM/FIFO	10	7	13	9
Maximum operating frequency	165.44MHz		170.95MHz	

- Simulation results

Matlab/Simulink allows simulation models performed a large number of tools in Simulink blocksets. We can also use other HDL simulators for the verification of self-generated codes. From Matlab/Simulink, we can, for example, appeal to the ModelSim simulator MenthorGraphics. This co-simulation is used to verify and compare the simulation results before going to the hardware implementation. In this work, we preferred the HDL simulation codes generated by System Generator using the same testbench files used for implementations of the first part. In-fact, all hardware architectures are simulated using a testbench file which can read images from extern files to feed them on the deblocking filter module and finally retrieve and rearrange the results. Figure 11 shows the simulation results of the proposed implementations. We used two images of Lena (256×256 pixels) : the first is a filtered image after compression-decompression using the DCT and inverse DCT with a compression ratio of 1/4 ; similarly for the second image with a compression ratio of 1/16. In the filtered images, we notice that the filter is adaptive; it allows filtering and smoothing surfaces away from image edges. On the natural image edges the filtering is conditioned by several parameters as mentioned in Figure 3. In these image areas, pixels are almost filtered or unfiltered to keep the contours of objects in the image. Other simulation methods are used to simulate the HDL in System Generator using a black box of the top VHDL entity as detailed in [8]. The co-simulation is realized using the ChipScope Analyzer. It's used where the complete HDL module is mapped onto hardware for real time simulation. The

ChipScope Analyzer utilizes the JTAG connections mapped to the Xilinx-XUPV5 board and load information from FPGA device.

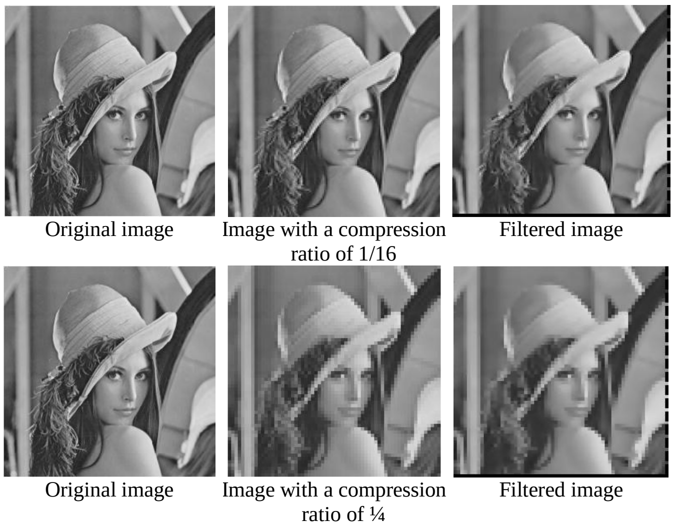

Original image
Image with a compression ratio of 1/16
Filtered image
Original image
Image with a compression ratio of ¼
Filtered image

Fig. 11. Simulation results of the proposed implementation.

VI. Conclusion

In this paper, we have presented and compared hardware implementations of the deblocking filter used in H.264/AVC. We have used two hardware implementations based on four and five elementary filters respectively using hdl level. The choice of using five elementary filters is made in order to reduce the number of clock cycles required to process a macroblock of pixels and to simplify the control unit circuitry, reducing as well the number of multiplexers and consequently the number of FPGA resources used. After, we have used System Generator to propose a high level model for each implementation. These models allow for rapid edits of the architectures and permit the implementation of filters used in other standards. Simulations and Synthesis results are compared with implementations realized using RTL level using Xilinx-Virtex5 platforms.

References

[1] M. Parlak, and I. Hamzaoglu. Low Power H.264 Deblocking Filter Hardware Implementations. IEEE Trans. on Consumer Elect., 54(2): 808-816, May 2008.

[2] K. YANG, C. ZHANG, and Z. WANG. Design of adaptive deblocking filter for H.264/AVC decoder SOC. Elsevier Science Direct, 16(1): 91-94, 2009.

[3] K. Messaoudi, E. B. Bourennane, S. Toumi, H. Mayache, N. Messaoudi, O. Labbani, ‘Use of the Array-OL specification language for self-generation of a memory controller especially for the H.264/AVC’, InderScience, International Journal of Embedded Systems, Vol. 7, No. 2, 2015, pp. 133-147, 2015.

[4] S. H. Shin, D. W. Oh, Y. J. Chai, and T. Y. Kim. Performance Improvement of H.264/AVC Deblocking Filter by Using Variable Block Sizes. Springer-Verlag Berlin Heidelberg 732-743, 2007.

[5] S. Wang, S. Yang, H. Chen, C. Yang and J. Wu. A Multi-core Architecture Based Parallel Framework for H.264/AVC Deblocking Filters. Springer, Sign Process Syst, 2008.

[6] P. List, A. Joch, J. Lainema, G. Bjøntegaard, and M. Karczewicz. Adaptive Deblocking Filter. IEEE Transactions on Circuits and Systems for Video Technology, 13(7): 614-619, 2003.

[7] L. Li, S. Goto, and T. Ikenaga. ‘A highly parallel architecture for deblocking filter in H.264/AVC’, IEICE Trans. on Inform.and Syst., E88(7):1623-1628, 2005.

[8] T. Saidani , D. Dia, W. Elhamzi, M. Atri and R. Tourki, ‘Hardware Co-simulation For Video Processing Using Xilinx System Generator’, in proceedings of the World Congress on Engineering (WCE 2009), Vol. 1, London, U.K, July, 2009.

[9] K. Messaoudi, E.B. Bourennane, S. Toumi and G. Ochoa, ‘Performance Comparison of Two Hardware Implementations of the Deblocking Filter Used in H.264 by Changing the Utilized Data Width’, IEEE conference, The 7th Inter. Work. on Syst. , Sig.Proc. and their Applic., Algeria, pp. 55-58, May 2011.

[10] K. Messaoudi, E. Bourennane, S. Toumi, M. Touiza, A. Yahi, ‘A Highly Parallel Hardware Implementation of the Deblocking Filter Used in H.264/AVC codecs’, Inter. Conf.on Soft. Eng. and New Tech., pp. 26-38, Tunisie, Dec. 2012.

[11] I. E. G. Richardson. H.264 and MPEG-4 Video Compression, John Willey & Sons. the Robert Gordon University, Aberdeen, UK, 2003.

[12] Chen, T., Lian, C. and Chen, L. ‘Hardware Architecture Design of an H.264/AVC Video Codec’, IEEE Vol.7D, No. 3, pp.750-757, 2006.

[13] K. Babionitakis, G. Doumenis, G. Georgakarakos, G. Lentaris, K. Nakos, D. Reisis, I. Sifnaios, ‘A real-time H.264/AVC VLSI encoder architecture’, Springer, Real-Time Image Proc, pp. 43–59, 2008.

[14] T. Wiegand, G. J. Sullivan, G. Bjøntegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. IEEE Transactions On Circuits And Systems For Video Technology 13(7):560-576, 2003.

[15] Z. CHEN, W. UAO, U. WANG, M. ZHANG, and W. ZHENG. A performance optimized architecture of deblocking filter for H .264/AVC. The Journal of China Universities of Posts and Telecommunications 14: 83-88, 2007.

[16] Xilinx. Inc. http://www.xilinx.com/support/documentation/ip_documentation/ h264_deblock_prodbrief_ds594.pdf.

[17] Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG. Draft of Version 4 of H.264/AVC (ITU-T Recommendation H.264 and ISO/IEC 14496-10 (MPEG-4 part 10) Advanced Video Coding), 2003.

[18] G. Khurana, A. A. Kassim, T. P. Chua, and M. B. Mi. A pipelined hardware implementation of In-loop Deblocking Filter in H.264/AVC. IEEE Transactions on Consumer Electronics, 52(2):536–540, 2006.

[19] C. C. Sheng, T. S. Chang, and K. B. Lee. ‘An In-Place Architecture for the Deblocking Filter in H.264/AVC’, IEEE Transactions on Circuits and Systems, 53(7): 530-534, 2006.

[20] S. Wood, D. Akehurst, G. Howells and K. M. Maier, ‘Array OL Descriptions of Repetitive Structures in VHDL’, 4th European conference on Model Driven Architecture: Foundations and Applications, pp. 137-152, Springer-Verlag Berlin, Heidelberg, 2008.

[21] A. D. Pimentel , T. Stefanov , M. Thompson , S. Polstra , E. F. Deprettere, ‘Tool Integration and Interoperability Challenges of a System-Level Design Flow: A Case Study’, Proceedings of the 8th international workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, pp.167-176, 2008.

[22] O. Labbani, J. L. Dekeyser, P. Boulet and É. Rutten, ‘Introducing Control in the Gaspard2 Data-Parallel Metamodel: Synchronous Approach’, International Workshop MARTES: Modeling and Analysis of Real-Time and Embedded Systems, Montego Bay, Jamaica, 2005.

[23] Xilinx doc., ‘Sys. Gen. for DSP User Guide’, UG640, Oct. 2012.

[24] Xilinx doc., ‘Sys. Gen. for DSP, Getting Started Guide’, UG639, Oct. 2012.

[25] S. K. Wood, D. H. Akehurst, W. G. J. Howells and K. M. Maier, ‘Mapping the Design of Repetitive Structures onto VHDL’, Inter.Workshop ModEasy'07, Forum on specification & Design Languages, Spain, pp. 13-16, Sept. 2007.