A Flexible, Hardware JPEG 2000 Decoder for Digital Cinema

Antonin Descampe, *Student Member, IEEE*, François-Olivier Devaux, *Student Member, IEEE*, Gaël Rouvroy, *Student Member, IEEE*, Jean-Didier Legat, *Member, IEEE*, Jean-Jacques Quisquater, *Member, IEEE*, and Benoît Macq, *Senior Member, IEEE*

Abstract

The image compression standard JPEG 2000 proposes a large set of features, useful for today's multimedia applications. Unfortunately, it is much more complex than older standards. Real-time applications, such as Digital Cinema, require a specific, secure and scalable hardware implementation. In this paper, a decoding scheme is proposed with two main characteristics. First, the complete scheme takes place in an FPGA, without accessing any external memory, allowing integration in a secured system. Second, a customizable level of parallelization allows to satisfy a broad range of constraints, depending on the signal resolution. The resulting architecture is therefore ready to meet upcoming Digital Cinema specifications.

Index Terms

Arithmetic coding, bit-plane coding, Digital Cinema, JPEG 2000, line-based, wavelet transform.

Manuscript received March 5, 2004. This paper was supported by the Walloon Region, Belgium, through the TACTILS project.

A. Descampe, F.-O. Devaux and B. Macq are with the Communications and Remote Sensing Laboratory (TELE), Université catholique de Louvain (UCL), Belgium. E-mail: {descampe,devaux,macq}@tele.ucl.ac.be.

G. Rouvroy, J.-D. Legat and J.-J. Quisquater are with the Microelectronics Laboratory (DICE), Université catholique de Louvain (UCL), Belgium. E-mail: {rouvroy,legat,jjq}@dice.ucl.ac.be.

A. Descampe is funded by the Belgian NSF.

A Flexible, Hardware JPEG 2000 Decoder for Digital Cinema

I. Introduction

The development and diversification of computer networks as well as the emergence of new imaging applications have highlighted various shortcomings in classic image compression standards, such as JPEG. Consequently the JPEG Committee decided to develop a new image compression algorithm : JPEG 2000 [1]. This standard has a much higher compression efficiency and enables inherently various features such as lossy or lossless encoding, resolution and quality scalability, regions of interest, error resilience,... A comprehensive comparison of this norm with other standards, performed in [2], demonstrates the functionality improvements provided by JPEG 2000.

The techniques enabling the features described above are a wavelet transform (DWT) followed by an entropy coding of each subband. In the JPEG 2000 baseline, the wavelet transform may use two filter-banks: a lossless 5-3 and a lossy 9-7. The entropy coding step consists of a context modeling and an arithmetic coding. The drawback of the JPEG 2000 algorithm is that it is computationally expensive, much more than the techniques used in JPEG [2]. This complexity can be a problem for real-time applications.

Digital Cinema (DC) is one of these real-time applications. As explained in [3], edition, storage or distribution of video data can largely benefit from the JPEG 2000 feature set. Concerning compression efficiency, the very high quality required by such application makes the JPEG 2000 intra-frame compression scheme a valuable alternative to inter-frame coding schemes from the MPEG family. Moreover, a video format called Motion JPEG 2000 has already been designed, which encapsulates JPEG 2000 frames and enables synchronization with audio data [4].

In the DC scenario, the movie is compressed and ciphered off-line and then transmitted securely to the movie theater. The movie is stored locally and has to be loaded in real-time at each screening. This process includes a decryption, a decompression and synchronization of the image and audio streams, a possible overlay addition (i.e. subtitles) and watermarking. Among these tasks, the decompression of each JPEG 2000 frame is the most complex one and requires most of the resources to stand a chance to satisfy the targeted throughputs.

In this paper, a hardware JPEG 2000 decoder architecture intended for Digital Cinema is presented. It has been designed in VHDL, and synthesized and implemented in an FPGA (Xilinx XC2V6000-6 [5]).

It should be noted that nothing in this architecture prevents to implement it as an ASIC. Nevertheless, a FPGA was chosen because we believe its flexibility is more adapted to the relatively small DC market. Although standards are being prepared, technical requirements are indeed very likely to continue to evolve. Moreover, recent improvements in the technology such as the integration of microprocessors or high-speed IOs further increase the interest for this kind of platform. The proposed architecture decodes images line by line without accessing any external memory, allowing integration in a secured system. It is highly parallelized and depending on available hardware resources, it can easily be adapted to satisfy various formats, and specific constraints like secure decoding, lossless capabilities, and higher precision (over 8 bits per pixel-component). The three main blocks of the architecture are an Inverse DWT block (IDWT), a Context Modeling Unit (CMU) and an Arithmetic Decoding Unit (ADU).

Concerning the DWT part, many architectures have been published in the literature. Fast and efficient designs are based on the lifting scheme [6] and consist of separated or combined 5-3 and 9-7 transforms [7]–[10]. The last two above-mentioned papers also propose low-memory wavelet image compressions based on a line-based transform. The most recent and efficient work [10] details an architecture combining 5-3 and 9-7 wavelet filters with one decomposition level and also proposes a solution to minimize the number of lines kept in internal buffers. Concerning the wavelet part of our paper, a complete implementation of the inverse discrete wavelet transform (IDWT) with five levels of recomposition is achieved. In order to meet lossless real-time applications and cost constraints, our design is focused on the 5-3 transform and is based on reduced internal memories.

Some papers detail a complete hardware entropy coding [11]–[13]. Each of these papers propose a different and interesting design approach to the entropy coding, but are all based on ASIC technology. Our approach is optimized for FPGA and is based on the parallel mode defined in JPEG 2000. This parallelization allows us to propose an innovating approach. Our context modeling part deals with three pass blocks in parallel and, compared to [11], reduces the RAM used by 25%. References [12] and [13] also propose an arithmetic encoding architecture. Contrary to them, we benefit from the parallel mode mentioned above, which significantly improves the global throughput of the entropy decoding chain.

Complete implementations have also been recently described in one academic paper [12] and at least two industrial papers [14], [15]. Nevertheless, the commercial papers show the global architecture only briefly and give direct results without important design details. The flexibility of our decoder and the large image size managed in real-time make our paper an attractive solution for the upcoming Digital Cinema specifications.

The rest of the paper is organized as follows. Section II briefly describes the JPEG 2000 algorithm. In

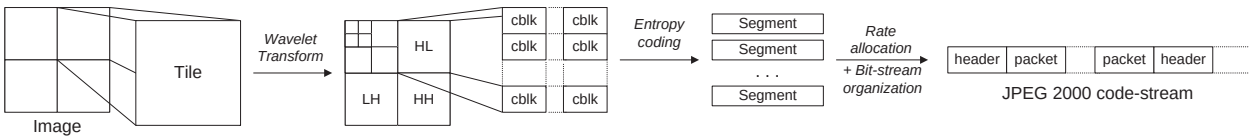

Fig. 1. JPEG 2000 coding steps.

Section III, we present our decoder architecture as well as our implementation choices. The main blocks of the architecture are described in more detail in Sections IV to VI. The performance of the system is discussed in Section VII and the paper is concluded in Section VIII.

II. JPEG 2000 BASICS

A. Algorithm overview

In this Section, concepts and vocabulary useful for the understanding of the rest of the paper are presented. For more details, [1] or [16] should be referred to. Readers familiar with JPEG 2000 should skip this Section. Although a *de*coder architecture has been implemented, *en*coding steps are explained here because their succession is easier to understand. The decoding process is achieved by performing these steps in reverse order. Fig. 1 presents the coding blocks that are explained below.

First of all, the image is split into rectangular blocks called tiles. They are compressed independently. An intra-component decorrelation is then performed on the tile: on each component a *discrete wavelet transform* is carried out. Successive dyadic decompositions are applied. Each uses a bi-orthogonal filter bank and splits high and low frequencies in the horizontal and vertical directions into four subbands. The subband corresponding to the low frequencies in the two directions (containing most of the image information) is used as a starting point for the next decomposition, as shown in Fig. 1. The JPEG 2000 Standard performs five successive decompositions by default. Two filter banks may be used: either the *Le Gall* (5,3) filter bank, for lossless encoding, or the *Daubechies* (9,7) filter bank, for lossy encoding. This part is further detailed in Section VI-A.

Every subband is then split into rectangular entities called code-blocks. Each code-block is compressed independently using a *context-based adaptive entropy coder*. It reduces the amount of data without losing information by removing redundancy from the original binary sequence. “Entropy” means it achieves this redundancy reduction by using the probability estimates of the symbols. Adaptability is provided by dynamically updating these probability estimates during the coding process. And “context-based” means

that the probability estimate of a symbol depends on its neighborhood (its "context"). Practically, entropy coding consists of

- *Context Modeling*: the code-block data is arranged in order to first encode the bits which contribute to the largest distortion reduction for the smallest increase in file size. In JPEG 2000, the Embedded Block Coding with Optimized Truncation (EBCOT) algorithm [17] has been adopted to implement this operation. The coefficients in the code-block are bit-plane encoded, starting with the most significant bit-plane. Instead of encoding the entire bit-plane in one coding pass, each bit-plane is encoded in three passes with the provision of truncating the bit-stream at the end of each coding pass. During a pass, the modeler successively sends each bit that needs to be encoded in this pass to the Arithmetic Coding Unit described below, together with its context. The EBCOT algorithm is further detailed in Section IV-A.
- *Arithmetic Coding*: the Context Modeling step outputs are entropy coded using a MQ-coder, which is a derivative of the Q-coder [18]. According to the provided context, the coder chooses a probability for the bit to encode, among predetermined probability values supplied by the JPEG 2000 Standard and stored in a look-up table. Using this probability, it encodes the bit and progressively generates code-words, called segments. This algorithm is further detailed in Section V-A.

During the *rate allocation* and *bit-stream organization* steps, segments from each code-block are scanned in order to find optimal truncation points to achieve various targeted bit-rates. Quality layers are then created using the incremental contributions from each code-block. Compressed data corresponding to the same component, resolution, spatial region and quality layer is then inserted in a packet. Packets, along with additional headers, form the final JPEG 2000 code-stream.

B. Complexity and hardware considerations

The main weakness of JPEG 2000 is its complexity. As shown in [13], this complexity is mainly due to the entropy coder, which requires over half the computation time. This can be explained by the bit-level processing of the JPEG 2000 entropy coder, as opposed to the JPEG Huffman coder, which only deals with entire samples. An N-bits sample is processed as N distinct samples by the EBCOT. Moreover, each bit-plane is entirely scanned by three successive passes, while each bit of this bit-plane is encoded only once, in one of the three passes. If we assume that each pass spends one clock-cycle on each bit of the bit-plane, each sample will then need $N * 3$ clock-cycles to be processed by the EBCOT, while one single clock-cycle will usually be sufficient in the other steps (like the DWT). Finally, this huge amount of clock-cycles needed by the entropy coder is even greater in the decoding scheme: a feedback loop that

TABLE I

RESOURCES REQUIRED TO REACH DIGITAL CINEMA THROUGHPUTS FOR EACH IMAGE PROCESSING BLOCK. THE FRAME DECOMPRESSION IS THE BOTTLENECK.

	Slices	**RAM [Kbits]**
Decryption [20]	200	54
Watermarking [21]	2500	72
Decompression (Our)	27 000	1602

does not exist at the encoder side forces the EBCOT to wait for the MQ-decoder answer before being able to further process the code-block. This means that main parts of the EBCOT and MQ algorithm cannot be executed concurrently, which implies additional idle time.

Concerning the DWT part, the computational complexity is much lower, but the memory requirements might be very high as tiles are handled entirely (in comparison with the 8×8 DCT-blocks in JPEG).

Eventually, the system level raises several implementation problems. In particular, resources required to interface the entropy coding and DWT sub-systems are non-negligible. In a decoding scheme which is the one of interest in this paper, samples are produced by the entropy coder, bit-plane by bit-plane, one code-block at a time. Those samples are then processed by the Inverse DWT, coefficient by coefficient, one line at a time. This difference in the way each sub-system processes data either implies tight synchronization between those sub-systems, or additional memory resources to let them work independently ([19], p.690).

Fortunately, well-chosen encoding options and hardware implementation choices can help face this complexity. Depending on the constraints of the application targeted, several trade-offs between area, throughput and compression efficiency can be found. In this paper, we have focused on the Digital Cinema application. Choices made in this framework are detailed in the next Section.

III. PROPOSED ARCHITECTURE

Among all the tasks that have to be carried out to provide a complete JPEG 2000 Digital Cinema solution, we are only concerned with the frame decompression in this paper. This task is indeed the bottleneck of such a solution. Other image processing blocks, such as decryption [20] or watermarking [21] (see Fig. 2), are far less expensive if we want to reach Digital Cinema throughputs. Table I compares the resources needed to achieve each of these operations in a Digital Cinema framework.

In this Section, we first present the constraints used for our JPEG 2000 decoder architecture. Then, implementation choices made to meet these constraints are explained. Finally, the complete architecture

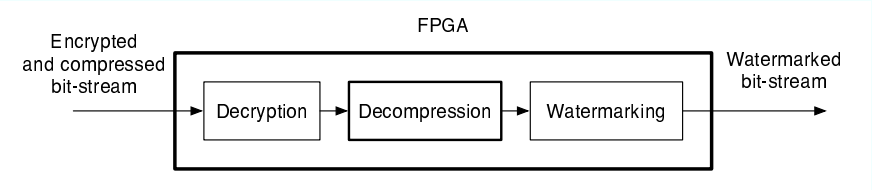

Fig. 2. A secure decoding scheme.

is presented.

A. Constraints

As our decoder is designed for real-time Digital Cinema processing, three main constraints have been identified:

- *High output bit-rate*: all implementation choices have been made to increase the bit-rate. Using the Xilinx XC2V6000-6, we wanted our architecture to meet at least the 1080/24p HDTV format. This means an output rate of about 1200 megabits per second (Mbps) for 8-bit 4:4:4 images.
- *Security*: no data flow may transit outside the FPGA if it is not crypted or watermarked. This constraint enables a completely secured decoding scheme, as the decompression block might be inserted between a decryption block and a watermarking block, these three blocks all being in the same FPGA (Fig. 2).
- *Flexibility*: computationally intensive parts of the decoding process must be independent blocks which can easily be duplicated and parallelized. This allows the proposed architecture to meet a broad range of output bit-rates and resource constraints. The design can therefore easily be adapted to upcoming Digital Cinema standards.

B. Implementation choices

To meet these constraints, the following implementation choices have been made. Note that some of these choices imply specific encoding options: the corresponding command-line is detailed in the Appendix.

No external memory has been used, meeting the security constraint and also increasing the output bit-rate, as the bandwidth outside the FPGA is significantly slower than inside. As internal memory resources are limited, large image portions cannot be stored and the decoding process must be achieved in a line-based mode.

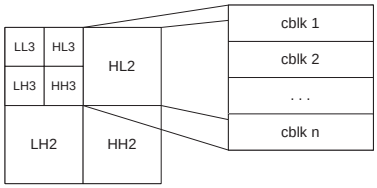

Fig. 3. Customized code-block dimensions.

To increase the output bit-rate, three *parallelization* levels have been used. The first is a duplication of the entire architecture, which allows various tiles to be simultaneously decoded. The second parallelization level tries to compensate the computation load difference between the Entropy Decoding Unit (EDU) and the IDWT. This is possible as each code-block is decoded independently. Finally, a third level of parallelization, known in the JPEG 2000 Standard as the parallel mode, is obtained inside each EDU. By default, each bit-plane is decoded in three successive passes but, by specifying some options ([19], p.508) during the encoding process, it becomes possible to decode simultaneously the three passes. This implies that each EDU contains one Context Modeling Unit (CMU) and three Arithmetic Decoding Units (ADU).

Another encoding option that increases the throughput is the *bypass mode* ([19], p.504). The more correlated the probability estimates of the bits to encode are, the more efficient the ADU is. This is especially the case in the most significant bit-planes while the last bit-planes are totally uncorrelated most of the time. With the bypass mode enabled, these latter bit-planes are therefore raw-coded[1].

Some choices about *image partitioning* have also been made. A 512×512 tile partition avoids the use of external memory and enables the first parallelization level mentioned above. Inside each tile, even if the maximum code-block size specified in the standard is 4,096 pixels, it does not exceed 2,048 pixels in our implementation. As we will see, this does not induce any significant efficiency loss but allows a 50% economy on memory resources.

Furthermore, the code-block dimensions have been chosen so that each systematically covers the width of the subband to which it belongs (Fig. 3). As the IDWT processes the subband data line by line, such code-block dimensions enable a line-based approach of the overall process, reducing the size of the image portions to be stored between the EDU and the IDWT.

These last implementation choices (parallel mode, bypass mode and image partitioning) imply an

[1]This means they are inserted as such in the bit-stream.

TABLE II

AVERAGE PSNR FOR FIVE 512X512 GRAY-SCALED IMAGES (*Lena*, *Boat*, *Goldhill*, *Barbara* AND *Woman*), 8 BPP.

Compression ratio	**PSNR [dB]**	
	Default options	**Options used**
1:4	42,33	41,76 (-1.35%)
1:8	37,42	36,75 (-1.78%)
1:16	33,36	32,67 (-2.08%)
1:40	29,17	28,58 (-2.04%)
1:80	26,74	26,20 (-2.01%)
1:160	24,77	24,29 (-1.93%)

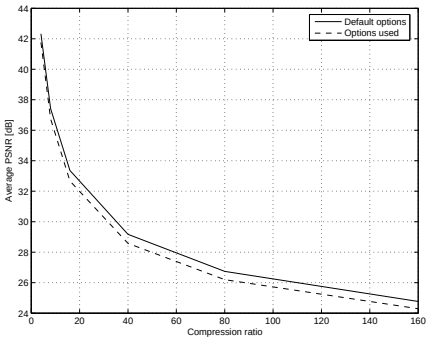

Fig. 4. Average PSNR vs Compression ratio for the five test images and the two encoding options sets.

efficiency loss during the encoding process. Table II and Fig.4 show the corresponding *PSNR* losses for various compression ratios. Five 512x512 gray-scaled images (*Lena*, *Boat*, *Goldhill*, *Barbara* and *Woman*) are used for the tests. In comparison with the improvements provided by these choices, quality losses are reduced, especially for small compression ratios, which are the ones used for Digital Cinema.

To allow the IDWT to process the image in a line-based way, the bit-stream is organized so that the whole compressed data corresponding to a specific spatial region of the image is contiguous in the bit-stream. Such a *data ordering scheme* corresponds to one of the five progression orders proposed in the JPEG 2000 Standard.

A last implementation choice aims at achieving some lightweight operations in software. These operations are mainly data handling and are easily implemented using pointers in C. Headers and markers (needed by these operations) are therefore not ciphered: only packet bodies are, keeping the decoding

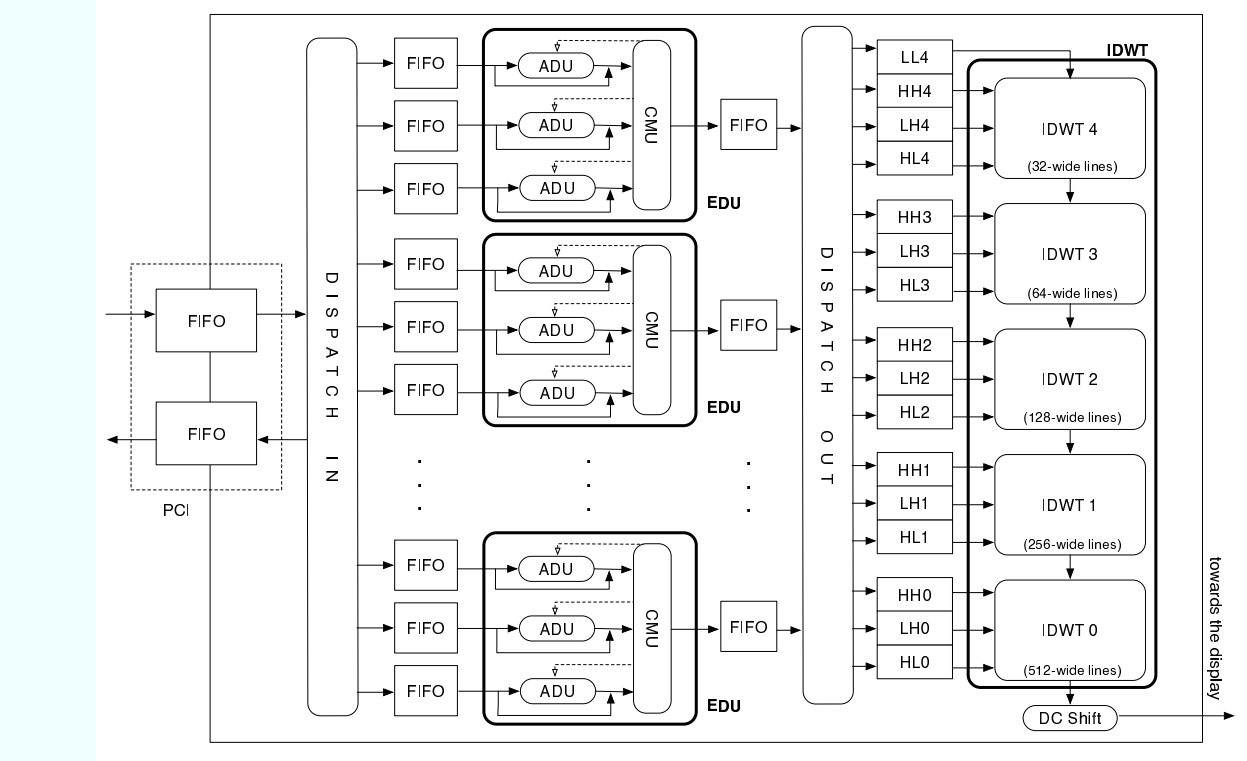

Fig. 5. Proposed architecture.

process secure.

As can be observed, some options, known by any universal JPEG 2000 encoder, must be specified during the encoding process. Our architecture is unable to decode a JPEG 2000 code-stream that has not been encoded using these options. As this architecture is dedicated to decode Digital Cinema streams at the highest output bit-rate, we did not consider it efficient to produce a universal decoder. The images used to test our architecture were generated using the OpenJPEG library [22]. The corresponding command-line is given in the Appendix.

C. Architecture

Fig. 5 presents the hardware part of our architecture. Each EDU contains three ADUs reflecting the parallel mode. The bypass mode is also illustrated by the bypass line under each ADU. The Dispatch IN and OUT blocks are used to dissociate the entropy decoding step from the rest of the architecture and to enable the flexibility mentioned above. When Dispatch IN receives a new JPEG 2000 code-stream from the PCI, it chooses one of the free EDUs and connects the data stream to it. Dispatch OUT retrieves

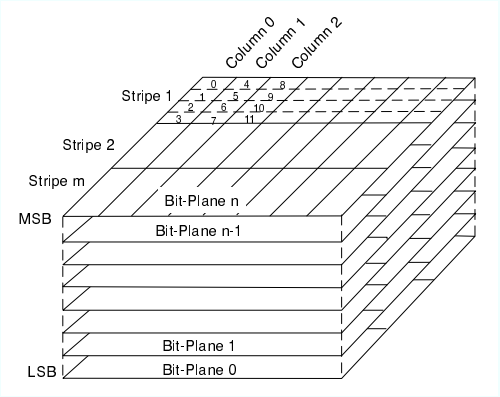

Fig. 6. Scanning order of the coding passes. The code-block is viewed as a succession of bit-planes structured in stripes.

decompressed data from each EDU and connects it to the correct subband FIFO. In this way, a maximum of EDUs are always used simultaneously. ADU, CMU and IDWT blocks are explained in more detail below.

IV. Context Modeling Unit

A. EBCOT algorithm

In the EBCOT algorithm, each code-block is encoded along its bit-planes, beginning with the most significant one. Three coding passes (called *significant*, *refinement* and *cleanup pass*) successively scan each bit-plane and select the bits to encode. This selection is based on the *significance* of the coefficients. A coefficient has an insignificant state at the beginning of the code-block encoding process, and becomes significant at a given bit-plane when its value in that bit-plane is 1 for the first time.

To process a bit-plane, the *significance pass* encodes insignificant coefficients with significant neighbors, then the *refinement pass* encodes already significant coefficients and finally the *cleanup pass* encodes the remaining coefficients. As shown in Fig. 6, the passes process the bit-planes in stripes, each consisting of four rows and spanning the width of the code-block.

For each bit to encode, the coding pass sends to the Arithmetic Coding Unit a pair of data: the value of the bit and its context. The context of a bit is based on the state of its neighbors in the bit-plane.

The decoding process is very similar to the encoding. The main difference is that the CMU only sends a context to the ADU and waits for the corresponding decoded bit, sent back by the ADU.

In order for each pass to select the bits and calculate its context, we have associated three state variables with each coefficient of the code-block and two with each bit of the bit-planes, as suggested in [23]. The

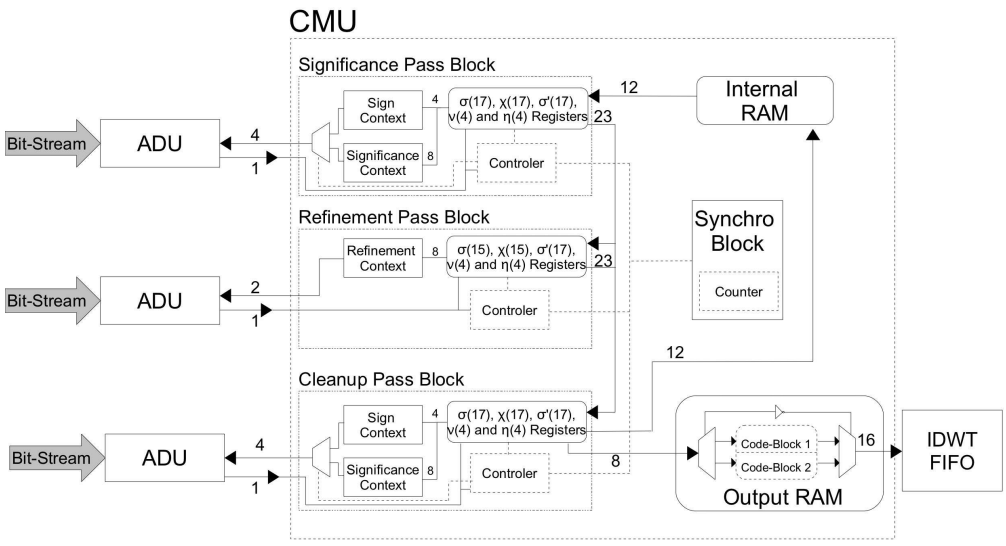

Fig. 7. The CMU architecture.

five state variables are:

- σ, corresponding to the coefficient state.
- χ, corresponding to the coefficient sign.
- σ' whose value changes from 0 to 1 when the coefficient is encoded for the first time by the refinement pass.
- ν corresponding to the value of the coefficient in the bit-plane.
- η indicating whether the coefficient has already been encoded in the bit-plane.

B. CMU architecture

A simplified view of our CMU architecture is presented in Fig. 7 and is based on that developed by Andra *et al.* in [11]. Each pass block is linked to an ADU (described in Section V) and contains one or two context calculation blocks as well as four registers handling the state variables. The state variables are loaded to the significance pass block from an Internal RAM and are transmitted from one pass block to another, as explained below. The global EDU decoding is organized by the Synchronization Block which generates control signals for the three passes. The decoded code-block is progressively sent to an Output RAM connected to the IDWT FIFOs (described in Section III-C).

The *Pass Blocks* have been designed in such a way that each bit-plane is decoded simultaneously by

the three blocks, as enabled by the parallel mode. As illustrated in Fig. 8, the *significance pass* is the first to decode the stripe and update the state variables. At the same time, the *refinement pass* analyzes the bits located two columns away, in order to use these updated variables in the context calculation (which requires the two columns surrounding the decoded one). The same shifting is applied to the *cleanup pass*. As described in the lower part of the figure, once a bit has been analyzed in a pass (and decoded if the state variables indicate so), the pass registers storing the state variables shift to the right by one bit position. When a stripe column is decoded, the *significance pass* receives the state variables corresponding to the next column from the internal RAM and the state variables move from one pass register to another. At the output of the *cleanup pass* registers, the σ, χ and σ' updated variables are sent back to the internal RAM, and the ν and χ updated variables are sent to the output RAM. Although the χ variable is sent at each bit-plane to the output, it will only accurately correspond to the coefficient sign at the last bit-plane. The register sizes depend on the state variables. For the σ variable, the significance context calculation requires variables from three columns (coefficients 0 to 16 in Fig. 8). The same is true for the χ variable used in the sign context calculation and the σ' variable used in the refinement context calculation. The σ, σ' and χ registers are thus 17-bit wide. Since only the four ν and η variables from the current stripe are used by the passes, their registers are 4-bit wide. The σ, χ and σ' variables from the last line of the previous stripe (line containing coefficients 4, 10 and 16 in Fig. 8) are stored in a cyclic LUT register, loaded in the significance pass registers and updated by the cleanup pass. Thanks to the parallel mode, variables from the next stripe are not needed during the decoding. However, they are present in the registers to enable a generalization of the context calculation.

In order to improve the decoding performances, the context calculation in each pass is done in a pipeline way: the context of coefficient *i+1* is calculated when coefficient *i* is analyzed.

This register-based architecture offers a significant memory reduction because the η state variables do not need to be stored in the RAM after the *cleanup pass*, all the bits having already been decoded. This means that compared to [11], only 75% of the internal RAM is necessary. The architecture also reduces the number of RAM accesses, only one access being necessary for the decoding of 4 bits. The state machine of the *significance pass* is represented in Fig. 9.

The *Synchronization Block* has two functions. First, it synchronizes the *Pass Blocks* at the end of each column, allowing each to decode a column at its own rhythm. Second, it generates control signals indicating to the three pass blocks their current position in the code-block. The signals for the *significance pass* are generated by a single counter and are simply registered for the two other passes. These signals are used by the three finite-state machines and modify register values at the code-block borders. For

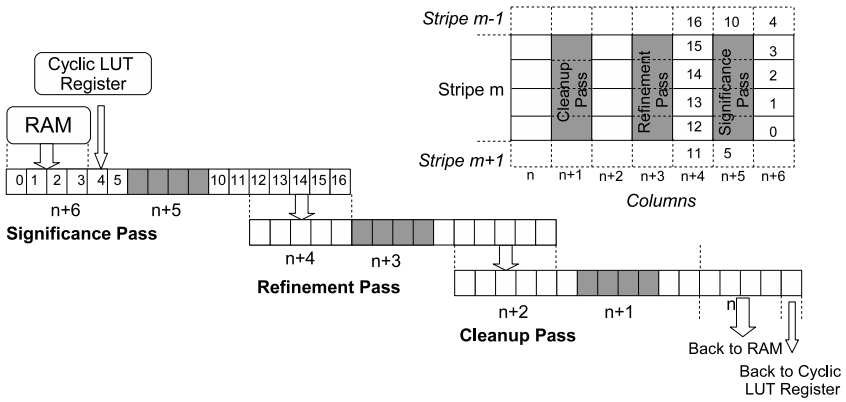

Fig. 8. Communication between the state variable σ registers of each pass. The other state variables registers operate in the same way. Grey-shaded boxes represents the bits currently handled by each pass. White descending arrows are synchronous operations made by the three passes at the end of each column. Right shifting operations inside each register are done asynchronously.

example, in the first stripe, the state variables of the upper neighbors (coefficients 4, 10 and 16 of Fig. 8) are not read from the circular register but are set to 0.

An *Output RAM* progressively receives the decoded code-block from the *cleanup pass*. Since the Entropy Decoding Unit (including the CMU and ADU blocks) is the slowest component of the decoder because of the EBCOT algorithm's complexity, it is essential to ensure that this unit can always process the input data and send the decoded code-blocks to the output. In order to achieve this, the *Output RAM* has been designed to store two code-blocks at a time. The EDU can thus begin to decode a new code-block while the previous one is still being processed by the IDWT.

Averaging on all the code-blocks of natural images, the CMU takes 2.1 clock-cycles[2] to output one bit. Table III presents the Virtex-II FPGA resources used after synthesis (XST tool) and implementation (ISE 6.01 tool).

The proposed CMU design contains various optimizations in comparison with [11], most of them due to an efficient use of the parallel mode. The register communication system between the three autonomous Pass Blocks, as well as the Synchronization Block single counter generating control signal for the three passes, allow significant resource savings. This CMU architecture is generic and decodes all sizes of code-blocks: from small 128 coefficient code-blocks with only one non-zero bit-plane to 2048 coefficient code-blocks with all non-zero bit-planes.

[2]This result does not take into account the ADU decoding time.

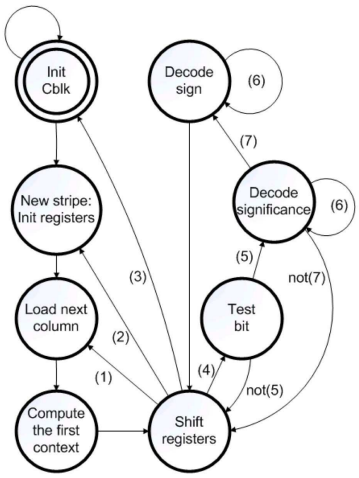

Fig. 9. Significance pass Finite State Machine. (1) End of column reached. (2) End of stripe reached. (3) End of code-block reached. (4) not(1,2,3). (5) Bit has to be decoded. (6) Wait for ADU's answer. (7) Bit decoded is significant.

TABLE III

SYNTHESIS RESULTS OF ONE CMU IN A XC2V6000-6.

Slices	489 over 33 792 (1.4%)
Look-Up Tables	729 over 67 584 (1.1%)
RAM blocks (18kbits)	1 over 144
Clock frequency	202.8 MHz

V. ARITHMETIC DECODING UNIT

A. MQ algorithm

The basic idea of a binary arithmetic coder is to find a rational number between 0 and 1 which represents the binary sequence to be encoded. This is done using successive subdivisions of the $[0;1]$ interval based on the symbols probability. Fig. 10 shows the conventions used for the MQ-coder.

C is the starting point of the current interval and also represents the current rational number used to encode the binary sequence. A is the size of the current interval. Q is the probability of the Least Probable Symbol (LPS) and is used to subdivide the current interval. According to the symbol to be encoded (MPS, i.e. Most Probable Symbol, or LPS), the following equations are respectively used:

$$A_{i+1} = A_i * (1 - Q) \quad \text{and} \quad C_{i+1} = C_i + A_i * Q$$

$$A_{i+1} = A_i * Q \quad \text{and} \quad C_{i+1} = C_i$$

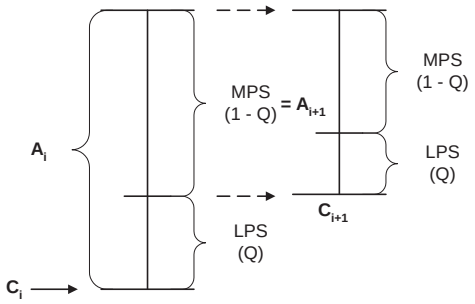

Fig. 10. Successive interval subdivisions in the MQ-coder (MPS encoding case). C is the starting point of the current interval and also represents the current rational number used to encode the binary sequence (0 when encoding the first symbol). A is the size of the current interval (1 when encoding the first symbol). Q is the probability of the Least Probable Symbol (LPS).

During the coding process, renormalization operations are performed in order to keep A close to unity. This leads to the following simplified equations, respectively for an MPS and an LPS:

$$A_{i+1} = A_i - Q_e \quad \text{and} \quad C_{i+1} = C_i + Q_e \tag{1}$$

$$A_{i+1} = Q_e \quad \text{and} \quad C_{i+1} = C_i \tag{2}$$

At each step, the Q_e-value (estimated Q-value) is retrieved from two serial look-up tables, using the context provided by the CMU.

Conversely, the decoding process consists in deciding to which interval (MPS or LPS) the rational number provided belongs, as well as progressively expanding the current interval until $[0; 1]$.

B. ADU architecture

As mentioned above, a drawback of the decoding process is a feedback loop from the ADU to the CMU. As a consequence, the CMU must wait for the ADU's answer before going on with the remaining bits to be decoded. In our architecture, priority has therefore been given to the highest ADU bit-rate, at the expense of a slight increase in resources. Thorough analysis of the MQ-algorithm [1] shows that only four steps are needed to decode one symbol:

1) *Load*: given a context, it retrieves the corresponding probability and the MPS-value.
2) *Compute*: during this step, the arithmetic operations are performed, needed to decide if an MPS

or an LPS must be decoded. They consist in only three *16 bits*-subtractions:

$$AQ_e = A_i - Q_e \quad (3)$$

$$CQ_e = C_i - Q_e \quad (4)$$

$$A2Q_e = A_i - 2 * Q_e \quad (5)$$

3) *Decide*: based on the results of equations (3) to (5) and on the *carry-out* bits of these operations, the A_{i+1} and C_{i+1} values, together with the decoded bit are deduced. The decoded bit is returned to the CMU, which can subsequently decide which context will be sent next. In some cases, the probability associated with the current context is updated[3].
4) *Renormalize*: as mentioned above, A has to be kept close to unity. Consequently, if $A_{i+1} < 0.75$, a renormalization (several left-shift operations) is performed until $A_{i+1} \geq 0.75$. The same amount of left-shift operations is done on C_{i+1}, to avoid corresponding bits of A and C having different weights. As C is actually the coded data, if the number of shifts to be done is greater than the number of bits left in the buffer, a new byte from the bit-stream is loaded from the memory.

Our ADU architecture is presented in Fig. 11 and detailed below. There are four main areas, corresponding to the four algorithm steps. Figure 12 presents the finite-state machine (FSM) used. Five different states (the four steps + an initialization state) control the registers and multiplexers.

The *load* area contains the RAM storing the probability estimate for each context. As each of the three ADUs located in an EDU (Fig. 5) is dedicated to one of the three passes, only part of the 19 contexts is stored in the RAM, depending on the ones used by each pass. This explains the variable width n of input CX ($n = 4$ for significance and cleanup passes, $n = 2$ for refinement pass). The $RamMQ$-block (detailed in Fig. 13) behaves in exactly the same way as a classic RAM-block with an entry for each context. The only difference lies in the fact that the value associated with each of these entries must be one of the 46 probability values. Those values are stored in a ROM inside the block. A speed-up technique is used to initialize the contexts to their initial values when a new codeword has to be decoded. Two RAM-blocks are used alternately: when one is used for decoding, the other is being initialized. In this way, the initialization process needs only one clock-cycle (to switch from one RAM to the other) instead of nc, where nc is the number of contexts. This is particularly useful in our architecture, as the parallel mode (see Section III-B) implies an initialization each time a new pass is decoded (rather than once per code-block with default encoding options).

[3]As explained in [1], this probability update occurs only if a renormalization is needed.

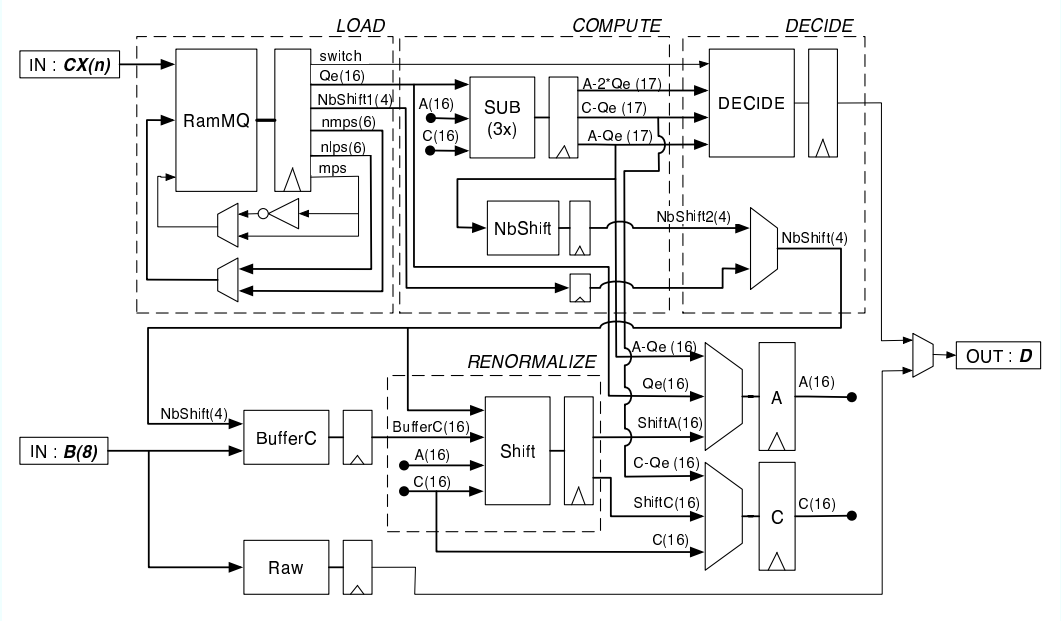

Fig. 11. The ADU architecture. Bus widths are indicated between brackets. There are four main areas, corresponding to the four algorithm steps : *load*, *compute*, *decide*, *renormalize*.

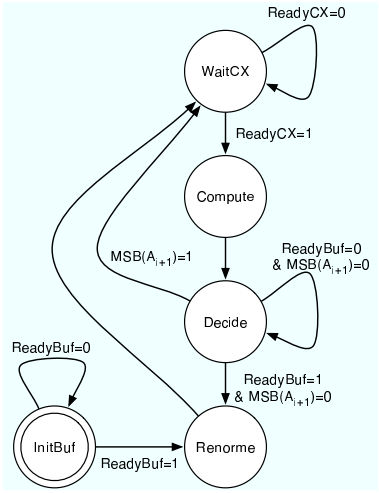

Fig. 12. The ADU finite-state machine.

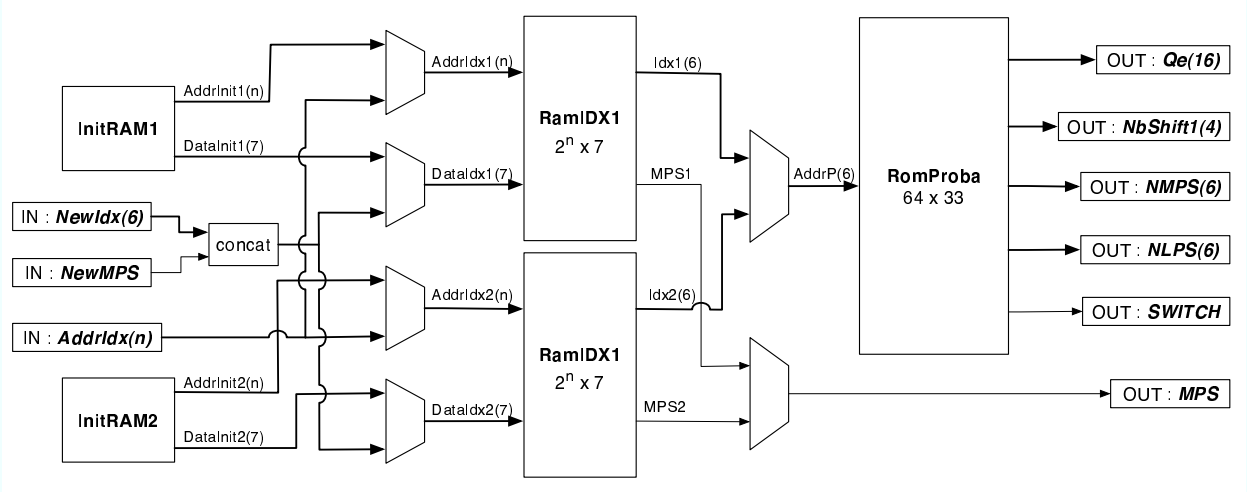

Fig. 13. The $RamMQ$-block. Bus widths are indicated between brackets. To speed-up initialization, two RAM-blocks are used alternately: when one is used for decoding, the other one is being initialized.

The *compute* area contains the three subtracters needed to perform (3) to (5) in one clock-cycle. Results are then used in the *decide* area. The number of left-shifts that may have to be done in the $renormalize$ step ($NbShift$) is computed as follows. Based on equations (1) and (2), there are two potential values for A_{i+1}. In the case of (1), analysis shows that $NbShift$ will only range from 0 to 2. In the case of (2), $NbShift$ ranges from 1 to 15 but depends only on Q_e and can be "hard-coded" in the $RamMQ$-block. As we want $NbShift$ to be ready to be used at the end of the decide step, both potential $NbShift$ values are generated, the first very easily computed in the $NbShift$-block and the second directly retrieved from the RAM. Then, a single multiplexer selects the correct value once A_{i+1} has been effectively chosen.

The $BufferC$-block interfaces the input-FIFO storing the bit-stream and the ADU itself. This process, independent from the main control part, guarantees that a minimum of 15 bits is always available at its output when a $renormalize$ step begins. This is indeed the maximum number of left-shifts that might have to be done and the renormalization can then be performed in one clock-cycle. It also undoes the bit-stuffing procedure that was performed during the encoding, to avoid a carry propagation.

The whole ADU architecture has been synthesized and implemented. Table IV presents the global resources used. No RAM-blocks were used, as all the memories inside the ADU are implemented with Look-Up Tables.

We summarize below some noteworthy characteristics, as they improve the architectures proposed in [12], [13].

TABLE IV

SYNTHESIS RESULTS OF THE ADU IN A XC2V6000-6.

Slices	498 over 33 792 (1.5%)
Look-Up Tables	944 over 67 584 (1.4%)
RAM blocks (18kbits)	0 over 144 (0%)
Clock frequency	140.4 MHz

- The main control state machine consists of only 5 states (against 28 in [12]). Furthermore, the CMU is waiting for the ADU answer during only three of them. Therefore a symbol may be decoded in 3 clock-cycles. The bypass mode improves this result even more.
- The compressed data loading is performed in an independent process, when the CMU is not waiting for an answer. As a consequence, the renormalization can be executed in one clock-cycle.
- To speed up the RAM initialization, which takes place each time a new codeword is sent to the ADU, two RAMs are used alternately.
- A speculative computation of the number of left-shifts is performed. Moreover, the most complex potential value does not need to be computed as it is pre-stored in the RAM.

VI. INVERSE DWT

A. DWT basics

In JPEG 2000, the DWT is implemented using a lifting-based scheme [6]. Compared to a classical implementation, it reduces the computational and memory cost, allowing in-place computation. The basic idea of this lifting-based scheme is to first perform a $lazy$ wavelet transform, which consists in splitting odd (d_i^0) and even (s_i^0) coefficients of the 1D-signal into two sequences. Then, successive *prediction* ($P_k(z)$) and *update* ($U_k(z)$) steps are applied on these sequences until wavelet coefficients are obtained. Fig. 14 illustrates the lifting step. The 2D-transform is simply performed by successively applying the 1D-transform in each direction.

In the architecture proposed, only the *Le Gall* (5,3) filter bank is implemented with an integer-to-integer wavelet transform. Quantification blocks (Q_{P1}, Q_{U1}, ...) perform smart rounding operations that ensure a reversible transform (that in turn allows for lossless capability). In (5,3) transformation, only one prediction step and one update step are needed to perform the whole 1D-transformation ($N = 1$) and the gain factors K_1 and K_2 equal to 1. Let $x(n)$ be the spatial coefficient sequence and $y(n)$ the

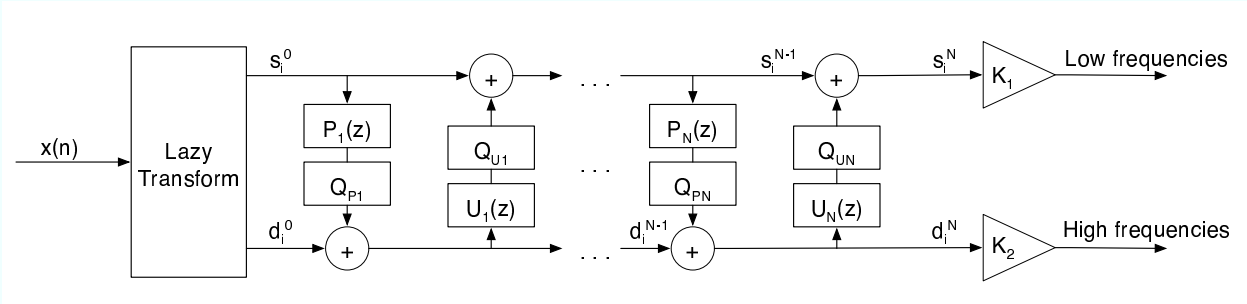

Fig. 14. The lifting-based Discrete Wavelet Transform.

wavelet coefficient sequence, the equations used to perform the transformation are:

$$\begin{aligned} y(2n+1) &= x(2n+1) - \left\lfloor \frac{x(2n)+x(2n+2)}{2} \right\rfloor \\ y(2n) &= x(2n) + \left\lfloor \frac{y(2n-1)+y(2n+1)+2}{4} \right\rfloor \end{aligned}$$

where $\frac{x(2n)+x(2n+2)}{2}$, $\frac{y(2n-1)+y(2n+1)+2}{4}$, and rounding operations respectively correspond to $P_1(z)$, $U_1(z)$ and (Q_{P1},Q_{U1}) blocks. Sequences $y(2n+1)$ and $y(2n)$ respectively correspond to high and low frequencies output in Fig. 14. The inverse transformation described below is simply obtained using the reverse system:

$$x(2n) = y(2n) - \left\lfloor \frac{y(2n-1)+y(2n+1)+2}{4} \right\rfloor \quad (6)$$

$$x(2n+1) = y(2n+1) + \left\lfloor \frac{x(2n)+x(2n+2)}{2} \right\rfloor \quad (7)$$

B. IDWT architecture

Our proposed solution is based on the combined 5-3 and 9-7 bank architecture detailed in [10]. We only investigated the 5-3 filter in order to implement an optimized and efficient IDWT level. In addition, we also developed a complete JPEG 2000 IDWT transformation with five recomposition levels.

To reconstruct one resolution level, an horizontal transformation is first applied, followed by a vertical one. The horizontal transformation architecture is shown in Fig. 15.

The first line is recomposed by taking the first line of the LL subband and the first line of the HL subband. LL (HL) wavelet coefficients are the even (odd) samples of the 5-3 transformation. The recomposition of the second line will use LH and HH samples as, respectively, even and odd values.

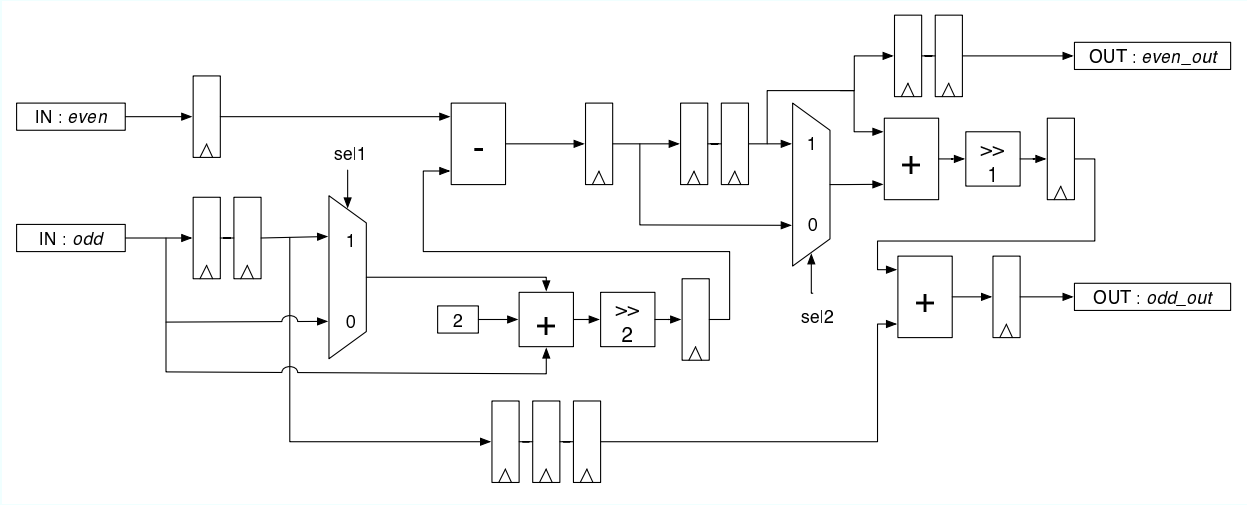

Fig. 15. Horizontal IDWT.

As can be observed, only four adders/subtracters, two shifters, two multiplexers and a few registers are needed to implement (6) and (7). In particular, no multiplier is used since all divisions are implemented with shifters.

This architecture is entirely pipelined: every new data couple "pushes" the data through the pipeline, toward the two outputs. Commands $sel1$ and $sel2$ deal with the edge effects and are computed using a small control part.

The vertical transformation is very similar to the horizontal one. The major difference is that neighbors are needed above and below each coefficient (instead of left and right neighbors). This implies buffering two entire lines of the reconstructed level. In Fig. 15, these buffers replace the two serial registers preceding each multiplexer. Fig. 16 details the resulting vertical IDWT architecture.

As two lines are simultaneously needed for the vertical filtering, a solution would be to use two parallel horizontal IDWT blocks, one for each line. However, those blocks would be used half the time, due to the single vertical filter. A better solution is to multiplex the operations of each line in a single horizontal entity. Therefore, the horizontal calculation process takes, cyclically, a (LL, HL) pair followed by a (LH, HH) pair. The complete 1-level IDWT is detailed in Fig. 17.

The whole IDWT architecture has already been presented in Fig. 5. In comparison with Chrysafis, who presented such an architecture in [9], various optimizations have been carried out. First, as mentioned above, the lifting scheme has been adopted for each level. Second, the blocks' interconnection has been

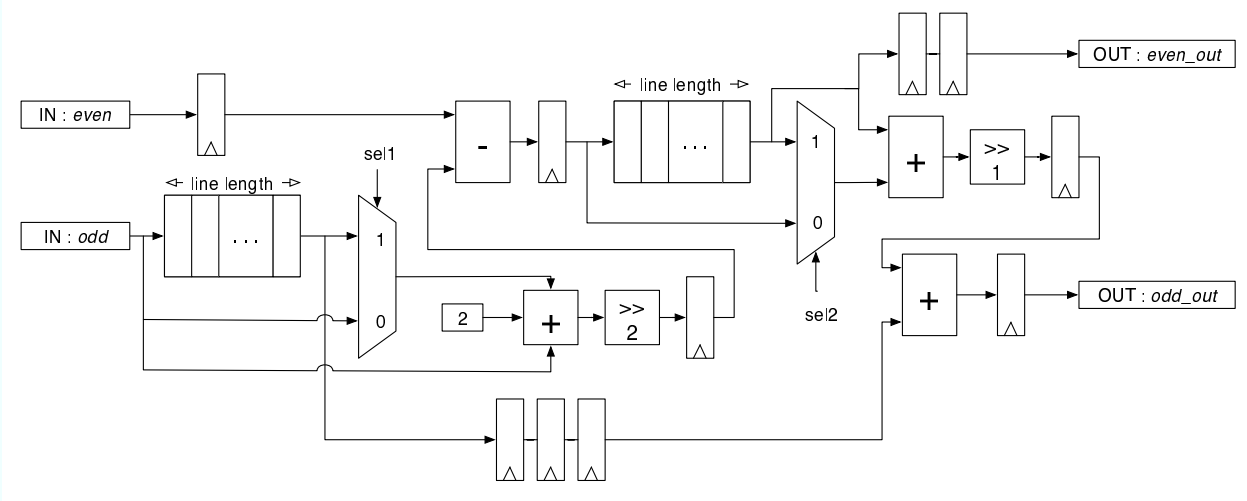

Fig. 16. Vertical IDWT.

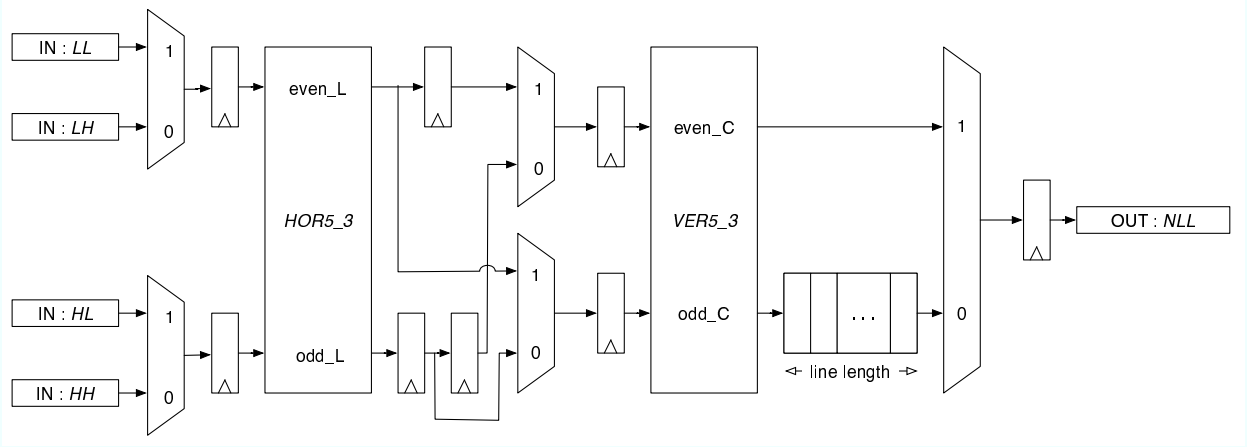

Fig. 17. Complete 1-level IDWT.

carefully studied and simplified. Each IDWTi-block ($i = 0..4$) reconstructs one resolution level and provides the next block with the reconstructed LL-coefficients exactly as a FIFO would. Therefore, the four inputs of each block behave in exactly the same way. Third, the pipeline characteristic, present in each level, has been extended to the whole architecture. Thanks to the progression order chosen, the sixteen FIFOs (one per subband) are filled as uniformly as possible. As soon as its four input FIFOs contain data (including the one simulated by the preceding level), an IDWTi-block begins reconstructing its level. When the pipeline is full, two coefficients of the reconstructed image are provided. Therefore,

TABLE V

SYNTHESIS RESULTS OF THE COMPLETE IDWT IN A XC2V6000-6

Slices	3 283 over 33 792 (9.7%)
Look-Up Tables	4 830 over 67 584 (7.1%)
RAM blocks (18kbits)	19 over 144 (13.2%)
Clock frequency	181.1 MHz

the IDWT0-block does not use a additional output buffer and produces two spatial coefficients from two consecutive lines at each clock-cycle. Finally, the scope for scalability is noteworthy as well. Addition or removal of IDWT levels is straightforward. Moreover, any intermediate resolution level is easily made available at the output. This is of high interest as this single architecture might therefore be connected to display devices with different resolutions.

We propose a complete IDWT able to produce $512 \times \infty$ tile images. For our tests, we have chosen an image height of 512 but it should be noted that there is virtually no height limit to the images compliant with the proposed architecture. As the latency of each IDWTi-block is the line length of the level being reconstructed, the whole pipeline latency is $(2^9 + 2^8 + 2^7 + 2^6 + 2^5) \simeq 2^{10}$. This latency is 128 times smaller than the $(512 * 512)/2 = 2^{17}$ clock-cycles needed to reconstruct an entire 512×512 image. This small latency enables a line-based image reconstruction, as only $\frac{1}{256}$ of the entire image needs to be buffered inside the IDWT architecture.

The complete IDWT with 5 different levels of recomposition has also been synthesized and implemented. Table V presents the global resources used. The nineteen RAM blocks correspond to the memories required to implement the sixteen different input FIFOs. Assuming 12 bit-planes for the wavelet coefficients, the five IDWTi blocks require a total of 3,696 bytes[4] of memory, which corresponds to 924 slices[5] using the shift register slice configuration.

VII. PERFORMANCES

The performance analysis of our architecture is based on three criteria: reconstructed image quality, resources consumption and achieved throughput. This analysis has been performed based on an architecture with 10 EDUs in parallel.

[4] $(2 \times 512 + 3 \times (256 + 128 + 64 + 32)) \times (12 \div 8)$ bytes.

[5] $(3,696 \times 8) \div (16 * 2)$ slices.

TABLE VI

SYNTHESIS RESULTS OF THE DECODING SCHEME IN A XILINX XC2V6000-6.

Slices	27 291 over 33 792 (80.8%)
Look-Up Tables	46 274 over 67 584 (68.5%)
RAM blocks (18kbits)	89 over 144 (61.8%)
CLK1 (EDUs & Dispatch)	116.9 MHz
CLK2 (IDWT)	181.1 MHz

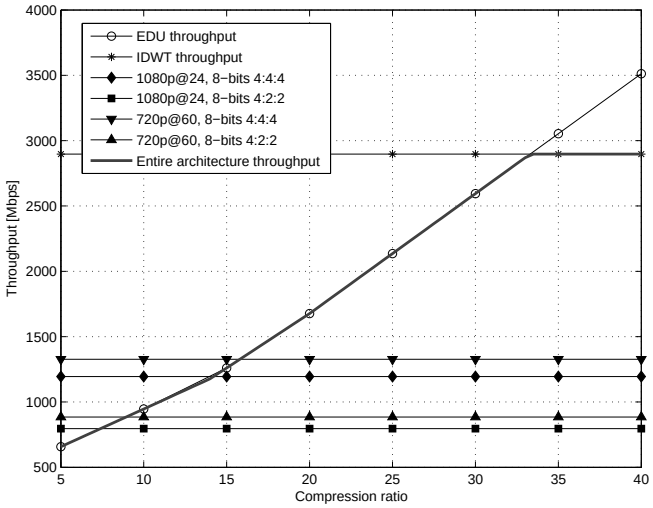

Fig. 18. Bit-rates achieved by the proposed architecture. Most common HDTV formats are reached for compression ratios between 7 and 16.

The first criterion (achieved quality) has already been studied in Section III-B. It has been shown that encoding options required by our decoding architecture give results visually identical to the ones with default encoding options.

Concerning resources and throughput, Table VI presents the synthesis and routing results in a Xilinx XC2V6000-6 FPGA. About 6 500 slices are still available in this configuration, which would be enough to add a decryption and watermarking modules, as described in Section III. Moreover, thanks to our line-based approach, only 61.8% of the RAM resources are used. Further development could then easily enable bigger tiles or code-blocks. The bit-rate achieved by our architecture highly depends on the compression ratio used at the encoding side. As it can be seen on Fig. 18, most common HDTV formats are reached

TABLE VII

COMPARISON OF 3 RECENT IMPLEMENTATIONS WITH THE PROPOSED ARCHITECTURE.

	Arizona Univ. [12]	**Analog Devices** [15]	**Barco Silex** [14]	**Proposed architecture**
Technology	ASIC 0.18μm	ASIC	FPGA XC2V3000-6	FPGA XC2V6000-6
Entropy coders	3	3	8	10
Wavelet filters used	(5,3)-lossless (9,7)-lossy	(5,3)-lossless (9,7)-lossy	(5,3)-lossless (9,7)-lossy	(5,3) lossy and lossless
Max. cblk size	32×32	$4\,096$ coeff.	32×32	$2\,048$ coeff.
Max. tile size	128×128	$2\,048 \times 4\,096$	128×128	$512 \times \infty$
Memory [Kbits]	310	not provided	$1\,332$	$1\,602$

for compression ratios between 7 and 16.

Table VII compares three other JPEG 2000 hardware implementations with our architecture. It shows that one of the main advantage of our solution is the memory usage. Compared to [12] and [14], tiles 4 times wider and *without height limitation* are managed while memory is only increased by a factor of respectively 5.2 and 1.2. The line-based approach proposed in this paper avoids any external memory usage, even with such bigger tiles, and enables a totally secure decoding scheme. It also reduces drastically the architecture latency.

Figure 19 presents a throughput comparison with the implementation from the same technology [14]. Throughputs estimations have been deduced from the FactSheet [14]. As it can be seen and given the significant difference in resource consumption between both architectures, the EDUs modules in the commercial implementation [14] perform better than ours. On the contrary, its IDWT module is far less efficient and limits quickly the throughput achieved. Beyond a compression ratio of 20, our architecture becomes better in terms of ratio (throughput/resources). Consequently, further research should focus on EDU resource reduction: a new architecture without any ADU duplication is already under development.

VIII. CONCLUSION

In this paper, we have proposed a hardware JPEG 2000 decoder for real-time applications such as Digital Cinema. It has been implemented in VHDL, and synthesized and routed in an FPGA.

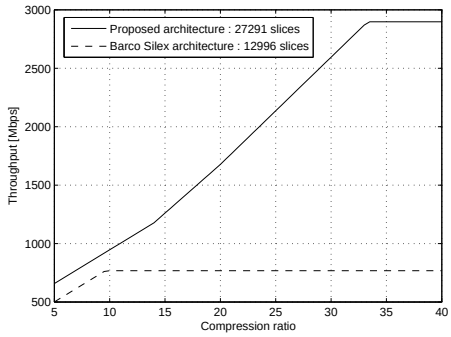

Fig. 19. Throughput comparison. The number of slices is given as an indication of resources consumption.

Various previous contributions have been joined together and optimized to provide a complete, secure, high performance and flexible decoding scheme.

The system proposed is secure because no external memory is used and the data flow is protected during the whole decoding process.

Thanks to three different levels of parallelization and line-based data processing, high output rates are achieved. With a compression ratio of 14 (7), the configuration synthesized in the FPGA supports the 1080/24p HDTV format for respectively 8-bit 4:4:4 images and 8-bit 4:2:2 images. In particular, the line-based data processing allows the inverse wavelet transform module to provide very high output rate with minimum memory and resources constraints.

Finally, the system proposed is highly flexible. In order to satisfy a broad range of constraints and avoid the problems inherent to multi-chips boards, two of the three parallelization levels are easily customizable. They allow the proposed architecture to fit in any single FPGA without further development. This underlines the interest of this kind of technology compared to ASICs, in fast-evolving markets such as video processing.

APPENDIX

ENCODING OPTIONS REQUIRED BY THE PROPOSED ARCHITECTURE

In order to generate images that our architecture is able to decode, following options must be specified during the encoding process:

- tile size: 512×512. The tile height is actually not limited to 512 and might be chosen bigger.
- code-block size: 8×256. This guarantees that each code-block spans the whole subband length.

- precinct size: 16×512 for all resolution levels except the smallest one which requires 8×512.
- switches: $reset$, $restart$, $causal$, $bypass$. The first three switches enable simultaneous pass processing (parallel mode)
- progression order: PCRL. This order groups together packets from the same spatial area.

The OpenJPEG library [22] was used to compress the test images, with the following command-line (example with $lena$ and a compression ratio of 16): "*image_to_j2k -i lena.pgm -o lena.j2k -r 16 -n 6 -t 512,512 -b 8,256 -c [16,512],[16,512],[16,512],[16,512],[16,512],[8,512] -p PCRL -M 15*".

REFERENCES

[1] M. Boliek, C. Christopoulos, and E. Majani, "15444-1: Information Technology-JPEG 2000 image coding system-Part 1: Core coding system," ISO/IEC JTC1/SC29 WG1, Tech. Rep., July 2001.

[2] D. Santa-Cruz, R. Grosbois, and T. Ebrahimi, "Jpeg 2000 performance evaluation and assessment," *Signal Processing: Image Communication*, vol. 17, no. 1, pp. 113–130, January 2002.

[3] S. Foessel, "Motion JPEG 2000 and Digital Cinema (N2999)," ISO/IEC JTC1/SC29 WG1, Tech. Rep., July 2003.

[4] T. Fukuhara and D. Singer, "15444-3: Information Technology-JPEG 2000 image coding system-Part 3: Motion JPEG 2000," ISO/IEC JTC1/SC29 WG1, Tech. Rep., July 2002.

[5] "Virtex-II platform FPGAs: Complete Data Sheet," Xilinx website, 2003. [Online]. Available: http://www.xilinx.com

[6] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *J. Fourier Anal. Applic.*, vol. 4, pp. 247–269, 1998.

[7] C. Diou, L. Torres, and M. Robert, "Wavelet ip core for data coding," in *Proc. Int. Workshop IP-Based Synthesis and SoC Design*, Grenoble, France, December 2000, pp. 97–102.

[8] G. Savaton, E. Casseau, and E. Martin, "High level design and synthesis of a discrete wavelet transform virtual component for image compression," in *Proc. Int. Workshop IP-Based Synthesis and SoC Design*, Grenoble, France, December 2000, pp. 103–108.

[9] C. Chrysafys and A. Ortega, "Line based, reduced memory, wavelet image compression," *IEEE Trans. on Image Processing*, vol. 9, no. 3, pp. 378–389, March 2000.

[10] G. Dillen, B. Georis, J.-D. Legat, and O. Cantineau, "Combined line-based architecture for the 5-3 and 9-7 wavelet transform of jpeg 2000," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 13, no. 9, pp. 944–950, September 2003.

[11] K. Andra, T. Acharya, and C. Chakrabarti, "Efficient VLSI implementation of bit plane coder of JPEG2000," in *Proc. SPIE Int. Conf. Applications of Digital Image Processing XXIV vol. 4472*, December 2001, pp. 246–257.

[12] ——, "A High-Performance JPEG2000 Architecture," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 13, no. 3, pp. 209–218, March 2003.

[13] C.-J. Lian, K.-F. Chen, H.-H. Chen, and L.-G. Chen, "Analysis and Architecture Design of Block-Coding Engine for EBCOT in JPEG 2000," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 13, no. 3, pp. 219–230, March 2003.

[14] "JPEG2000 Decoder: BA111JPEG2000D Factsheet," Barco-Silex, October 2005. [Online]. Available: http://www.barco.com

[15] "JPEG 2000 Video CODEC (ADV202)," Analog Devices, 2003. [Online]. Available: http://www.analog.com

[16] M. Rabbani and R. Joshi, "An overview of the jpeg 2000 still image compression standard," *Signal Processing: Image Communication*, vol. 17, no. 1, pp. 3–48, January 2002.

[17] D. Taubman, "High performance scalable image compression with ebcot," *IEEE Trans. on Image Processing*, vol. 9, no. 7, pp. 1158–1170, July 2000.

[18] J. L. Mitchell and W. B. Pennebaker, "Software implementations of the Q-coder," *IBM J. Res. Develop.*, vol. 32, no. 6, pp. 753–774, November 1988.

[19] D. Taubman and M. W. Marcellin, *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Boston, MA, USA: Kluwer Academic, 2002.

[20] G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat, "Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael Very Well Suited for Small Embedded Applications," in *International Conference on Information Technology (ITCC 2004), special session on embedded cryptographic hardware*, 2004.

[21] G. Rouvroy, F. Lefèbvre, F.-X. Standaert, B. Macq, J.-J. Quisquater, and J.-D. Legat, "Hardware Implementation of a Fingerprinting Algorithm Suited for Digital Cinema," in *Eusipco 2004*, September 2004.

[22] "OpenJPEG : an open-source JPEG 2000 codec," Communications and Remote Sensing Laboratory (TELE), UCL, Belgium. [Online]. Available: http://www.openjpeg.org

[23] D. Taubman, E. Ordentlich, M. Weinberger, and G. Seroussi, "Embedded block coding in jpeg 2000," *Signal Processing: Image Communication*, vol. 17, no. 1, pp. 49–72, January 2002.