View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositorio da Universidade da Coruña

J Sign Process Syst manuscript No.
(will be inserted by the editor)

High-speed FPGA Architecture for CABAC Decoding Acceleration in H.264/AVC Standard

Roberto R. Osorio · Javier D. Bruguera

Abstract Video encoding and decoding are computing intensive applications that require high performance processors or dedicated hardware. Video decoding offers a high parallel processing potential that may be exploited. However, a particular task challenges parallelization: entropy decoding. In H.264 and SVC video standards, this task is mainly carried out using arithmetic decoding, an strictly sequential algorithm that achieves results close to the entropy limit. By accelerating arithmetic decoding, the bottleneck is removed and parallel decoding is enabled. Many works have been published on accelerating pure binary encoding and decoding. However, little research has been done into how to integrate binary decoding with context managing and control without losing performance. In this work we propose a FPGA-based architecture that achieves real time decoding for high-definition video by sustaining a 1 bin per cycle throughput. This is accomplished by implementing fast bin decoding; a novel and area efficient context-managing mechanism; and optimized control scheduling.

Work supported in part by Ministry of Science and Innovation of Spain, co-funded by the FEDER funds of the European Union, under contract TIN2010-17541, and by the Xunta de Galicia, Program for Consolidation of Competitive Research Groups ref. 2010/6 and 2010/28.

Roberto R. Osorio
Dept. of Electronic and Systems,
University of A Coruña, Spain
E-mail: roberto.osorio@udc.es

Javier D. Bruguera
Centro de Investigacion en Tecnologias de la Informacion (CITIUS),
University of Santiago de Compostela, Spain
He is a member of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC).
E-mail: jd.bruguera@usc.es

Keywords H.264, CABAC, FPGA, arithmetic codes, video coding

1 Introduction

The new H.264/AVC [15] video coding standard represents a significant improvement in quality and compression ratio over previous standards such as MPEG-2 and MPEG-4 Part 2. This advantage comes with a price as H.264 is characterized by high computational complexity.

In H.264, the Context-based Adaptive Binary Arithmetic Coding (CABAC) algorithm offers a significant compression gain with respect to the baseline method ($\gtrsim$10%). However, this algorithm cannot be parallelized to a significant extent.

Sequential tasks pose a threat on performance as they limit the available parallelism due to Amdahl's law. This problem is addressed in [3] where some solutions are proposed, such as high clock frequency processors that are allocated to sequential tasks, or reconfigurable accelerators.

The need for hardware acceleration in arithmetic decoding (AD) has been addressed in [1] and [13] among others. Subsequently, the remaining decoding tasks are carried out in parallel using either reconfigurable hardware, or a number of programmable processors in a manycore system.

FPGA is a mature technology that allows accelerating computing tasks without coming into the high non-recurring costs and lack of flexibility of ASICs. Reconfigurability is a key factor, as the combination of programmable and reconfigurable parallel processing provides the adaptability required by modern multi-media platforms that must implement multiple standards.

Several architectures have been proposed for AD in recent years that apply different strategies. In [8, 12, 14, 19] the goal is to decode one bit of information (*bin*) per cycle. More recent papers acquiesce in higher complexity in order to decode several *bins* each cycle [2, 6, 9, 10, 18, 20, 21]. Actual throughput, however, varies widely. In this way, only [21] actually decodes more than 2 *bins* and only [2, 9, 10] sustain more than 1 *bin*. In many works, throughputs in the 0.25 - 0.75 range are obtained.

In general, just a few of the works found in the literature deal with the complexity of implementing context modeling and inverse binarization for several *bins* in the same cycle. In [2, 8, 18, 19, 21] this is addressed in more or less detail, but most of the best performing architectures [9, 10] do not even consider this subject. In many works, complexity is reduced by decoding multiple *bins* only under certain conditions. This leads to low and input-dependent throughput, as mentioned in the previous paragraph. Also, the convenience of multiple *bin* decoding is difficult to assess as none of those papers details how much the cycle length increases compared to a single *bin* architecture.

In this work, a new FPGA-based architecture is proposed for single *bin* arithmetic decoding. The strengths of this architecture are: a sustained throughput of 1 *bin* per cycle is achieved, independently of the video sequence, by eliminating stalls; context modeling is fully implemented; inverse binarization is performed for all the syntax-elements; the scheduling is carried out considering all the macroblock types; a novel scheme based on simple transpositions is used to evaluate all data dependencies; and the characteristics of FPGAs are exploited in order to achieve an efficient implementation.

In Section 2, the arithmetic coding and decoding algorithms are presented. Section 3 explains the decoding challenges at macro-block level. Section 4 describes the architecture, focusing on the high level control. Section 5 describes syntax-elements and *bin* decoding. Section 6 presents the results and compares with other architectures.

2 Arithmetic decoding in H.264

The Context-based Adaptive Binary Arithmetic Coding (CABAC) [11] is the preferred entropy coding algorithm in H.264 Main and High profiles. CABAC encodes the different pieces of information produced by the H.264 encoder, such as motion vectors, transform coefficients and flags, referred to as *syntax-elements*. The binary arithmetic coder in CABAC deals only with bit-size data, called *bins*. Therefore, all syntax-elements are converted to a sequence of *bins* before compression in a process called *binarization*.

bin	1	1	1	1	1	1	1	1	1	0	0	0	1	1
meaning	> 0	> 1	> 2	> 3	> 4	> 5	> 6	> 7	> 8	< 17	+0×4	+0×2	+1	neg
context offset	0,1, or 2	3	4	5	6	6	6	6	6	← exp-golomb codes no context used →				

Fig. 1 Decoding of value -10 as a motion vector

Hence, decoding consist of both recovering *bins* from the bitstream, and also reconstructing the original syntax-elements and take further decoding decisions. In this section, the encoding procedures will be explained first, and decoding details will be provided therafter.

Syntax-elements are binarized by applying some basic rules as described in [11]. As an example, Fig. 1 shows the breakdown of value *-10* as a motion vector coordinate, consisting of *unary* code of 9 bits, followed by an *Exp-Golomb* code of 4 bits, and the sign.

Each *bin* is compressed using arithmetic coding, exploiting the statistics of the source, that is, the proportion between the number of 0's and 1's. CABAC deals with the statistical properties of the *bins* using 2 variables: *mps* (most probable symbol) which signals with a single bit whether 0 or 1 has the highest probability at a given time; and *state*, which is a 6-bit measure of the skewness of the *mps* probability. Both values are updated using transition tables every time a *bin* is encoded. The combination of *mps* and *state* is called a *context*.

2.1 Contexts

In CABAC, more than 400 contexts are used to reflect the different statistics of the different types of *bins*. In the last row in Fig. 1 we see which context is used for each *bin* in the example (actually, an offset with respect to a base context is shown). Some *bins* share the same context as they have similar statistics, and some others do not have a context assigned. The latter ones are the *bypass bins*, for which 0 and 1 are equally probable. Another kind of *bins* with a fixed probability, *"final"*, also exist.

As it can be seen in Fig. 1, the first *bin* is encoded using 1 out of 3 possible contexts. This is done in CABAC because the statistics are dependent on the value of neighboring syntax-elements. This is further addressed in Section 3.

2.2 Encoding

The encoding process itself uses 2 variables: *low* and *range*. For every *bin* that is encoded, they are updated as follows:

$$\textbf{MPS} \quad low_{new} = low$$
$$range_{new} = range - rLPS \quad (1)$$
$$\textbf{LPS} \quad low_{new} = low + range - rLPS$$
$$range_{new} = rLPS \quad (2)$$

where *LPS* stands for *least probable symbol* and *rLPS* is the probability of *LPS* scaled to the current value of *range*. The *rLPS* is read from a look-up-table using *range* and *state*. The value of *range* is a 9-bit number that must be in [256, 511]. If it drops below 256, it is left-shifted (*normalized*) as many bits as needed to put it back in the right interval. The value of *low* is shifted in the same way as *range*. The bits shifted out of *low* form the resulting bitstream.

2.3 Decoding

CABAC decoding consists of reversing the encoding process shown above. The steps to follow are listed below:

1. Find out which syntax-element to decode
2. Determine which context to use for the next *bin*
3. Decode the *bin* using Eq. 3
4. Update the context
5. Update *low* and *range* similarly to Eq. 1 or 2
6. Decide if the syntax-element is fully decoded. If positive, go to 1)
7. If it is not decoded yet, go to 2)

When Eq. 3 produces a non negative result, an *MPS* is decoded. A negative outcome will result in a *LPS*. The actual value of the *bin* will be 0 or 1 depending on which one is the *mps* for the current context. Afterwards, Eq. 1 or 2 are used to update *low* and *range*. Finally, normalization may require taking bits from the bitstream and appending them to *low*, contrarily to the encoder.

$$low - range + rLPS \geq 0 \Leftrightarrow \textbf{MPS}$$
$$< 0 \Leftrightarrow \textbf{LPS} \quad (3)$$

3 Macro-block processing

The main coding unit in H.264 is the 16x16 pixel macroblock (MB). Decoding a MB produces a variable number of syntax-elements: motion vectors, transform coefficients and flags, depending on the MB type and its complexity. In this section we introduce the syntax-elements that make a MB and explain how contexts are chosen using information from neighboring MBs.

Table 1 Characteristics of MB types

	I MB	P MB	B MB
On slices...	I, P, B	P, B	B
Skip MB	none	skip	skip & direct
Partitions	4x4, PCM, 16x16	16x16, 8x16, 16x8, 8x8	16x16, 8x16, 16x8, 8x8
Sub-partitions	none	8x4, 4x8, 4x4	8x4, 4x8, 4x4
Intra-mode prediction	4x4 & 16x16	none	none
8x8 transform	only if 4x4 prediction	yes	yes
Motion vectors	none	only forward	forward & backward
Reference frames	none	one list	two lists

In H.264, there are 3 kinds of frames, slices and MBs: intra (I), forward (P) and bi-directionally (B) predicted. Table 1 summarizes their characteristics: the kind of slice in which they can appear, partitions sizes, prediction types, transform size and motion vectors.

Decoding a MB may require as few as one *bin* (the *skip* flag) and as many as hundreds or thousands. In general, an I-frame will require decoding the MB type; followed by 2, 5 or 17 prediction modes, up to several hundreds of flags; and a variable number of transform coefficients (up to 384). For a typical P-frame, the *skip* flag and MB type are decoded. Then, 4 partition types and reference frames, followed by up to 16 motion vectors. Again, the number of flags and transform coefficients may be up to several hundreds for the most complex MBs. For B-frames, the number of reference frames and motion vectors may be up to 8 and 32 respectively.

Motion vectors and transform coefficients (residual information) are the most important pieces of information in CABAC as they take most of the *bins* in the bitstream. The rules to decode them are complex and will be explained in latter sections. As an example, Fig. 2.(a), shows a variety of partition types (8x8, 8x4, 4x8 and 4x4) in the same MB for which 9 motion vectors must be decoded. Fig. 2.(b) tries to reflect the complexity of decoding non-zero transform coefficients and locating them within a MB. A more detailed description of the different syntax-elements may be found in [11].

Scheduling the decoding of the different syntax-elements poses some challenges as:

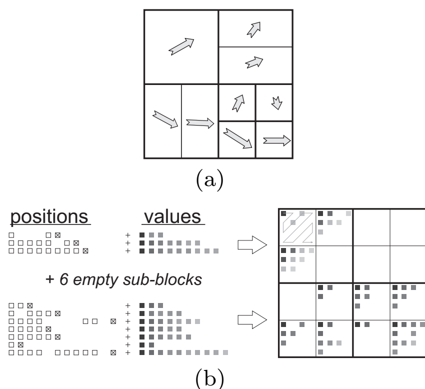


Fig. 2 Example of MB complexity. Variety of partitions (a) and location of the residual coefficients within a MB (b)

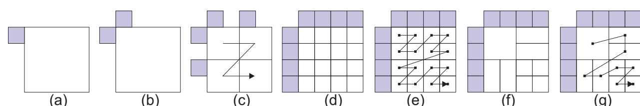


Fig. 3 Neighborhood relations for different syntax-elements

- Contexts are computed by analyzing neighboring MBs, blocks (8x8) and/or sub-blocks (4x4) depending on the types of partitions
- The value of a just-decoded *bin* or syntax-element is usually needed to decode the next one
- Most syntax-elements are made of a variable number of *bins*

3.1 Neighboring MBs

As said before, information from neighboring MBs is needed to decode the current one. Fig. 3 shows all the possible cases. Depending on the syntax-element, only the last processed MB may be needed (Fig. 3.(a)); or both the left and top MBs (Fig. 3.(b)). In other cases, 8x8 blocks are used (Fig. 3.(c)). Therefore, decoding those blocks depends on both the neighbor MBs and blocks from the current MB that have been already decoded. Finally, some syntax-elements are decoded at 4x4 sub-block level (Fig. 3.(d)). Moreover, irregular patterns are possible as shown in Fig. 3.(f). Figs. 3.(e) and 3.(g) show the scanning order in those cases.

The *bins* in a syntax-element are decoded using a fixed set of contexts, like a road map for each syntax-element. However, in most cases, the context to use for the first *bin*, and only the first *bin*, depends on neighboring values. Usually, the first *bin* encodes if the value is zero or not, so the context is chosen after checking if neighboring values are zero or not.

Table 2 Context selection

Syntax-element	Type	Equation	Total bits
skip	MB	$a+\alpha$	1
direct	MB	$a+\alpha$	1
intra mode	MB	$a+\alpha$	1
dqp	MB	a	1
chroma pred.	MB	$a+\alpha$	1
cbp chroma	MB	$a+2\alpha$	1
cbp chroma AC	MB	$a+2\alpha$	1
cb-flag chroma DC	MB	$a+2\alpha$	1
cb-flag luma DC	MB	$a+2\alpha$	1
cbp luma	block	$a+2\alpha$	2
cb-flag chroma AC	block	$a+2\alpha$	2x2
ref. MB	block	$a+2\alpha$	2x2
cb-flag luma (AC)	sub-block	$a+2\alpha$	4
dif. motion vectors	sub-block	*complex*	2x2x24

If we call a to the left neighbor and α to the top one, the context offset is obtained as either a, $a+\alpha$ or $a+2\alpha$ depending on the syntax-element. As an exception, finding the context offset to decode motion vectors follows a more complex procedure. Table 2 summarizes how context offsets are obtained for different syntax-elements and whether they are evaluated at MB, block or sub-block level.

4 High-level decoder

In previous sections we have highlighted the relevance of sequencing the different decoding operations. We propose a control structure in 2 levels. The high-level control decides which syntax-element has to be decoded at a given time and which initial context to use. The initial context is assessed by analyzing the value of the same syntax-elements in neighbor MBs, blocks or sub-blocks. The low-level control decodes syntax-elements proceeding *bin* by *bin*, using the information provided by the high-level control for the first *bin*, but deciding by itself which contexts to use for the remaining *bins*.

4.1 Implementation of the control in two levels

Control realization must consider the different challenges for each level. The high-level decoder deals with a moderate number of syntax-elements that are sequenced following complex rules. Also, contexts are assigned in different ways for each syntax-element. The low-level decoder, on the contrary, deals with a large number of *bins* and syntax-elements that, however, are decoded following a set of simple rules.

Therefore, we chose to realize high-level control using a cluster of Distributed Decoder Modules (DDM). Each of them is speciallized in dealing with a particular syntax-element or a set of related ones. At a given

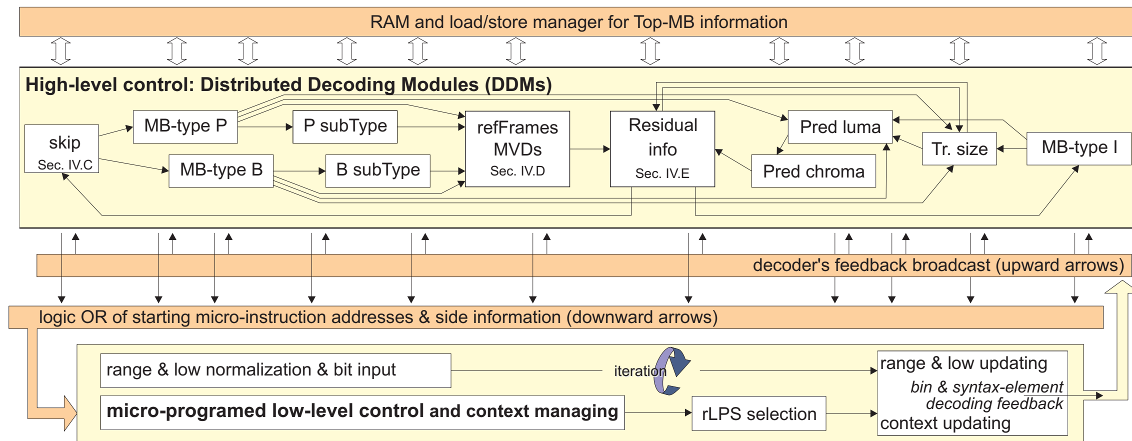


Fig. 4 Global architecture including neighbors storage, high-level control and low-level control for syntax-elements decoding

time, only one DDM will be active and interchanging information with the low-level control. After finishing its tasks, the active DDM will pass an activation signal to the appropriate DDM and switch itself into a wait state.

The low-level control is implemented using a micro-programmed approach that will be detailed in Section 5. The micro-program consists of a set of subroutines that are called by the DDMs in the high-level control. Each subroutine decodes a given syntax-element by executing a set of micro-instructions. Each of them decodes 1 *bin* and sequence the next micro-instruction.

This scheme is shown in Fig. 4. The high-level control is made of several DDMs. Some of them are seen in Sections 4.3, 4.4 and 4.5. Lets suppose that the *skip* DDM (on the left in the figure) is active. The DDM will issue an starting address to the low-level control (bottom of the figure). All the other DDMs are inactive and issuing zeros. The low-level control will execute a portion of the micro-program and report the decoded value to the high-level control. The *skip* DDM will get the decoded flag and pass the activation signal to another DDM (*MB-type P*, *MB-type B*, or *skip* again).

On the bottom part of Fig. 4, the low-level control and the decoding iteration are coarsely described. The iteration basically consists of updating *range* and *low* as seen in Section 2. The micro-programmed control contains the program memories and the logic for sequencing the instructions. A more detailed description will be given in Section 5.

4.2 Neighbors management

A novel and efficient management mechanism is proposed for accessing and updating neighbors information at MB, block and sub-block levels (Fig. 3). Contexts may be evaluated in a simple manner and just-decoded syntax-elements are readily accounted for future decoding operations.

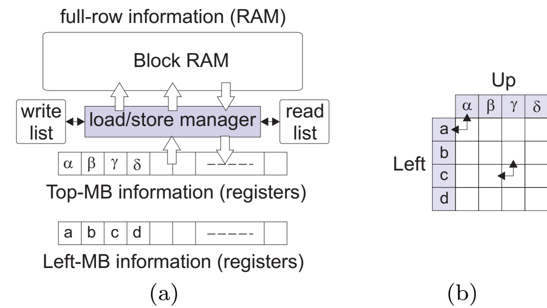


Fig. 5 Neighboring MBs information storage

When decoding a syntax-element, some information is needed from the MBs, blocks or sub-blocks on the left and on top of the current one in order to evaluate which contexts to use for the first bin.

The DDMs in the high-level control store that information in registers, and each DDM will prompt the low-level control to decode all the syntax-elements related with the information it stores. Indeed, the DDMs are built around the registers that store the neighbors information they must manage.

For the left neighbors, information is always kept in registers within the DDM. Managing this information is simple as, after decoding a syntax-element, this will become the left neighbor of the next one. For top neighbors, however, information is produced and consumed with a gap of a whole row of MBs. Therefore, that information is stored in RAM, and it is loaded in the DDM registers when needed. This scheme is shown in Fig. 5.(a) and also in Fig. 4.

The number of bits needed for each neighbor is summarized in Table 2. First, it depends on whether the

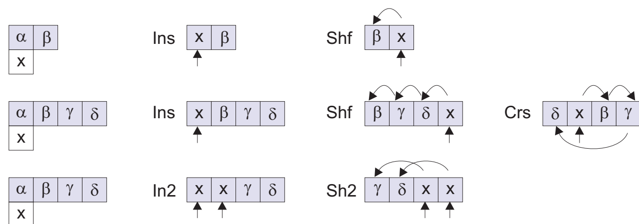


Fig. 6 Neighborhood relations for different syntax-elements

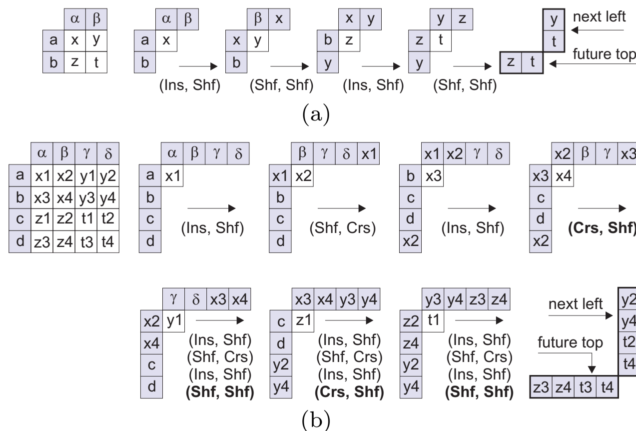


Fig. 7 Neighbor information managing for blocks (a) and sub-blocks (b)

syntax-element applies to a MB (1), block (2) or sub-block (4). Part of the chrominance (colour) related information applies to U and V components thus, a 2x factor is applied. For motion related information, up to 2 lists of reference frames are needed. Also, motion vectors, that will be further explained in this section, have 2 components (x and y).

The total amount of information is quite high, therefore, the load/store manager in Fig. 5.(a) deals with read and write lists in order to optimize memory accesses.

4.2.1 Neighbors addressing in the Distributed Decoder Modules

Whereas managing neighbors information at MB level is simple, some challenges arise when dealing with blocks and sub-blocks. This is illustrated in Fig. 5.(b), where the shaded boxes represent the *left* and *top* registers, while the white boxes represent the information being decoded for that MB. The first sub-block fetches information from the shaded boxes, but the following ones not. Addressing the required neighbors is not straightforward [5], especially when mixed block-sizes are used.

We propose an efficient implementation that only deals with the *left* and *top* registers, which content is updated by means of simple transpositions in such a way that the required information is always located in the same place. In Fig. 6 we show all the transpositions needed to manipulate the neighbors at block and sub-block levels. The way in which elementary transpositions are combined is now explained.

In Fig. 7.(a) the sequence of transpositions at block level is shown. The intent is to decode x, y, z and t given a, b, α and β. We actually only store the top row and the leftmost column. Only 2 transpositions are needed, **Ins**ert and **Shif**t. As it can be seen, the correct neighbors are always located in the top-left corner. Additionally, after the last transpositions, the row and column contain the information that will be needed for future MBs. The transposition pairs (Ins, Shf) and (Shf, Shf) must be interpreted as (*column*, *row*).

In Fig. 7.(b), the same procedure is shown for 4x4 sub-blocks. Here, **Ins**ert and **Shif**t are redefined to transpose 4 elements and a new transposition is introduced, **Cross**. In the figure, the 4 x_i values are decoded as shown. The other values: y_i, z_i and t_i are decoded in a similar way. The transpositions in bold font are used to change from x_4 to y_1, y_4 to z_1, z_4 to t_1 and t_4 to the final arrange.

Transpositions **In2** and **Sh2** are used when motion vectors are decoded for 8x8 blocks. All the transpositions can be combined, so that it is perfectly possible to decode motion vectors for 8x8 blocks mixed with 4x4 sub-blocks.

As only a small number of transpositions are needed, a simple implementation using just a few multiplexers, is possible. On top of solving the addressing issue, we avoid storing all the decoded values. Instead, only the ones that will be needed in the future are kept. The other ones are sent to be processed in other stage, away from the critical path, where they should be stored more efficiently in RAM. In this way, we achieve a 66% register saving withing the arithmetic decoder.

4.3 Simple syntax-element decoding

The DDM for decoding the *skip* flag is shown in Fig. 8. As just 1 *bin* is decoded, there are only 2 states: wait and active. When active, the FSM sends a decoding address (bold line) to the low level control. In the same cycle, the decoded *bin* is received (bold line), the FSM goes inactive and propagates the activity signal to the *MB-type P* or *B* FSMs, or starts with a new MB.

The decoding address differs for P and B frames. Also, and according with Table 2, the value of the left and top neighbors is added. This is a simple case of a flag at MB level. As said before, the left and top values

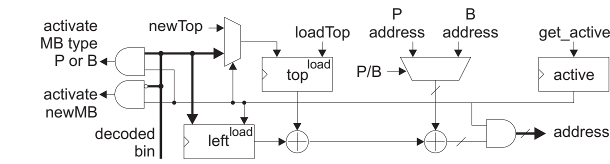


Fig. 8 Distributed Decoder Module for *skip* flag decoding

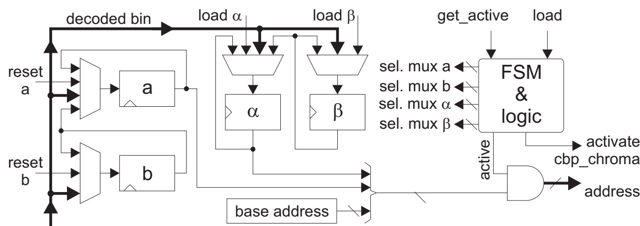


Fig. 9 Distributed Decoder Module for luminance *cbp* flags decoding

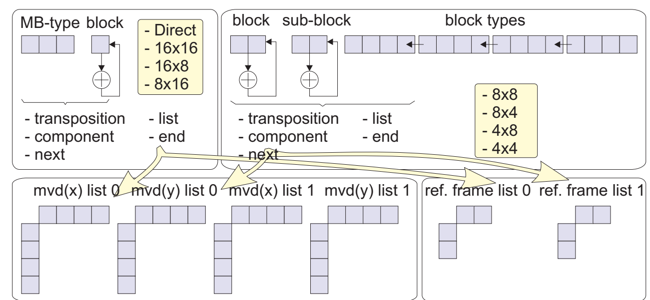


Fig. 10 Distributed Decoder Module for differential motion vectors decoding

are kept in registers, but the top value is loaded from the neighbors memory for every MB, and stored back there before proceeding with a new one.

Fig. 9 shows the slightly more complex case of the *luma coded-bit-pattern*, for which 4 *bins* must be decoded. The decoding address is calculated from a base address and the values in a and α. The muxes implement the transpositions **Ins**ert and **Shif**t, updating the content of the registers each cycle. After 4 cycles, the FSM goes inactive and passes the activation signal to the *cbp_chroma* FSM.

4.4 Motion information decoding

Motion vectors and reference frames are some of the most important syntax-elements. We focus on the most complex case: B-frames, that are predicted using both frames from the past (backward prediction) and from the future (forward prediction). Depending on the MB type, forward, backward or bidirectional prediction can be used. For each direction, one list of reference frames is kept. Thus, motion vectors and reference frames may belong to any of the 2 lists.

From the control implementation point of view, the following data structures are needed to store information from neighboring MBs and also blocks and sub-blocks within the current MB:

- 1 MB type
- 4 block types
- 4 reference frames for each list
- 8 differential motion vectors for each list

It must be said that the AD works with differential motion vectors. Actual motion vectors are obtained outside the AD following an elaborated prediction process.

Due to the complexity of this DDM, the explanation will be focussed on the data structures involved, instead of the logic required to transpose data and generate signals. Fig. 10 depicts the basic data structures in the DDM.

The top-left enclosure deals with all the MB partition sizes excepting 8x8 and smaller. A 1-bit counter iterates over the 2 partitions, if needed. The top-right enclosure deals with the small partition sizes. Two counters are needed to iterate over the 4 blocks and up to 4 sub-blocks. The MB and block types have been decoded previously by other DDMs and loaded into this one. When a block has been processed, the buffer that contains the block types is shifted and a new block is processed. This is the case depicted in Fig. 3.(f).

The part of the DDM represented by the 2 top enclosures issues control signals to the bottom enclosures indicating which transitions to apply and to which list and component (x or y). Internally, signals for switching to another partition and ending the process are generated.

The bottom part of Fig. 10 shows the data structures that store the reference frames and differential motion vectors. Each of them is a combination of registers (see Table 2 and Fig. 5.(a)) and logic, that applies the transpositions. For reference frames, contexts are easily computed. For differential motion vectors, however, the number of bits involved is larger and the rules to obtain the context are complex. The following equations show how the context offset is computed:

$$e(a, \alpha, comp) = |mvd(a, comp)| + |mvd(\alpha, comp)| \quad (4)$$

$$ctx_mvd(comp) = \begin{cases} 0, & e(a, \alpha, comp) < 3 \\ 1, & 3 \leqslant e(a, \alpha, comp) \leqslant 32 \\ 2, & e(a, \alpha, comp) > 32 \end{cases} \quad (5)$$

For neighbors a and α, the absolute values of vector component $comp$ are summed up obtaining $e(a, \alpha, comp)$. Depending on the magnitude of e, a different context is used, provided that finding a zero value surrounded

by small values is quite probable, but it becomes less probable as surrounding values increase.

Despite obtaining e is relatively complex, it does not introduce any data hazard, provided that components x and y of the motion vectors are decoded alternately. Hence, there is a least one whole cycle available to compute e for the next x component while y is decoded, and reversely.

Computing absolute values is not needed as the differential motion vectors are decoded in sign-magnitude format. As it can be seen, values above 32 do not make any difference. Therefore, all decoded values are saturated to 32. Each time a differential motion vector is decoded, its magnitude and sign are transmitted, but the DDM just keeps the magnitude saturated to 32. Hence, 6 bits are needed per component. In total, 96 bits must be stored for each neighbor. Despite this number is large, it actually represents a 66% saving with respect to a straightforward implementation.

4.5 Residual information

Residual information makes for a significant part of the *bins* in a MB. It basically, consists of the transformed and quantized values obtained in the encoder after motion compensation or intra-mode prediction. Four basic types of syntax-elements must be decoded:

- Coded Block Pattern (*cbp*), that identifies empty blocks
- Coded Block Flags (*cb-flag*), that pinpoint empty sub-blocks
- Significance map, that locate zero-valued coefficients
- Non-zero coefficients

In the example in Fig. 2.(b), the *cbp* is *1011* and the *cb-flag* is *1110 0111 1111*. The significance map is made of *zeros*, *non-zeros*, *last* and *no-last* flags following a zig-zag pattern. For the first sub-block it would be *nz-nl, z, z, z, nz-nl, nz-l*. Next, the non-zero coefficients are encoded/decoded similarly to the motion vector in Fig. 1. However, contexts are assigned in a quite complex way that depends on the relative position of the values.

In must be taken into account that, in H.264, up to 6 types of transform functions are used (Fig. 11). Decoding follows different rules depending on the transform type. The required data structures to book-keep *cbp* and *cb-flag* for all transform types are displayed in Fig. 12.(a).

Fig. 12.(b) summarizes the decoding process, where 3 different routes are shown depending on the MB type.

For the special case of 16x16 intra-prediction, the leftmost route is followed ($\mathcal{F},\mathcal{G}$), as *cbp* is not decoded

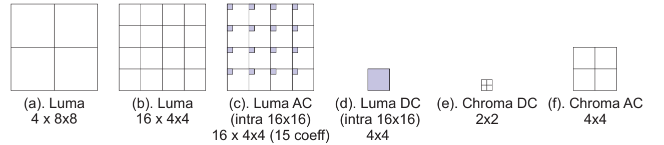


Fig. 11 Transform types

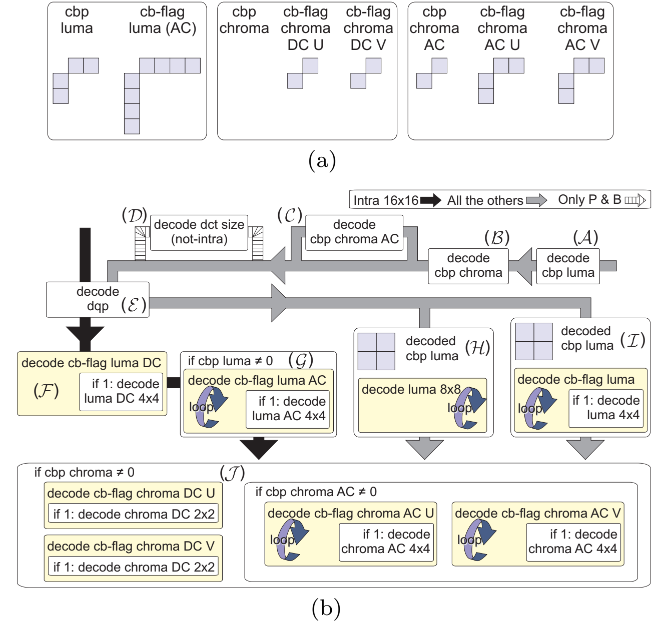


Fig. 12 Residual information decoding. Data structures (a) and Distributed Decoder Module (b)

and transform decoding is done according to Fig. 11.(c) and (d).

For all the other cases, the route starts on the top-right corner, where the 4 *cbp luma* bits ($\mathcal{A}$) are decoded first (Section 4.3). Each of them applies to a 8x8 block in Fig. 11.(a) and (b). For P and B MBs, the transform size ($\mathcal{D}$) must be decoded after *cbp chroma* ($\mathcal{B},\mathcal{C}$). If 8x8 transform is used, coefficients are directly decoded for not-empty blocks ($\mathcal{H}$). In other case, up to 16 *cb-flag* bits are decoded ($\mathcal{I}$), followed by transform coefficients for not-empty sub-blocks.

Decoding chrominance components ($\mathcal{J}$) is performed in the same way in all the cases and bears some differences with respect to luminance decoding: *cbp chroma* is made of just 2 bits (DC and AC) for both the U and V components; and DC values use a 2x2 transform size (Fig. 11.(e)).

The high-level control does not deal directly with the significance map and the transform coefficients. Instead, it decides which blocks and/or sub-blocks to decode based on the MB type, *cbp* and *cb-flags* while the low-level control (Section 5) iterates through those syntax-elements.

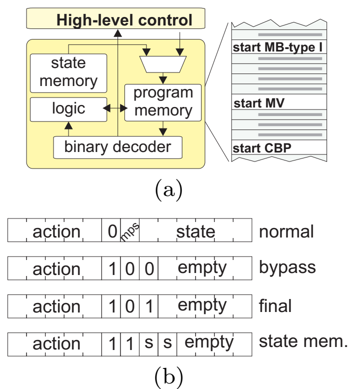


Fig. 13 Program-based control (a) and instruction formats (b)

5 Low-level decoding for syntax-elements

In this section, syntax-element decoding is explained, consisting of 2 parts: low-level control and arithmetic decoding.

5.1 Low-level control implementation

The low-level control follows a micro-programmed approach, in which there is one instruction for each *bin* to decode. One instruction is executed each cycle and, provided that no stalls happen between instructions or syntax-elements, a 1 *bin* per cycle throughput is achieved.

Fig. 4 shows the position of low-level decoding in the global architecture, and Fig. 13.(a) locates the micro-program within the lovel-level decoder.

The process of calling a subroutine in the micro-program from the high-level control was described in Section 4.1. In this section we will focus on: how context information is stored within the micro-instructions and the state memory; the format of the micro-instructions; and the special case of decoding residual information.

In order to ease the understanding, we will anticipate that a micro-instruction consists of 2 parts: context information; and the action to take after decoding.

5.1.1 Context coding

There are a number of works [4] [9] [21] [18] that address the challenging task of loading and updating context information. In a straight implementation, the context number is read first, then the context itself (*state* and *mps*), next *rLPS* and, eventually, the context is updated.

In order to reduce the number of memory accesses, we propose that the instructions do not contain a context number, but the *state* and *mps*. Therefore, loading the instruction brings also the context information. The value of *rLPS* is then read from distributed RAM in the FPGA. More aggressive optimizations can be found in [4, 17], but they incur in significantly higher cost.

Context updating consists, actually, in writing the last instruction back in memory with updated values of *state* and *mps*. As FPGA block-memories are mainly double ported, it is possible to load an instruction and write an old one at the same time. When the same instruction is executed in consecutive cycles, a simple buffer avoids using an outdated value.

5.1.2 State memory

As contexts are now attached to an instruction, there should not be 2 instructions using the same context. This poses a serious problem, as some contexts need to be reused (Fig. 1 gives us a good example).

Therefore, contexts are divided in 2 classes. In the first class, there are a vast majority of contexts (>85%) that are used by only one instruction and, therefore, follow the scheme explained above. In the second class, a few contexts are stored in a separated *state memory* which is governed by a *state logic*. Thus, some instructions do not contain the *state* of a context, but an escape code that asks a value to the *state logic*. The *state memory* is complemented with a one-element buffer that allows using the same context in 2 consecutive cycles.

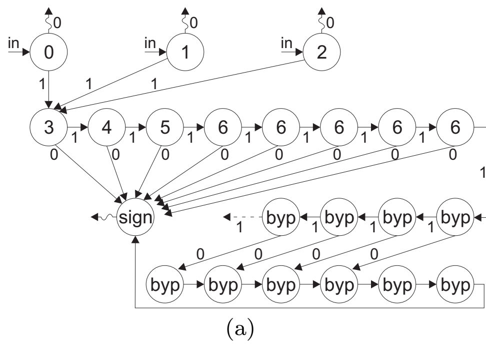
This scheme solves the problem stated above. An example of syntax-element that heavily requires the state FSM is the significance map for 8x8 transform size, for which contexts are shared by more than 4 coefficients on average.

5.1.3 Instructions and program format

The format of the instructions is shown in Fig. 13.(b). As it can be seen, *bypass* and *final bins* do not need to store *state* and *mps* as they are implicit. Those instructions that take the *state* from the *state memory* have also a hint (*ss*) for the *state logic*.

Some examples of actions are: *stop*; *stop if 0 or 1*; *continue*; *branch if 0 or 1*; or *decode while 1*.

Fig. 14.(a) shows the flowgraph for decoding a differential motion vector component as in Fig. 1. Up to 7 different contexts offsets (labeled 0 to 6) are used. Depending on the initial context offset, 3 entry points are possible (0, 1 or 2), and exit points are signaled with a curly arrow.



(a)

context	action	decodes
0	skip 2 / stop	is 0
1	skip 1 / stop	is 0
2	stop if 0	is 0
3	skip 3 if 0	> 1
4	skip 2 if 0	> 2
5	skip 1 if 0	> 3
6	skip 1 if 1	> 4
bypass	stop	sign
6	skip 3 if 0	> 5
6	skip 2 if 0	> 6
...	...	...

(b)

1	1	1	1	0	0	m	s	s	s	s	s	s
1	1	1	0	1	0	m	s	s	s	s	s	s
0	0	0	1	0	0	m	s	s	s	s	s	s
1	0	0	1	0	0	m	s	s	s	s	s	s
1	0	0	0	1	0	m	s	s	s	s	s	s
1	0	0	0	0	0	m	s	s	s	s	s	s
1	1	0	0	0	1	1	0	1				
0	0	0	0	1	1	0	0					
1	0	0	1	0	1	1	0	1				
1	0	0	0	1	1	1	0	1				
...					...							

(c)

Fig. 14 Motion vector decoding flowgraph and program

Fig. 14.(b) translates part of the flowgraph to the control program in a symbolic manner, whereas Fig. 14.(c) represents the same program in a binary way. We highlight that context 6 is used more than once, so it is stored in the *state memory*, not in the program. Bypass *bins* are a special case for which the state is implicit. This is actually one of many possible implementations of the program.

5.1.4 Residual information decoding

This is an special case that is implemented using additional hardware: 2 small counters that record the number of non-zero coefficients and trailing ones [11], and a small amount of logic. The significance map is decoded as a single syntax-element. Next, each non-zero coefficient is decoded. From the high-level control this is seen as a single syntax-element decoding.

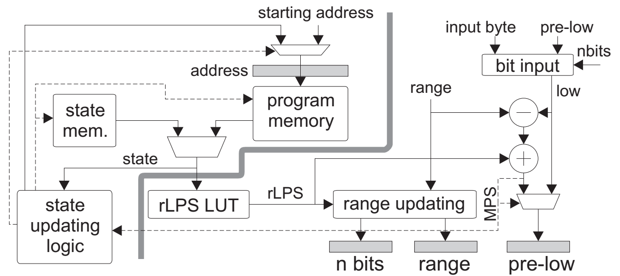


Fig. 15 Proposed architecture for binary decoding

5.2 Binary decoder architecture

The architecture can be divided in 2 parts: low-level control and context management; and pure binary decoding. The first part encompasses the program and state memories and the logic for sequencing the instructions and updating the contexts. The second part includes implementing Eq. 3 for decoding *MPS* or *LPS*, and updating and normalizing *range* and *low* according to Eq. 1 and 2.

Fig. 15 provides a detailed view, where both parts are separated by a divisory line. Processing starts by reading the program memory. The address comes either from the high-level control or the local logic. Context information is read from the just loaded micro-instruction or overridden by the state memory. Next, the *rLPS* is read from table implemented using distributed memory, and the state is updated.

For *range* and *low* updating, Eq. 1 and 2 are implemented in parallel, and one of the results is eventually selected. The cycle should end with the normalization of *range* and *low* and the input of new bits [12]. However, these tasks are delayed to the beginning of the next cycle in order to overlap with the memory access (top right in Fig. 15).

Our architecture is made of just 2 stages: high-level control and syntax-element decoding, as we have found out that further pipelining does not reduce cycle length significantly, while increases complexity and may introduce stalls. In this sense, using the same cycle for reading the context and decoding the *bins* is crucial. In other works, memory is accessed in a separated cycle and, despite speed is increased to some extent, pipeline hazards arise and stalls undermine performance.

Hence, our critical path is largely defined by accessing the program memory and fetching *rLPS* ($\approx 75\%$). However, as most of the operations overlap with memory access (those involving *rLPS* are the main exception), the cycle length is kept short.

Table 3 Area and speed figures

Clock	Slices	Memory	Registers	Power
100 MHz	646 (0.7%)	3 BRAM	~ 250	72 mW

6 Evaluation and comparison

Our proposal has been synthesized on a Xilinx Virtex-4 LX-200 FPGA [16]. The results are briefly summarized in Table 3. A 100 MHz clock frequency has been achieved. One block of RAM is used to store the micro-program, another one for the *state memory* and a third one for storing top neighbors information. A significant part of the area is devoted to the registers as described in Fig. 5. In our previous paper [12] this was not fully implemented and the figures were lower. It must be considered, however, that our neighbor management strategy allows for a 66% reduction in the number of registers. The dissipated power has been stimated using Xilinx XPower Analyzer. According to it, dynamic dissipation is 72 mW, whereas static (quiescent) dissipation is 1.5 W. Therefore, the consumption of our architecture is not significant within the context of a whole video decoding architecture.

As the clock frequency of this architecture is just moderate, the throughput per cycle is a crucial metric. As said before, decoding one *bin* always takes one cycle. Also, the low-level decoder is able to decode a whole syntax-element without stalls, and the high-level decoder does not require any waiting cycle between syntax-elements. Hence, a 1 *bin* per cycle, or 100 *Mbin* per second, throughput is achieved, which qualifies for HD Blu-Ray video decoding (100 Mbin/s $\geq$ 40 Mbit/s). In our previous work [12], stalls happened between a *skipped* or *direct* MB and the next one, but that problem has been solved in the present work.

In Table 4 we present a comparison with the most relevant works found in the literature. All of them decode multiple *bins* per cycle but, as explained in the introduction, the maximum throughput is seldom achieved. Columns 3, 4 and 5 show the performance using different metrics.

In column 6 (*Multiple bin*), the ways in which multiple *bins* can be decoded are shown. They range from brute force [10] to restricting to *bins* in the same syntax-element (SE) [2,9,18]. In [21] a different strategy is used, consisting of decoding up to 16 *MPS* per cycle.

Area figures are similar and they do not include memory. In [10] contexts can be accessed in parallel by implementing all the storage using registers. This is an expensive solution, that it is partially implemented in [9] (254 contexts in registers) and [21] (57 contexts). In [2] two overlapping RAM blocks are used, in [18] and [17] caches store, respectively, 9 and 44 contexts.

The last column shows how syntax-element decoding and context managing are addressed in each paper. The most complete descriptions are found in [2] and [21], whereas other works just focus on *bin* decoding. We extend our review on these works in the following paragraphs.

The main strength of [2] is the high throughput it achieves. This is mainly due to using 130 nm technology, as the throughput lags behind the targeted 2 bins/cycle. Context managing is correctly addressed but not at the same level as this paper.

The architecture in [21] allows HD decoding with low power dissipation. Despite precise figures are not provided in the paper, consumption must be extremely low at just 45 MHz. The performance is very similar to ours, but using a completely different platform and architecture. Mainly, context control is performed in the Probability Propagation modules in a hardwired manner as opposite to our programmed approach. The main advantages of our architecture are: it is tailored for FPGA implementation, and programmability makes easier to design new decoders for future video standards. Distinctively, [21] is better suited for ASIC implementation, it offers low power dissipation, and a high level of performance that may scale with technology.

Finally, despite of the advantages of using reconfigurable logic, few FPGA implementations of CABAC decoding have been found in the literature. In [17], a Stratix TM-based multiple *bin* architecture is presented. Both Virtex-4 and Stratix TM use 4-input LUTs thus, a fair comparison can be made. Up to 2 regular and a *bypass bin* can be decoded in the same cycle, but only when decoding the absolute value of transform coefficients. The actual throughput is, however, not disclosed. For storing *rLPS*, a 64x112-bit table is used, which partially explains the large area.

Performance comparisons between standard-cells and FPGA implementations are arguable. However, according to [7], ASICs should be, on average, 3.5 faster than FPGAs. Hence, as a conservative estimation, the performance of our architecture may be doubled, if needed, by targeting an ASIC implementation.

The most remarkable features of our work are: the targeted performance of 1 *bin* per cycle has been achieved by means of a balanced design and avoiding stalls. Syntax-elements decoding and context managing are efficiently implemented. Neighbors management is implemented using a solution based on a reduced number of registers and an efficient transposition scheme. FPGA characteristics are integrated in the design: double port

Table 4 Results comparison with other works

Work	Tech. (nm)	ThP (bin/cyle)	Freq. (MHz)	ThP (Mbin/s)	Multiple bin	Area (gates)	Ctx. memory architecture	SE decoding & ctx. control
Lin [10]	90	1.98	222	440	2 always	82-k	registers	no
Liao [9]	90	1.83	264	483	2 in SE	42-k	regs + RAM	no
Hong [6]	130	0.73-1.08	333	243-360	2 or 4 bypass	47-k	RAM + cache	no
Yu [20]	180	∼0.6	150	∼90	2 + byp.	$0.3mm^2$	RAM	partial
Yang [18]	180	0.86	140	120	2 in SE	35-k	RAM + cache	partial
Chen [2]	130	1.02-1.32	238	243-314	2 in SE	44-k	2 overl. RAM	yes
Zhang [21]	180	2.27	>45	>102	up to 16	42-k	regs + RAM	yes
Xu [17]	FPGA	n.a.	80.5	n.a	2 + byp.	7020 LUT	RAM + cache	partial
This	FPGA	1	100	100	single	1372 LUT	RAM	yes

RAMs in FPGAs are as fast as logic and we take advantage of putting memory access in the critical path.

7 Conclusions

In this work, a high-speed FPGA-based architecture for arithmetic decoding has been proposed that matches the requirements for HD real-time decoding. We prove that a pipeline with just 2 stages obtains high performance and works without stalls. To this end, the 2-level control is crucial.

We have introduced novel solutions for storing and accessing neighbors information; managing contexts and reducing the latency of *bin* decoding. Our neighbor management strategy allows for a 66% reduction in registers. Moreover, our architecture makes use of specific FPGA characteristics, such as fast dual-port block-RAM and distributed memory.

References

1. Azevedo, A., Juurlink, B., Meenderinck, C., Terechko, A., Hoogerbrugge, J., Alvarez, M., Ramirez, A., Valero, M.: A highly scalable parallel implementation of H.264. Trans. on High-Performance Embedded Architectures and Compilers (HiPEAC) **4**(2), 1–25 (2009)
2. Chen, J.W., Lin, Y.L.: A high-performance hardwired CABAC decoder for ultra-high resolution video. IEEE Trans. on Consumer Electronics **55**(3), 1614–1622 (2009)
3. D.A. Patterson et al: The landscape of parallel computing research: A view from Berkeley. Tech. rep. (2006)
4. Eeckhaut, H., Christiaens, M., Stroobandt, D., Nollet, V.: Optimizing the critical loop in the H.264/AVC CABAC decoder. In: FPT, pp. 113–118 (2006)
5. FFmpeg: `http://ffmpeg.mplayerhq.hu`. Last accessed: September 2012.
6. Hong, Y., Liu, P., Zhang, H., You, Z., Zhou, D., Goto, S.: A 360mbin/s CABAC decoder for H.264/AVC level 5.1 applications. In: Proc. ISOCC, pp. 71–74 (2009)
7. Kuon, I., Rose, J.: Measuring the gap between FPGAs and ASICs. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **26**(2), 203–215 (2007)
8. Li, L., Song, Y., Li, S., Ikenaga, T., Goto, S.: A hardware architecture of CABAC encoding and decoding with dynamic pipeline for H.264/AVC. J. Signal Process. Syst. **50**, 81–95 (2008)
9. Liao, Y.H., Li, G.L., Chang, T.S.: A high throughput vlsi design with hybrid memory architecture for H.264/AVC CABAC decoder. In: Proceedings of ISCAS, pp. 2007–2010 (2010)
10. Lin, P.C., Chuang, T.D., Chen, L.G.: A branch selection multi-symbol high throughput CABAC decoder architecture for H.264/AVC. In: Proceedings of ISCAS, pp. 365–368 (2009)
11. Marpe, D., Schwartz, H., Wiegand, T.: Context-based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard. IEEE Trans. CSVT **13**(7), 620–636 (2003)
12. Osorio, R., Bruguera, J.: An FPGA architecture for CABAC decoding in manycore systems. In: Proceedings of ASAP, pp. 293–298 (2008)
13. Själander, M., Terechko, A., Duranton, M.: A look-ahead task management unit for embedded multi-core architectures. In: Proceedings of DSD, pp. 149–157 (2008)
14. Son, W., Park, I.: Prediction-based real-time CABAC decoder for high definition H.264/AVC. In: ISCAS, pp. 33–36 (2008)
15. Wiegand, T., Sullivan, G.J., Bjøntegaard, G., Luthra, A.: Overview of the H.264/AVC video coding standard. IEEE Trans. CSVT **13**(7), 560–576 (2003)
16. Xilinx: Virtex 4. `http://www.xilinx.com/products/silicon-devices/fpga/index.htm`. Last accessed: September 2012.
17. Xu, M., Cheng, Y., Ran, F., Chen, Z.: Optimizing design and FPGA implementation for CABAC decoder. In: Proceedings of HDP, pp. 1–5 (2007)
18. Yang, Y.C., Guo, J.I.: High-throughput H.264/AVC High-Profile CABAC decoder for HDTV applications. IEEE Trans. CSVT **19**(9), 1395–1399 (2009)
19. Yi, Y., Park, I.: High speed H.264/AVC CABAC decoding. IEEE Trans. CSVT **17**(4), 490–494 (2007)
20. Yu, W., He, Y.: A high performance CABAC decoding architecture. IEEE Trans. on Consumer Electronics **51**(4), 1352–1359 (2005)
21. Zhang, P., Xie, D., Gao, W.: Variable-bin-rate CABAC engine for H.264/AVC high definition real-time decoding. IEEE Trans. Very Large Scale Integr. Syst. **17**, 417–426 (2009)

Roberto R. Osorio got his Ph.D. in Physics in 1999 at the University of Santiago de Compostela, Spain. In 2000, he joined IMEC v.z.w., where he contributed to MPEG-21. From 2003 he was a researcher within the Ramón y Cajal program, and associate professor from 2008. In 2010 he joined the University of A Coruña. His main research interests are in the area of application specific hardware for image and video coding.

Javier D. Bruguera received the graduated degree in Physics and the PhD degree from the University of Santiago de Compostela, Spain, in 1984 and 1989, respectively. Currently, he is a professor in the Department of Electronic and Computer Science at the University of Santiago de Compostela, Spain, and he is also with the "Centro de Investigación en Tecnologías de la Información" (CITIUS), University of Santiago de Compostela. Previously, he was an assistant professor at the University of Oviedo, Spain and at the University of A Corunna, Spain. Dr. Bruguera has been serving as Chair of the Department between 2006 and 2010. He was a research visitor in the Application Center of Microelectronics at Siemens, Munich, Germany, in 1993, for five months, and in the Department of Electrical Engineering and Computer Science at the University of California at Irvine, from October 1993 to December 1994.

His primary research interests are in the area of computer arithmetic, processor design, and computer architecture. He is author/coauthor of nearly 150 research papers on journals and conferences. Dr. Bruguera has served on program committees for several IEEE, ACM, and other meetings. Dr. Bruguera is member of the IEEE, the IEEE Computer Society and ACM.