Real-Time H.264/AVC Entropy Encoder Hardware Architecture in Baseline Profile

Ben Hamida Asma[1], Dhahri Salah[2]
[1,2] Laboratory of Electronic and Micro-Electronic (LAB-IT06),
Faculty of Sciences of Monastir,
University of Monastir, Tunisia

Zitouni Abdelkrim[3]
[3]College of Education in Jubail,
University of Dammam,
Saudi Arabia

***Abstract*—In this paper, we present a new hardware architecture of an entropy encoder for an H.264/AVC video encoder. The proposed design aims to employ a parallel module at a pre-encoding stage to reduce a critical path. Additionally, the arithmetic table elimination method is used to eliminate the memory cost. Besides, the reduction in the size of VLC tables offers area saving. This architecture is synthesized on an FPGA Virtex IV. The simulation results show that this design can operate up to 234 MHz, which allows processing a 4CIF video format in real time.**

***Keywords*—H.264/AVC; CAVLC; Exp-Golomb**

I. Introduction

The entropy encoder is the last part of an H.264/AVC encoder. H.264/AVC identifies two types of entropy coding methods, which are the Context-Based Adaptive Variable Length Coding (CAVLC) and the Context-Based Binary Arithmetic Coding (CABAC) [1]. In a baseline profile, only the CAVLC is utilized as an entropy coder mode with Exponential-Golomb (Exp-Golomb) codes. The CAVLC produces coding with higher efficiency than the conventional VLC coding. However, the CAVLC adds a high computational complexity due to context-adaptive characteristics.

Some work has presented the VLSI architecture of the CAVLC encoder to improve the performance of the entropy encoder. However, most work has focused only on how to increase the throughput of the CAVLC encoder. For instance, the pipelining architecture is usually used [2, 3, 4]. The work in [2] proposed a two-stage pipeline architecture. This method could reduce the time needed to process a block until reaching half of the mean time but it involved double memory size to store all syntax element information. In [3], the parallel coding of level and run-before sub-module encoders was applied. Moreover, the authors in [5] tried to increase the throughput by scanning the coefficient in parallel. However, it clearly doubled the area cost.

To reduce this area cost, [5] put forward optimized coefficient token (coeff-token) VLC Look-Up Tables (LUTs) into 9-bit words instead of storing 16-bit words. An arithmetic manipulation of encoding levels was exploited in [6] to eliminate some of the large size of conventional VLC LUTs.

On the other hand, some work has concentrated on designing a low-power CAVLC encoder. For instance, the authors in [7] used the side information-aided and symbol look-ahead techniques to minimize memory access.

This paper presents full hardware architecture of entropy coding, which contains Exp-Golomb and CAVLC encoders for an H.264/AVC baseline profile. To improve the timing performance, parallel coding modules are introduced at the pre-coding stage. To decrease the cost memory, an arithmetic table elimination technique is exploited to encode level and run-before sub-module encoders instead of using conventional VLC LUTs. Furthermore, the optimized coeff-token VLC and total-zero LUTs are applied to reduce the memory size as well.

This paper is organized as follows. Section 2 introduces both CAVLC and Exp-Golomb entropy encoding algorithms. The proposed architecture designs of the CAVLC and the Exp-Golomb are illustrated respectively in sections 3 and 4. Finally, the conclusion is drawn in section 5.

II. Entropy Coding Algorithm in H.264

In the baseline profile, H.264 uses two tools for entropy coding: the CAVLC coding and the Exp-Golomb one, as presented in Fig.1. The residual information (quantized coefficients) is coded using the CAVLC, while the other data are coded utilizing the Exp-Golomb.

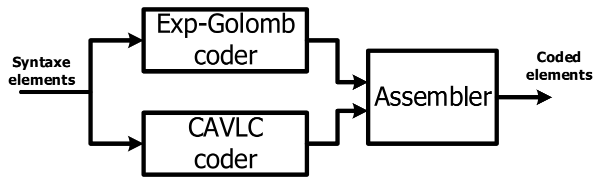


Fig. 1. Block diagram of entropy coder in baseline profile

A. *CAVLC algorithm*

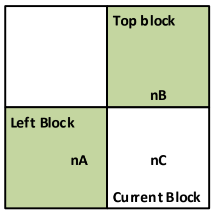
The CAVLC is the entropy encoding used to encode the residual information in 4x4 or 2x2 blocks, which are generated by the quantification step [1]. Each block must be firstly scanned in a zigzag order to produce five main syntax elements. The latter were defined in [1] as:

- The coeff-token represents two values : the total number of non-zero coefficients (total-coeff) and the number of trailing ones (TT1s) in the block. The trailing ones (T1s) are non-zero coefficients whose values are+/- 1 at the end of the zigzag sequence. Each block has at most three T1s.

- The signs of T1s are the coefficients with absolute value equal to one from zero to three bits wide. They represent the signs of the T1s coefficients in the reverse order.
- The levels are the values of each non-zero coefficient in the block, other than the T1s case. They are taken in the reverse order.
- The total-zeros is the total number of zero coefficients before the last non-zero coefficient in the zigzag sequence.
- The run-before represents the runs of zeros before each non-zero coefficient in the reverse order.

After that, these syntax elements will be encoded into five sequentially coding steps. The coeff-token, run-before and total-zero steps are encoded through different VLC LUTs. The CAVLC encoder steps are depicted in Fig.3.

- In step 1, the coeff-token are encoded using four VLC LUTs, based on the number of the total coefficients in the left block (nA) and the upper block (nB) of the current block (the context-adaptive notion), as shown in Fig.2.
- In step 2, each T1s is encoded with its corresponding bit sign in a reverse order. The positive sign is represented by '0', and the negative sign is represented by '1'.
- In step 3, the level values of the 4x4 block are encoded in a reverse order using seven VLC LUTs selected by the total-coeff and TT1s. The choice of the VLC LUTs to encode each level depends on the magnitude of the last encoded level (the context-adaptive notion).
- In step 4, 15 VLC LUTs are utilized to encode the total zeros, indexed by the total-coeff value.
- In step 5, the run-before is coded with codewords taken from seven VLC LUTs selected by zero-left values, which is the total number of the remaining zero coefficients.



(a)

```
If (nA = nB =0) then
nC =0;
Else
nC =(nA +nB)/2;
End if;
```

(b)

Fig. 2. Context-adaptive notion at coeff-token coding step (a) Data dependence (b) Correspond pseudo-code

B. *Exp-Golomb algorithm*

The Exp-Golomb coding is performed on two stages as provided in Fig.3.

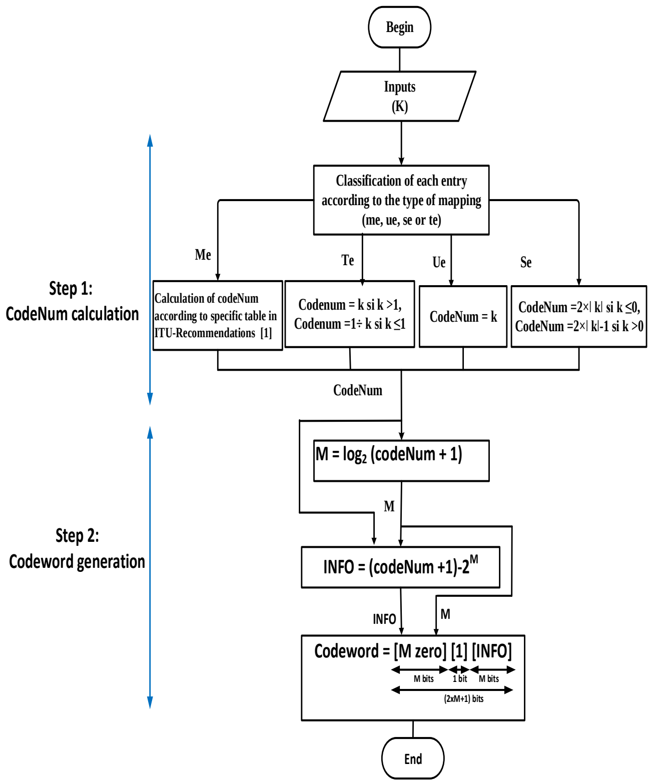

Begin
Inputs (K)
Classification of each entry according to the type of mapping (me, ue, se or te)
Me
Te
Ue
Se
Step 1: CodeNum calculation
Calculation of codeNum according to specific table in ITU-Recommendations [1]
Codenum = k si k >1, Codenum =1÷ k si k ≤1
CodeNum = k
CodeNum =2×| k| si k ≤0, CodeNum =2×| k|-1 si k >0
CodeNum
Step 2: Codeword generation
M = log2 (codeNum + 1)
M
INFO = (codeNum +1)-2^M
INFO
M
Codeword = [M zero] [1] [INFO]
M bits
1 bit
M bits
(2xM+1) bits
End

Fig. 3. Diagram of Exp-Golomb algorithm

- Firstly, each syntax element to be coded with the Exp-Golomb noted k is mapped to a non-negative integer named "codeNum." Based on the statistical characteristic, each syntax element is represented by a codeNum in various ways [1].
- If a syntax element is always larger than zero or equal to zero and if the most frequently occurring values are the lower ones, the applied process will be called "unsigned Exp-Golomb (ue) coding". The value of the corresponding codeNum is the same value of the unsigned element.
- If a syntax element is signed and the expectation value is zero, the applied process will be named "signed Exp-Golomb (se) coding". The value of the corresponding codeNum is mapped to the syntax element value k as follows:
    - ➢ CodeNum = 2|k| when ($k \leq 0$)
    - ➢ CodeNum = 2|k| - 1 when ($k > 0$)
- If an unsigned element has different statistical characteristics from the ue, its corresponding codeNum is then mapped to its value in a special way, as indicated in ITU-T recommendations [1]. The applied process is called "mapped Exp-Golomb (me) coding."
- If an unsigned element has 1 as the largest possible value, then "the truncated Exp-Golomb (te) coding" will be applied ; i.e., the bit representing the syntax element is the inverted value of the element.

Secondly, the codeNum parameter is mapped to coded string bits. The latter has the following generic form:

$$\{\text{M-zeros, 1, M-bit INFO}\} \quad (1)$$

where M and INFO are given by equations 2 and 3.

$$M = \text{floor}(\log2\ [\text{codeNum} + 1]) \quad (2)$$

$$\text{INFO} = \text{codeNum} + 1 - 2M \quad (3)$$

III. Proposed CAVLC Architecture

The suggested design processes each 4x4 block through two sequential stages. The pre-coding stage produces the Syntax Elements (SEs) to be encoded from the residual input frames, and the encoding stage translates each SE into a related codeword length and codeword value. In the following subsections, both stages are described.

A. *Pre-encoding CAVLC stage architecture*

The pre-encoding architecture is depicted in Fig.4. It has five main modules and four Random Access Memories (RAMs). The main modules are depicted in the figure below:

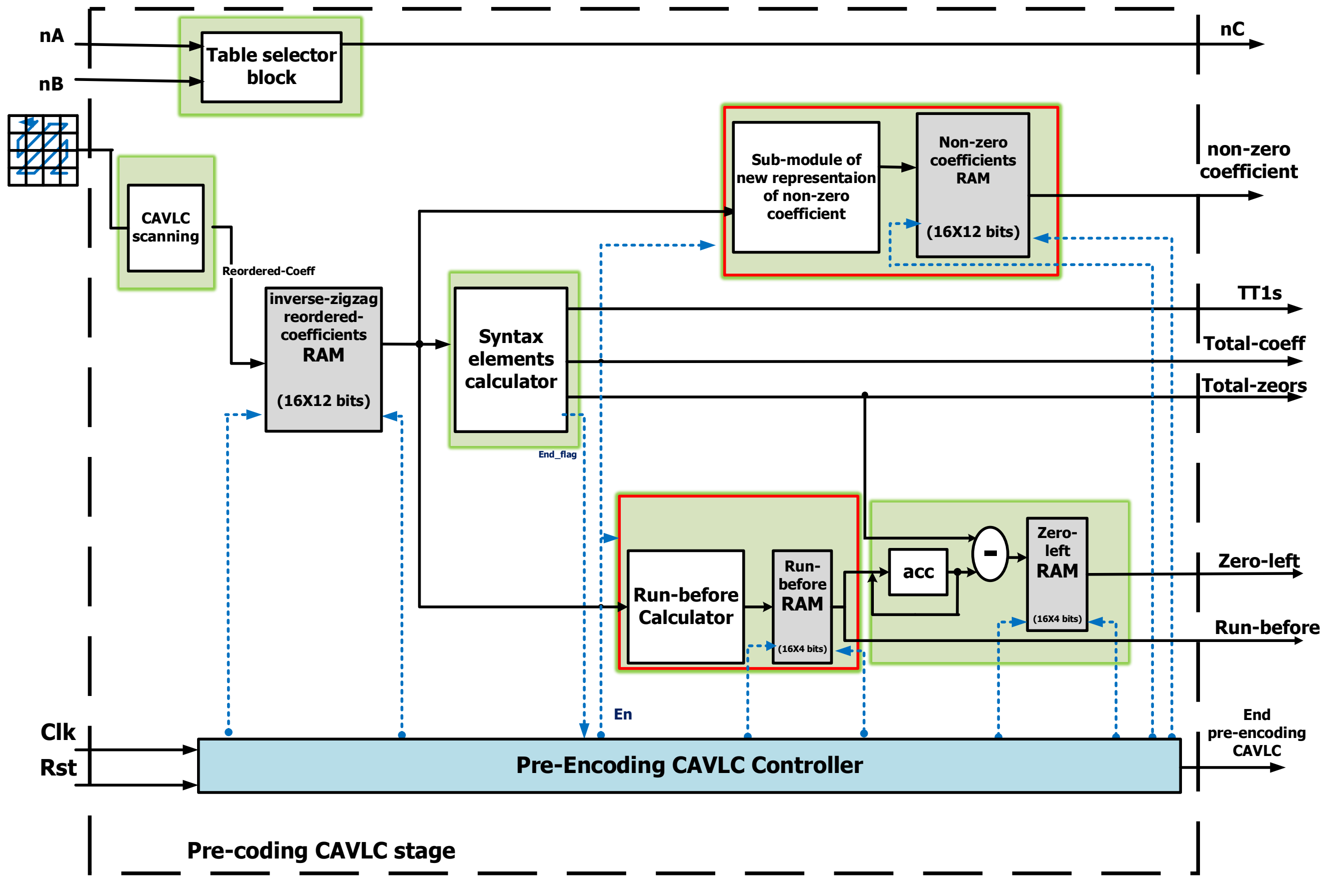
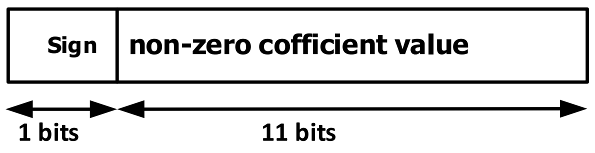

nA
nB
Table selector block
nC
CAVLC scanning
Reordered-Coeff
inverse-zigzag reordered-coefficients RAM (16X12 bits)
Syntax elements calculator
End_flag
Sub-module of new representaion of non-zero coefficient
Non-zero coefficients RAM (16X12 bits)
non-zero coefficient
TT1s
Total-coeff
Total-zeors
Run-before Calculator
Run-before RAM (16X4 bits)
acc
-
Zero-left RAM (16X4 bits)
Zero-left
Run-before
En
Clk
Rst
Pre-Encoding CAVLC Controller
End pre-encoding CAVLC
Pre-coding CAVLC stage

Fig. 4. Pre-encoding CAVLC architecture

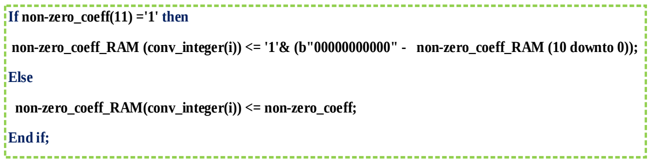
The zigzag module is responsible for ordering in an inverse zigzag order the residual information coming from the quantification process. After that, the zigzagged reordered coefficient is stored in a first memory called “inverse-zigzag reordered-coefficient RAM.” This module is not included in the CAVLC modules, but it is required for its correct operation.

The generator module of syntax elements has as an input the reordered coefficients. This module generates the first syntax elements to be produced, which are the TT1s, the total-coeff, and the total-zeros. When the values of these syntax elements are calculated, the next two modules, shown in red squares, start to be processed. Both modules are independent. Consequently, they are processed in parallel.

The parallel module on the top is responsible for storing the T1s and the level values into a “non-zero coefficient RAM” memory. The total number of levels and TT1s represent all total non-zero coefficients. Each non-zero coefficient is saved with a new format that represents the absolute value of the non-zero coefficient in 11 bits and the sign bit in the 11th bit, as illustrated in Fig.5. This format allows simplifying the level encoding process.



(a)



```
If non-zero_coeff(11) ='1' then
 non-zero_coeff_RAM (conv_integer(i)) <= '1'& (b"00000000000" -   non-zero_coeff_RAM (10 downto 0));
Else
 non-zero_coeff_RAM(conv_integer(i)) <= non-zero_coeff;
End if;
```

(b)

Fig. 5. (a) New representation of non-zero coefficient and (b) its correspondent pseudo-code

The second parallel module is formed by combinatorial circuits and two RAMs needed for storing each run-before and zero-left syntax element, respectively. First, this module permits calculating the different run-before values. After that, each calculated run-before value will be put into the "Run-before RAM" memory. When all the run-before values are detected and stored, the controller enables the process of the next module. This latter calculates the set of zero-left values and stores them into a "Zero-left RAM "memory. The zero-left value is initially equal to the total-zeros, and then this value is decremented with the accumulation of run-before values. The mathematical relationship between the zero-left and the run-before is shown below.

$$\text{Zero-left (i)} = \text{Total-zeros} - \sum \text{Run-before} \qquad (4)$$

It is worth noting that the size of all used memory is 16 elements, which is the maximum number of non-zero run-before and zero-left coefficients per 4x4block. Besides, the use of the inverse-zigzag reordered-coefficient, Run-before and Zero-left RAM memories is required for bitstream correctness.

The nC is also generated at this stage by a combinatorial circuit shown in Fig. 6. It selects the appropriate VLC LUTs for coeff-token coding.

The controller at the pre-encoding stage is in charge of defining the control unit of the different RAMs and synchronizing the various modules. When the end-pre-encoding signal is set active, all the syntax elements will be ready to be encoded.

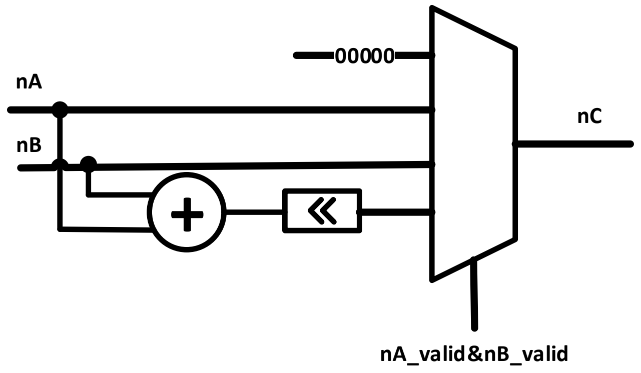


Fig. 6. Table selector architecture

B. *Encoding CAVLC stage architecture*

The encoding CAVLC architecture is illustrated in Fig.7. The CAVLC hardware design has the outputs of the CAVLC pre-encoder design as inputs. It is composed of seven main modules: five modules in charge of encoding the different syntax elements, one module for the main controller, and another one for the output packet. These various modules and the optimized techniques used at this stage are detailed in the following subsections.

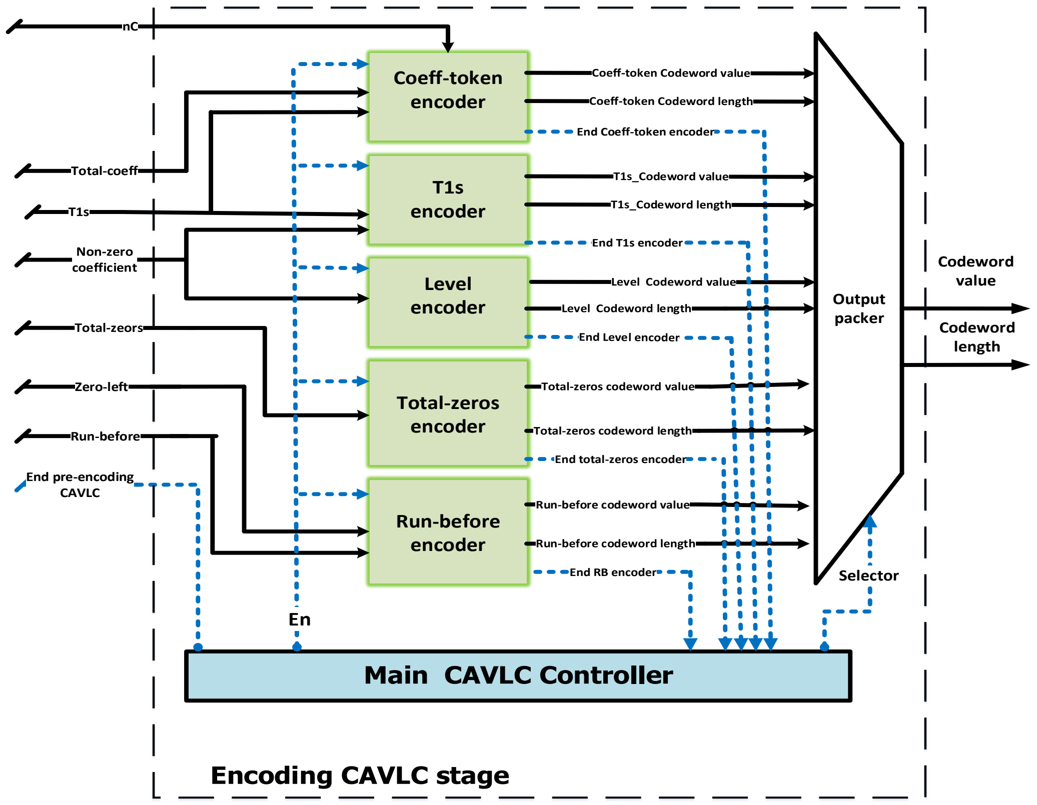


Fig. 7. Encoding CAVLC architecture

1) Optimized VLC LUTs for coeff-token and total-zero encoders:

The coeff-token and total-zero encoders are conventionally coded by different VLC LUTs in the ITU-T Recommendations [1]. However, large memory size is required to store the whole codewords' values and lengths, as presented in these traditional VLC LUTs. In the light of these details, we suggest a new representation of the codeword length and codeword value into small size. For instance, the length of the original codewords in conventional VLC coeff-token LUTs is in the range of 1 to 16, and their values are in the range of 0 to 63. Therefore, 5 bits are enough to represent the length information into the "coeff-token codeword value ROM" memory, and 6 bits are enough to represent the value information into the "coeff-token codeword length ROM" memory. An example of the new representation of codewords is given in Table I.

This method is applied for all VLC LUTS needed for coeff-token and total-zero sub-module encoders .It enables optimizing the VLC LUTs for both coeff-tokens and total-zeros. An example of an optimized VLC LUT is depicted in Fig.8.

TABLE. I. AN EXAMPLE OF A NEW REPRESENTATION OF CODEWORD IN VLC LUT

Original codeword		**Proposed codeword**	
length	***Value***	***Length***	***Value***
10000	0000000000000010	10000	000010
5 bits	16 bits	5 bits	6 bits

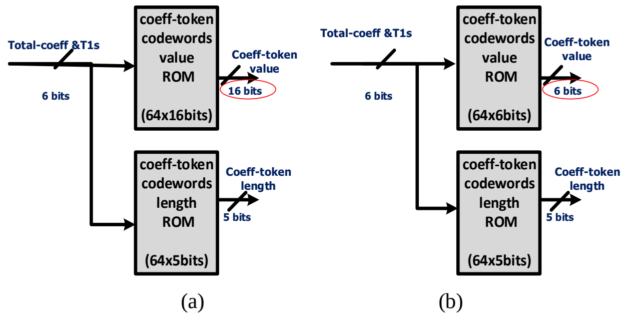

Total-coeff &T1s
6 bits
coeff-token codewords value ROM (64x16bits)
Coeff-token value
16 bits
coeff-token codewords length ROM (64x5bits)
Coeff-token length
5 bits
Total-coeff &T1s
6 bits
coeff-token codewords value ROM (64x6bits)
Coeff-token value
6 bits
coeff-token codewords length ROM (64x5bits)
Coeff-token length
5 bits
(a)
(b)

Fig. 8. Block-diagram representation of an example of (a) traditional VLC coeff-token LUT (b) optimized VLC coeff-token LUT

2) Arithmetic table elimination technique for level encoder:

Levels are encoded using the arithmetic table elimination technique to replace seven level VLC LUTs represented in the ITU-T recommendations [1]. This technique reported from [6] permitted the reduction in the memory cost area. Table II reports the pseudo-code describing the elimination procedure, which presents the advantage of a very simple implementation circuitry.

The format of the level code is arranged as follows. The maximum width of codewords' length is 28 bits.

$$\text{Code} = \underbrace{0\ldots0}_{\text{Prefix length}}\ 1\ \underbrace{x\ldots x}_{\text{Suffix length}}\ s \qquad (5)$$

Note where s is the level sign, 1 for negative, 0 for positive, and the sequence of zeros on the left of 1 and the sequence of bits on its right are respectively the level prefix and the level suffix, whose lengths, prefix length and suffix length, distinguish the codewords.

This step also illustrates the context-adaptive characteristic such that suffix length N (ranging from 0 to 6), used for encoding the actual level, must be the same one to encode the previous level. Otherwise, it will be eventually incremented if its magnitude satisfies ($3x2^{(N-1)}$). The pseudo-code of the adaptive context is shown in Fig.9.

```
If | Level | > $3x2^{(N-1)}$ then
        N <= N +1;
Else
        N <= N;
End if;
```

Fig. 9. Pseudo-code of adaptive-context at level coding step

TABLE. II. CODING ALGORITHM FOR LEVEL SYMBOL

N	Range	Coding algorithm

N=0	|level|≤7	Code=0…0 1
		Prefix length= (|level|≪1)-2+s
		Suffix length=0
		Size=prefix length+1

	8≤|level|≤15	Code=0…0 1 s Prefix length=14 Suffix length=3 Size=19 Level suffix=binary value(|level|)
	|level|≥16	Code=0…0 1 x…x s Prefix length=15 Prefix length=11 Size=28 Level suffix=|level|-1-[15≫ (N-1)]
N=1 to 6	All	Code=0…0 1 x…x s If (|level|-1< [15≪ (N-1)]) then Prefix length= (|level|-1) ≫ (N-1) Suffix length=N-1 Size =prefix length + suffix length Level suffix=|level-1|%2(N-1) Else as case |level|≥16 for N=0 End if

3) Arithmetic table elimination technique for run-before encoder:

The seven VLC LUTs required for run-before encoding are eliminated and substituted by a circuitry implementing the pseudo-code in Table III. With this approach, we achieve a reduction in the memory cost as well.

TABLE. III. CODING ALGORITHM FOR RUN-BEFORE SYMBOL

Zeroleft	Coding algorithm

<3	If Runbefore(i)=0 then
	Code=1
	Size=1
	Else
	Code=Zeroleft(i)-Runbefore(i)
	Size =Zeroleft(i)
	End

≥3 and <6	If RunBefore(i)≤6-Zeroleft(i) then
	Code=3-RunBefore(i)
	Size=2
	Else
	Code=Zeroleft(i)-RunBefore(i)
	Size=3
	End

=6	If RunBefore(i)=0 then
	Code=3
	Size=2
	Elsif RunBefore(i)=1 then
	Code=0
	Size=3
	Elsif RunBefore(i)=6 then
	Code=4
	Size=3
	Else
	  If LSB[RunBefore(i)]=0then
	  Code =RunBefore(i) >> 1
	  Else
	  Code =RunBefore(i)
	  End if
	Size=3
	End

>6	If RunBefore(i)<6 then
	Code =7-RunBefore(i)
	Size=3
	Else
	Code=1
	Size=RunBefore(i)-3
	End

4) Main CAVLC controller:

The proposed CALVC controller is presented in Fig.10. The "idlestate " represents the initial state. When the pre-encoding stage is finished (indicated by the signal "end pre-encoding CAVLC"), the finite state machine will go to the "coeff-token state". When the coeff-token encoder process is finished (indicated by the signal "end coeff-token encoding"), the finite state machine will affect the appropriate value of the signal "mux-selector" to select the output of the coeff-token encoder as final outputs. Afterwards, the finite state machine will go to the "T1s state" . When the T1s encoder process is completed, the finite state machine will produce an appropriate value for the signal "mux-selector" to select the outputs of the T1s encoder as final ones.

This process, which is produced in the coeff-token and T1s states, will be replicated at level, total-zero and run-before states. At the end of the run-before encoding process, the signal "end run-before encoding" is set high, informing that the CAVLC completely encodes the 4x4 block, and a new block can be encoded.

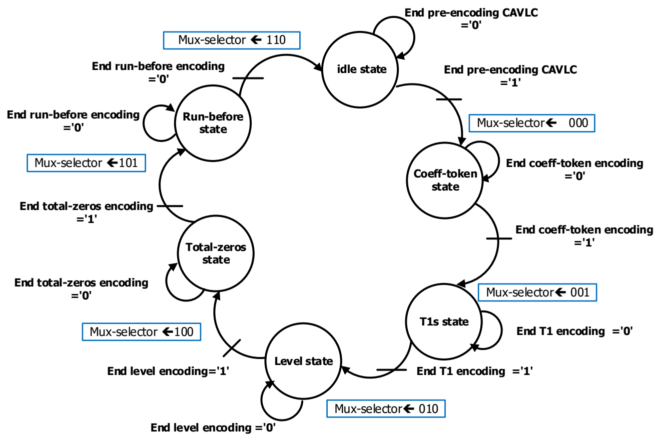


Fig. 10. Main CAVLC controller

5) Output packer:

The output packet receives as an input the signal "mux-selector" from the main controller and all the outputs of the encoder modules (codeword values and codeword lengths). Two-word multiplexers compose this module: one to select the appropriate codeword value and the other to select the appropriate codeword length. The codeword value and codeword length serve as final outputs of a CAVLC coder.

IV. PROPOSED EXP-GOLOMB ARCHITECTURE

The proposed Exp-Golomb design is presented in the form of modules in Fig.11. Every module represents the functioning way of each stage of the Exp-Golomb algorithm already explained in section II.

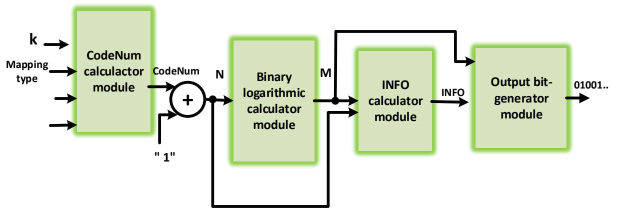

k
Mapping type
CodeNum calculactor module
CodeNum
+
" 1"
N
Binary logarithmic calculator module
M
INFO calculator module
INFO
Output bit-generator module
01001..

Fig. 11. Block diagram of Exp-Golomb

Firstly, in every k entry, the"codeNum generator" module generates the corresponding codeNum value according to a mapping type (ue, te, se or me). When the mapping type is ue, te or se, the codeNum value will be generated from the"codeNum generator1" Module. This block produces the codeNum according to various mathematical operations described in section 2, which only involves shifting, complementation, and increasing by 1. Otherwise (mapping type =me), the codeNum will be generated by a second generator module, called "codeNum generator2", based on four ROMs according to two mode types (intra or inter) and to the prediction mode. The detailed architecture of these two generators of codeNum values is shown, respectively, in Fig.12 and Fig.13.

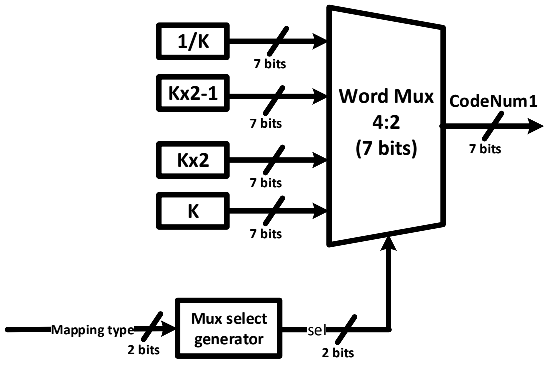


Fig. 12. Diagram of codeNum generator1

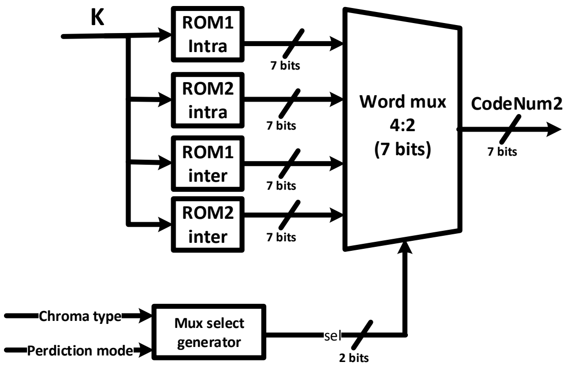


Fig. 13. Diagram of codeNum generator2

The logarithm operation is required to produce the value of M, which is utilized for the calculation of the codeword length (equivalent to 2M+1). However, its implementation requires an expensive circuit that constitutes the hardware challenge of implementing an Exp-Golomb encoder. This problem can be solved in the following way: Consider that log2(N) is equivalent to the number of M times divided by 2 until the output reaches the zero value as in equation 3. Thus, we acquire an approach to get the value of M by computing the shift operation number.

$$M = \log2(N) \leftrightarrow N = 2^M = \underbrace{2x2x2x2\ldots.x2}_{\text{M times}} \qquad (6)$$

The suggested architecture of the logarithm operation is given in Fig.14. The output of the barrel shifter is loaded in the register FF. The output Q of this register is connected to the inputs of the multiplexer and the combinatorial circuit of the OR gates. This circuit is responsible for checking whether the output Q reaches the value 0 or not by producing a one-bit value, noted C, as an output.

Initially, the counter is set at 0. If the value of Q is different from zero; the value C is equal to 1. Consequently, the AND gate will be an ascending counting; the counter will count up by a single step. In this case, the multiplexer is going to assign the value Q to the input K of the barrel shifter. When Q reaches the value 0, the value of the output C is set to 0. Therefore, the logic 0 generated on the output of the AND gate stops the counter. In this case, the output of the counter corresponds to the output value M such as M =log2 (N).

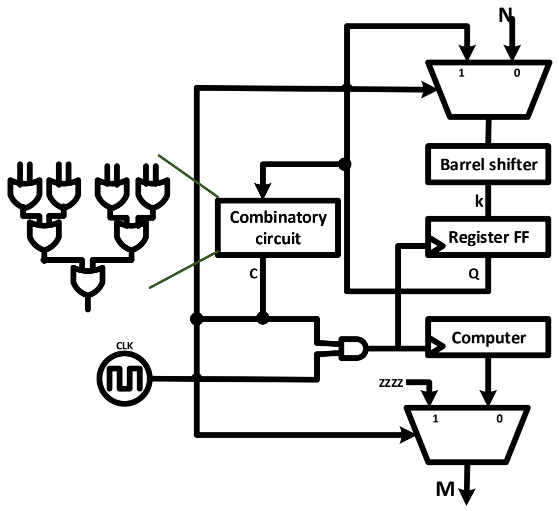


Fig.14. Architecture of binary logarithm

After the logarithm operation, the INFO value should follow formula (3), which involves shifter and subtraction operations.

The last module (Exp-Golomb bit-generator) is in charge of producing the output code word considering the value of M and INFO. It is designed by the implementation of the finite state machine that contains two states, as shown in Fig.15. The ith counter is initialized to state 1. The second state corresponds to the generation of the output codewords bit by bit, following the structure presented in formula (1). Each bit is generated in one clock.

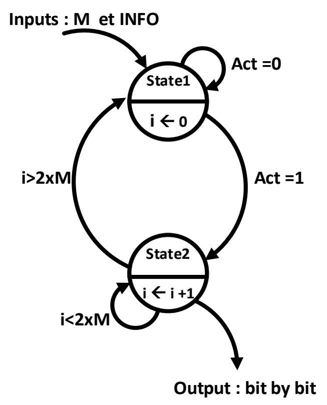


Fig. 14. Finite state machine of bit Exp-Golomb bit-generator

V. Performance Analysis and Comparison

A. Performance analysis

The proposed CAVLC and Exp Golomb architectures are modeled in VHDL, simulated, and synthesized by Modalism 6.4 and Xilinx ISE development tools 14.1, respectively. The synthesis results of physical resource utilization on Virtex VI for CAVLC and Exp-Golomb modules are reported respectively in Table VI and Table V.

TABLE. IV. Physical Resources Utilization of CAVLC Modules on Virtex VI

Module		Slice LUT	Slice register
Pre-encoding CAVLC		298	467
nC calculator		10	0
Encoding CAVLC	Coeff-token encoder	143	0
	T1s encoder	42	21
	Level encoder	316	77
	Total-zero encoder	62	0
	Run-before encoder	156	103
CAVLC controller		6	3
Output Packet		61	0
Total CAVLC		589	557

TABLE. V. Synthesis Results of Exp-Golomb on Virtex VI

Module	Slice LUT	Slice register
CodeNum generator	24	0
Binary logarithm	36	38
INFO-calculator	10	30
Exp_golomb bit-generator	44	26
Exp-Golomb Controller	26	13
Total Exp-Golomb	126	109

Through the obtained results, it is possible to verify that the CAVLC coder achieves an operation frequency of 234.14 MHz and requires an area occupancy of 847 LUTs. The maximum frequency of the Exp-Golomb architecture is 234.14 MHz, and the memory cost is 847 in terms of LUTs.

It is worth mentioning that no external or embedded memory is used to give a platform independent estimation of memory cost reduction, suitable for ASICs and FPGAs of different generations and families.

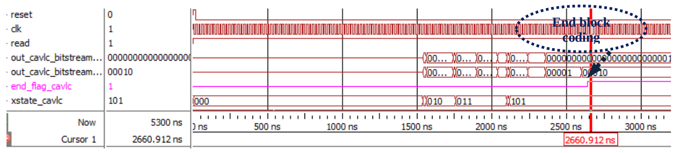


Fig. 15. Simulation results of CAVLC

The simulation results provided in Fig.16 show that the processing time per block exhibit a large variety. We take an average of 131 cycles per block. The performance of our proposed architecture is calculated as follows:

The number of clock cycles needed for 4CIF (704 x 576) video with 30-fps=

The number of clock cycles per block

x the number of blocks per macroblock

x the number of macroblock per 704

x 576 video frame x the number of frames per second=

131 x 27 x 1665 x 30 clock cycles=

176,673,150 clock cycles

The number of required clock cycles is calculated. We assume the worst case without the skip mode. This means that our suggested architecture can meet the real-time processing of 4CIF @30fps when running at 234 MHz.

B. State-of-the-art comparison

To give a reasonable comparison, both designed CAVLC and Exp-Golomb are synthesized with different FPGA platforms, as presented in Table VI.

TABLE. VI. COMPARISONS TO HIGH-PERFORMANCE DESIGNS CAVLC

	Tech	**Frequency** (MHz)	**Area**	
			Gates (ASIC)	**LUTs** (FPGA)
[2]	0.13	250	32K	-
[3]	Virtex 5	180	-	1079
[8]	Stratix IV	200	-	6549
[9]	Spartan 3	62.5	-	3447
[10]	Virtex 5	204.3	-	2563
[11]	0.18	100	73.5k	-
[12]	0.18	125	15K	-
Proposed	Spartan 3	91.43	-	754
	Virtex 5	234.14	-	847
	Spartan 3	313.87		1176

Concerning speed performance, the proposed CAVLC design exhibits a maximum operating frequency, which is mostly superior compared to other CAVLC design solutions. The memory cost of our design is also very promising, thanks to the optimized VLC LUTs and the arithmetic table elimination techniques.

TABLE. VII. COMPARISONS TO HIGH-PERFORMANCE DESIGNS EXP-GOLOMB

	Tech (um)	**Logic (LUTs)**	**Frequency (MHz)**
[10]	Virtex VI	134	309.98
[13]	Stratix II	199	191.8
Our design	Virtex VI	126	254.4

Table VII summarizes the specification of the suggested Exp-Golomb encoder and gives a comparison with the work presented in [10] and [13]. The operating frequency of the proposed architecture is lower than those presented in [10], but the suggested design has a lower area demand. Compared to the design shown in [13], the proposed architecture employs a higher area demand and a higher operating frequency.

VI. CONCLUSION

In this paper, a full hardware design entropy encoder for the H.264/AVC baseline profile has been put forward. Different techniques have been employed to improve the performance of the entropy encoder. Parallel modules are applied to speed up the coding efficiency. Meanwhile, the employment of the optimized VLC LUTs and Arithmetic method have been used to reduce the area cost. The synthesis results on Virtex IV have shown that the design occupies about 847 LUTs and can be targeted for a real-time 4CIF video format when operating at 234 MHz.

REFERENCES

[1] Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 ISO/IEC 14496-10 AVC)," in Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, JVT-G050rl, May 2003.

[2] HuiboZhong,Yibo FAN, Xiaoyang ZENG , A Parallel CAVLC Design For 4096x2160p Encoder, in Proc. 2012 IEEE International Symposium on Circuits and Systems

[3] F.L.L. Ramos, B. Zatt, T.L. Silva, A. Susin, and S. Bampi. A High Throughput CAVLC Hardware Architecture with Parallel Coefficients Processing for HDTV H.264/ AVC Enconding. In Proceedings of the 17th IEEE International Conference on Electronics, Circuits, and Systems (ICECS), 2010, Dec 2010, pages 587 – 590

[4] C. D. Chien, K. P. Lu, Y. H. Shih, and J. I. Guo, "A high performance CAVLC encoder design for MPEG-4 AVC/H.264 video coding applications," in Proc. IEEE ISCAS'06, 2006, pp. 3838–3841."

[5] Nguyen, N. M., Tran, X. T., Vivet, P., & Lesecq, S. (2012, October). An efficient Context Adaptive Variable Length coding architecture for H. 264/AVC video encoders. In Advanced Technologies for Communications (ATC), 2012 International Conference on (pp. 158-164). IEEE."

[6] Albanese, L. F., & Licciardo, G. D. (2010, September). An area reduced design of the Context-Adaptive Variable-Length encoder suitable for embedded systems. In I/V Communications and Mobile Network (ISVC), 2010 5th International Symposium on (pp. 1-4). IEEE.

[7] Tsai, C. Y., Chen, T. C., & Chen, L. G. (2006, July). Low power entropy coding hardware design for H. 264/AVC baseline profile encoder. In 2006 IEEE International Conference on Multimedia and Expo (pp. 1941-1944). IEEE.

[8] M. P. Hoffman, E. J. Balster, and W. F. Turri, "High-throughput CAVLC architecture for real-time H. 264 coding using reconfigurable devices," Journal of Real-Time Image Processing, vol. 11, pp. 75-82, 2016.

[9] Albanese, L.F.; Licciardo, G.: High-speed CAVLC encoder suitable for field programmable platforms. In: Proceedings of 2010 international conference on signals and electronic systems (ICSES), pp. 327–330 (2010)

[10] Thiele, C. C., Vizzotto, B. B., Martins, A. L., da Rosa, V. S., & Bampi, S. (2012, October). A low-cost and high efficiency entropy encoder architecture for H. 264/AVC. In VLSI and System-on-Chip, 2012 (VLSI-SoC), IEEE/IFIP 20th International Conference on (pp. 117-122). IEEE.

[11] N.-M. Nguyen, E. Beigne, S. Lesecq, P. Vivet, D.-H. Bui, and X.-T. Tran, "Hardware implementation for entropy coding and byte stream packing engine in H.264/AVC," in Proceedings of the International Conference on Advanced Technologies for Communications (ATC), Ho Chi Minh City, October 2013, pp. 360–365.

[12] Licciardo, G.D., Albanese, L.F."Design of a context-adaptive variable length encoder for real-time video compression on reconfigurable platforms." Image Process. IET 6(4), 301–308 (2012)

[13] Silva, T., Vortmann, J., Agostini, L., Bampi, S., & Susin, A. (2007, February). FPGA based design of CAVLC and exp-golomb coders for H. 264/AVC baseline entropy coding. In Programmable Logic, 2007. SPL'07. 2007 third Southern Conference on (pp. 161-166). IEEE.