PAPER

Design of a High-Throughput CABAC Encoder

Chia-Cheng LO[†], *Student Member*, Ying-Jhong ZENG[†], *Nonmember*, *and* Ming-Der SHIEH[†a)], *Member*

SUMMARY Context-based Adaptive Binary Arithmetic Coding (CABAC) is one of the algorithmic improvements that the H.264/AVC standard provides to enhance the compression ratio of video sequences. Compared with the context-based adaptive variable length coding (CAVLC), CABAC can obtain a better compression ratio at the price of higher computation complexity. In particular, the inherent data dependency and various types of syntax elements in CABAC results in a dramatically increased complexity if two bins obtained from binarized syntax elements are handled at a time. By analyzing the distribution of binarized bins in different video sequences, this work shows how to effectively improve the encoding rate with limited hardware overhead by allowing only a certain type of syntax element to be processed two bins at a time. Together with the proposed context memory management scheme and range renovation method, experimental results reveal that an encoding rate of up to 410 M-bin/s can be obtained with a limited increase in hardware requirement. Compared with related works that do not support multi-symbol encoding, our development can achieve nearly twice their throughput rates with less than 25 % hardware overhead.

key words: *H.264, CABAC, entropy encoder, high-throughput*

1. Introduction

Context-based adaptive binary arithmetic coding (CABAC) and context-based adaptive variable length coding (CAVLC) are the two entropy coding methods defined in the latest video compression standard H.264/AVC [1]. CABAC is based on the binary arithmetic coding with the probability model being dynamically adjusted by early coded syntax elements such as the macroblock (MB) type, motion vector and residual coefficient. CAVLC adopts the variable length coding with the help of early coded syntax elements to select the lookup table. Compared with CAVLC, CABAC can obtain a better compression ratio at the price of higher computation complexity. In CABAC, the range of least probable symbols (R_{LPS}) depends on not only the probability distribution employed in Q-coder [2] but also the actual value of range R. This leads to a more accurate result than employing only the Q-coder and QM-coder [3]–[5], the origins of CABAC, with an increase in the complexity of CABAC encoders. Moreover, CABAC uses a complex context-adaptive scheme, with 398 context models for further compression, and also introduces the encoding of equally probable symbols. This special procedure is implemented so that these symbols are processed with a lower cost than normal ones.

The difficulty of designing high-performance CABAC encoders comes from the fact that the current operation depends on the previous results which restrict parallel and pipelined architectures used for throughput improvement. For high-definition video applications such as 720p HD (1280×720), high-throughput designs are usually required to meet the real-time constraint. In contrast to a processor-based design approach which is generally employed for fixed bit-width operations, an application-specific design approach can effectively handle variable bit-width operations required in CABAC encoding. Li [6] adopted a dynamic pipelined architecture for a binary arithmetic coder (BAC); however, its throughput is limited by the bubble insertion due to the multi-cycle operation in the normalization stage. To further improve throughput, a fully pipelined architecture was designed for a BAC accelerator [8], [9]. In [11]–[13], the authors proposed a complete CABAC, including syntax binarization, BAC, context management, and bit packaging, for easier integration in system-on-chip designs. These designs require a single cycle to deal with one bin in the optimal scenario. In [14]–[16], a software-hardware co-design methodology was adopted to realize the whole CABAC system. Although the maximum throughput can reach four bins per clock cycle, it only handles the coefficient-related syntax elements. Moreover, the communication overhead between software and hardware could be a critical issue in [14]–[16]. Other works such as [7], [10] were focused on combining the CABAC encoder and decoder due to their similar coding flow.

Because CABAC deals with various types of syntax elements, the controller's complexity increases dramatically if multiple symbols are encoded per clock cycle. The recursive coding characteristic of CABAC also obstructs improvements in the encoding rate. In our work, a high-throughput CABAC encoder is implemented with only a slight increase in hardware overhead. The unique features in our development can be summarized as follows: 1) By analyzing the distribution of binarized bins and the syntax element processing flow, only a certain type of syntax elements with a high possibility of occurrence is processed two bins at a time. In this manner, the resulting throughput can be greatly increased with limited hardware overhead. 2) The critical path in the binary arithmetic coder is further reduced by reordering the procedure of range renovation. After the binary arithmetic coding, renormalization is performed to meet the precision of interval if the range of interval is lower than the

Manuscript received July 29, 2008.
Manuscript revised November 7, 2008.

[†]The authors are with the Department of Electrical Engineering, National Cheng-Kung University, Tainan 70101, Taiwan, ROC.
a) E-mail: shiehm@mail.ncku.edu.tw

DOI: 10.1587/transinf.E92.D.681

predefined lower bound. This data dependency will cause pipeline stalling if the renormalization procedure requires more than one cycle in a pipelined design. In our work, the range renovation is completed exactly within one cycle; therefore, the pipeline stalling is eliminated and the resulting throughput is increased accordingly. 3) An efficient memory arrangement scheme is developed to support the multi-symbol encoding. Our memory arrangement scheme also takes into account the context initialization to reduce the time required for initialization before encoding a new slice. Experimental results validate the advantages of employing the proposed design methods.

The rest of this paper is organized as follows. In Sect. 2, we briefly review the CABAC encoding process. The proposed CABAC architecture is described in Sect. 3. Experimental results are shown in Sect. 4. The conclusions are given in Sect. 5.

2. Overview of CABAC Encoding

The original alphabet in CABAC is replaced with only two notations: Least Probable Symbol (LPS) and Most Probable Symbol (MPS). The approximation of multiplication within the BAC procedure is also imported to reduce the computational time required to update the bounds of the interval. With these modifications, the interval renovation can be modified as follows:

If the input symbol is equal to MPS,

$$L_{new} = L_{old}$$
$$R_{new} = R_{MPS} = R_{old} - R_{LPS} \tag{1}$$

If the input symbol is equal to LPS,

$$L_{new} = L_{old} + R_{MPS} = L_{old} + R_{old} - R_{LPS}$$
$$R_{new} = R_{LPS} \tag{2}$$

where R_{new} and L_{new}, respectively, represent the range size and the lower bound of a new interval; R_{old} and L_{old} correspond to the range size and the lower bound of the old interval; R_{LPS} and R_{MPS} denote the size of subintervals associated with LPS and MPS, respectively.

Figure 1 shows a generic block diagram of CABAC, which is composed of three elementary components: a binarizer, a context modeler, and BAC.

2.1 Binarizer

The binarizer transforms non-binary syntax elements into

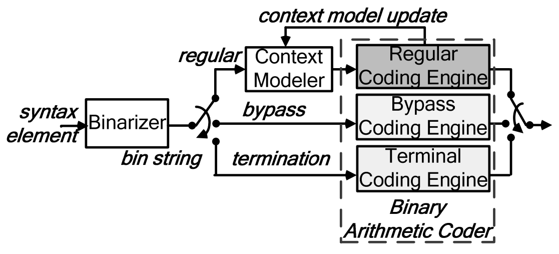


Fig. 1 The CABAC block diagram.

bin sequences, the so-called bin strings. This operation not only simplifies the original M-ary arithmetic coder into a binary arithmetic coder, but also reduces the complexity of probability renovation while using the regular mode. Within the binarizer, each syntax element has a corresponding binarization technique, including unary, truncated unary, *k*-th order Exp-Golomb, and fixed length coding, as specified in the H.264/AVC standard.

2.2 Context Modeler

One of the most important properties of CABAC is its ability to utilize a clean interface between the probability modeling and the coding procedure. In the modeling stage, a probability model called context, specifying the probability of LPS and MPS, is assigned to the current bin. Each context consists of one *valMPS* and one *pStateIdx*, where *valMPS* defines what MPS is and *pStateIdx* indicates the probability state of LPS. In the subsequent coding stage, the actual coding engine is then applied to generate a sequence of bits as a coded representation of these bins according to the selected context. Since the context determines the coded bitstream and its efficiency, it is crucial to design an adequate model that explores the statistical dependencies to a large extent and keeps this model up to date during the coding procedure.

Four basic types of context are specified in CABAC. The first one is based on the related bin values of the two neighboring elements to the left and on top of the current syntax element. The second one is only defined for two syntax element types, *mb_type* and *sub_mb_type*. For this kind of context, the values of prior coded bins $(b_0, b_1, b_2 \ldots b_{i-1})$ are used to choose the context for the current bin with the *i*th index. Both the third and fourth types are applied to the residual data only. The third one depends on the scanning order, not on the past coded data. There are two scanning orders defined in H.264/AVC. One is the zigzag scan for the frame mode; the other is the field scan for the field mode. They are used in reordering 4×4 DCT residual blocks. The fourth one specifies the modeling functions that involve the evaluation of the accumulated number of coded coefficient levels prior to the current bin.

2.3 Context Initialization & Renovation

All the contexts must be initialized at the beginning of coding a slice. Each context is initialized according to the quantization parameters of the current slice ($SliceQP_Y$), m and n, where m and n are initialization parameters defined in the H.264/AVC standard. The initialization parameters m and n may be different for each context, and the initialization procedure is based on the following equations:

$$CtxState = \text{Max}(1, \text{Min}(126, ((m * SliceQP_Y) >> 4) + n));$$
$$pStateIdx = (CtxState \le 63)?$$
$$(63 - CtxState) : (CtxState - 64);$$
$$valMPS = (CtxState \le 63)?0 : 1;$$

After initialization, *pStateIdx* and *valMPS* will be assigned to each context as the initial state. Since *pStateIdx* denotes the probability state of LPS, the probability renovation can be replaced by the variation of *pStateIdx*. Therefore, if the input symbol is equal to MPS, *pStateIdx* is increased; otherwise *pStateIdx* is decreased during the CABAC encoding procedure.

2.4 Binary Arithmetic Coder

The binary arithmetic coder supports three execution modes: the regular, bypass, and termination modes. When there is a high correlation between the current coding bin and early coded bins, the regular mode is used and the context that records the associative probabilities are imported to deal with the bin. From Eqs. (1) and (2), R_{LPS} is based on the current range $R[7:6]$ (the two most significant bits of R) and *pStateIdx* (see Table 9-35 in the H.264/AVC standard); each *pStateIdx* has four values of R_{LPS}; one of the four values of R_{LPS} is chosen according to $R[7:6]$.

The arithmetic coding is done by recursive subinterval division. Thus, the range size of the interval becomes smaller and smaller as the arithmetic coding proceeds. In contrast, the precision needed to represent R becomes larger and larger during the coding procedure. To prevent an increase in the required precision, renormalization is used when R becomes lower than the predefined lower bound (256). Renormalization is done by recursively multiplying R by 2 until it is larger than the predefined lower bound; hence, R is always higher than 256 before the next bin is coded. The times of renormalization depends on the number of leading zero (LZ) bits of R. The bypass mode is used if the correlation of bins is low, e.g., the coefficient levels. Since the probabilities of 0's and 1's are both 0.5, no context is needed and only one left shit operation is necessary to renormalize the computed R to prevent it from being lower than 256. The termination mode is adopted only when two syntax element types, i.e., *mb_type* and *end_of_slice*, are coded to terminate CABAC coding.

3. Proposed CABAC Encoder Architecture

The number of cycles needed to encode one syntax element is proportional to the number of bins generated by the binarizer. The encoding rate can then be increased by allowing multiple bins to be processed at a time. However, by doing so, the complexity of controllers and the associated data path components are increased. How to derive a high-throughput CABAC encoder with only a slight increase in hardware overhead is thus a challenging task.

We observed that when handling syntax elements, most generated bins and their processing orders are rather regular. Simulation results also reveal that most bins are acquired from binarized syntax elements associated with the residual data, including the significant coefficient (SC) map, signs of residual coefficients, and residual coefficient levels. Table 1 shows the distribution of binarized bins among different video sequences. In this table, the number of bins obtained from both the SC map and coefficient levels occupies up to 75 % of the total number of bins. This implies that dealing with only these syntax elements can efficiently speed up the encoding rate. In this manner, we can improve the resulting throughput with only a slight increase in the controller's complexity.

Figure 2 depicts the block diagram of the proposed CABAC encoder. The main building blocks are described in the following subsections.

3.1 Syntax Element Processing Flow

According to the syntax element processing flow specified in the H.264/AVC standard, the coding flow associated with the residual data (SC map and coefficient levels) is rather simple compared with the entire encoding system. The overhead of handling residual data can thus be greatly alleviated if we process two bins per clock cycle only when encoding the SC map and coefficient levels. As a result, the binarizer has to provide one or two bins per clock cycle based on the type of syntax elements. In our development, it is assumed that the bins obtained from the binarizer are consumed immediately. This implies that the bin generation rate of the binarizer is exactly equal to the bin consumption rate of the binary arithmetic coder; this avoids the need for additional buffers to store the generated bins.

Table 1 Distribution of binarized bins.

Sequences	# of Bins	SC Map	Coefficient Levels	Total
akiyo	1699082	430020 (25.31%)	401296 (23.62%)	48.93%
mother	2044590	425958 (20.83%)	355163 (17.37%)	38.20%
hall	2980897	885242 (29.70%)	830308 (27.85%)	57.55%
news	3568014	1044353 (29.27%)	1037967 (29.09%)	58.36%
silent	4141896	1267472 (30.60%)	1047218 (25.28%)	55.88%
foreman	5318555	1510215 (28.40%)	1273117 (23.94%)	52.34%
coastguard	9522209	3841470 (40.34%)	2885745 (30.31%)	70.65%
mobile	13454760	5185887 (38.54%)	4733768 (35.18%)	73.72%
stefan	18200193	6775738 (37.23%)	6926396 (38.06%)	75.29%

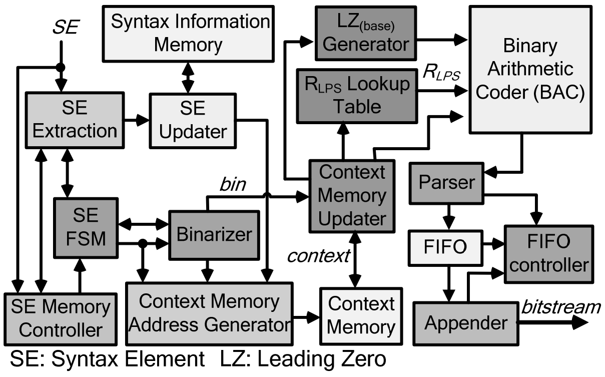


Fig. 2 Block diagram of the proposed CABAC encoder.

3.2 Syntax Element Extraction

The context selection for some specific bins is based on a modeling function of the related bin values of previous syntax elements in neighboring macroblocks (MBs) to the left of and on top of the current MB as stated in Sect. 2.2. As a result, syntax elements of the current MB must be stored for choosing appropriate contexts in future coding procedure. In fact, we can store only required information instead of the raw syntax elements to reduce memory space. Applying such a modification in our development, merely 152 bits of memory space are sufficient to store the extracted information of one MB. Note that exactly one row of MBs within the current slice must be held because only those MBs to the left of and on top of the current one may be referred to. Hence, the size of the single-port SEE (syntax element extraction) memory is proportional to the picture width in terms of MB.

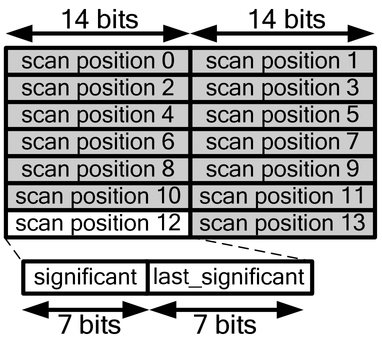
3.3 Context Memory Arrangement

The context memory is partitioned into two distinct parts in our design because different strategies are used to deal with bins associated with the residual block and remaining contexts. The first part, denoted as *SigLast* context memory, is used to handle two bins per clock cycle for the SC map in the residual block and the second one, named as *ExSigLast* context memory, is allocated to process remaining contexts. Note that dual-port SRAM is adopted to implement the context memory for supporting concurrent read and write operations in our development.

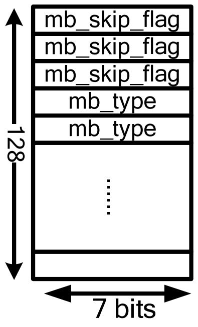
Figure 3 (a) shows the arrangement of *SigLast* context memory which consists of contexts associated with the SC map. Because at most two bins may be encoded for the SC map, each entry in *SigLast* context memory contains four 7-bit contexts (two for "significant" and two for "last_significant"). The scan position numbers in Fig. 3 (a) are arranged based on the scan order as stated in Sect. 2.2. Each context in a scan position comprises the values of *valMPS* (1 bit) and *pStateIdx* (6 bits), as depicted in Fig. 3 (c), to determine the value of R_{LPS}.

The *ExSigLast* context memory shown in Fig. 3 (b) stores the remaining contexts with each entry consisting of only one context. Note that the contexts associated with coefficient levels in the residual block are stored in *ExSigLast* context memory because the same context can be recursively used while encoding contiguous bins of coefficient levels. In this manner, the updated context can be used immediately to encode the current bin to save memory space and still support the need of multi-symbol encoding for the coefficient levels. In other words, the updated context can be directly sent to the binary arithmetic coder when written back into *ExSigLast* context memory. Moreover, according to CABAC encoding flow, different slice coding procedures (I, P or B slice) will involve different sets of contexts; therefore, only the required contexts are loaded to *ExSigLast* context memory during context initialization procedure. As a result, it takes only storage of 128*7 bits, instead of 276*7 bits, to store all the contexts except for those associated with the SC map.

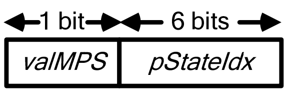
Because there are 398 contexts in CABAC encoding and context initialization should be done before delivering the first MB of the current slice to the CABAC core, the encoding performance will be strongly affected by the frequent initialization process if not properly handled. To reduce the initialization latency, the initialization table is divided into five smaller ones according to the syntax element type and the slice type since only a part of contexts are used in different slice coding procedures. Figure 4 depicts the partitioned initialization table in which two of the sub-tables are associated with the SC map and three are for the remaining syntax elements. Note that the contexts associated with the SC map and those with the remaining syntax elements can be initialized concurrently to further reduce the initialization time because they are allocated in different parts of the context memory. The avoidance of loading unused contexts also helps to reduce the initialization time.



(a) *SigLast* context memory.



(b) *ExSigLast* context memory.



(c) Structure of context in (a) and (b).

Fig. 3 Context memory arrangement.

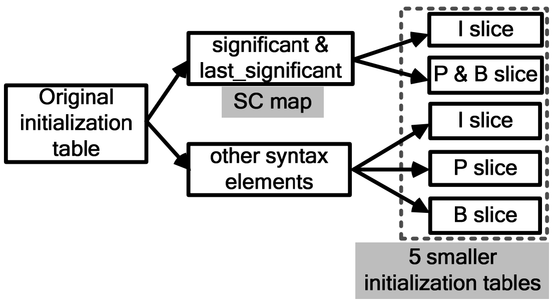

Original initialization table
significant & last_significant
SC map
other syntax elements
I slice
P & B slice
I slice
P slice
B slice
5 smaller initialization tables

Fig. 4 Initialization table partitioning.

3.4 Binary Arithmetic Coder Design

Equations (3), (4) and (5) show the renovation of R and L when two bins are encoded at the same time. In Eqs. (3) and (4), bin_1 and bin_2 denote the 1st and 2nd bin coming from binarizer; R_{LPS1} and R_{LPS2} are the range of LPS for the bin_1 and bin_2 respectively. Intuitively, R_{LPS2} depends on the value of R_{temp} which is determined by the value of bin_1. To get L_{new}, L_{inc1} and L_{inc2} are the increases should be added to L_{old} according to the value of bin_1 and bin_2 respectively.

$$R_{temp} = \begin{cases} (R_{old} - R_{LPS1}), & \text{if bin}_1 = valMPS_1 \\ R_{LPS1}, & \text{otherwise} \end{cases}$$
$$R_{new} = \begin{cases} (R_{temp} - R_{LPS2}), & \text{if bin}_2 = valMPS_2 \\ R_{LPS2}, & \text{otherwise} \end{cases} \tag{3}$$

$$L_{inc1} = \begin{cases} 0, & \text{if bin}_1 = valMPS_1 \\ (R_{old} - R_{LPS1}), & \text{otherwise} \end{cases}$$
$$L_{inc2} = \begin{cases} 0, & \text{if bin}_2 = valMPS_2 \\ (R_{temp} - R_{LPS2}), & \text{otherwise} \end{cases} \tag{4}$$

$$L_{new} = L_{old} + L_{inc1} + L_{inc2} \tag{5}$$

Because the required operations in Eqs. (3) and (4) are the same and the derivation of the range R_{new} is independent of the value of L, we can employ a two-stage pipelined structure to improve the operation frequency. That is, those operations related to R are accomplished in the first stage; the derived values (L_{inc1} and L_{inc2}) are then added to the old value of L to obtain the new L (L_{new}). Since the binary arithmetic coder has to deal with either one or two bins in one cycle depending on the content of syntax elements, we can implement those related operations on R in the first stage as a cascaded structure of two identical units. The unit designed to handle one bin at a time is shown in Fig. 7 (a) which can be directly mapped from Fig. 5 (a). When the output is connected to the input R of the following identical unit, the cascaded structure can then be used to take care of two bins at a time. Because there are only two simple addition operations in the second stage, the resulting critical path delay of R renovation in the first stage becomes the bottleneck of the pipelined structure. As a result, reducing the critical path of R renovation can further improve the encoding rate.

Figure 5 (a) gives the pseudo-codes of R renovation according to the reference software of H.264/AVC. From Fig. 5 (a), we observe that R must be renormalized (see the *RenormE* function) while the value of R is lower than the predefined lower bound. However, the times of renormalization cannot be known until the new R is determined according to the current bin value (binVal). Therefore, the critical path delay is dominated by the accumulated delay for accomplishing the required operations to find R_{MPS} and the following normalization process. To relax this data dependency, we adopt a two-step normalization scheme to reduce the critical path delay. In the first step, normalization is done depending on the value of *pStateIdx* which is known in advance; therefore, it can be executed in parallel with the required operations to find R_{MPS}. In the second step, at most a simple left shift will be required as shown in Fig. 5 (b), discussed as follows.

```
R_Renovation (valMPS, pStateIdx, binVal)
  pRIdx=R [7:6];
  $R_{LPS}$=rangeTabLPS[pStateIdx][pRIdx];
  $R_{MPS}$=R - $R_{LPS}$;
  if (binVal != valMPS) R=$R_{LPS}$;
  else                  R=$R_{MPS}$;
  RenormE();     //Described below
End
RenormE()
  if (R<256)
    {R=R<<1; RenormE();} //Recursive normalization
  else
    exit(); //Leave the RenormE()
End
```

(a) Pseudo-code of R renovation according to reference software.

```
R_Renovation_Modified (valMPS, pStateIdx, binVal)
  pRIdx=R [7:6];
  $R_{LPS}$=rangeTabLPS[pStateIdx][pRIdx];
  $LZ_{(base)}$=LZG(pStateIdx); //Refer to Fig. 6 (b)
  $R_{MPS}$=R - $R_{LPS}$;
  $R_{LPS}$= ($R_{LPS}$<<($LZ_{(base)}$));
  if (binVal == valMPS)  R=$R_{MPS}$;
  else                   R=$R_{LPS}$;
  R=(R[8]!=1)?(R<<1):R; //At most one left-shift
End
```

(b) Modified flow of R renovation.

Fig. 5 R renovation flow.

To justify the derivation of Fig. 5 (b), we partition the R renovation procedure into two cases. (C.1) $R = R_{LPS}$: In such a case, one to seven times of renormalization might be done according to the number of leading zeros (LZs) in R_{LPS} (see Fig. 5 (a)). This implies that the number of required renormalization operations is unknown before computing the value of R_{LPS}. However, as can be observed from Table 9-35 (Specification of R_{LPS} depending on *pStateIdx* and $R[7:6]$) in the H.264/AVC standard, the difference of LZs for a given *pStateIdx* is not larger than one regardless of the value of $R[7:6]$. As an example, Fig. 6 (a) shows one of the rows in Table 9-35. For *pStateIdx* = 36, the value of R_{LPS} is ranged from 22 to 37, which means that the number of LZs is either 3 for $(37)_{10} = (\mathbf{000}100101)_2$, or 4 for $(22)_{10} = (\mathbf{0000}10110)_2$. Since at least three times of renormalization are required, these three operations are performed in the first step of renormalization. The extra renormalization is deferred to the next step if necessary. Note that this property holds for all the rows in Table 9-35 as listed in Fig. 6 (b), in which the second column shows the number of LZs of the four R_{LPS}'s; the last column denotes the minimum number of LZs for each *pStateIdx*. (C.2) $R = R_{MPS}$: In this case, the value of R_{MPS} should be larger than 128 before encoding the next bin because the conditions $R \geq 256$

pStateIdx	R[7:6]			
	0	1	2	3
36	22*	27	32	37**

* $(22)_{10}=(\underline{000}10110)_2$
**$(37)_{10}=(\underline{000}100101)_2$

(a) Example of the LZs for the four R_{LPS}'s assuming *pStateIdx* = 36.

pStateIdx	No. of LZs in four R_{LPS}'s	$LZ_{(base)}$
0-2	1	1
3-12	2-1	
13-15	2	2
16-25	3-2	
26-29	3	3
30-38	4-3	
39-42	4	4
43-52	5-4	
53-57	5	5
58-62	6-5	
63	7	7

(b) Summary of LZs for different values of *pStateIdx*.

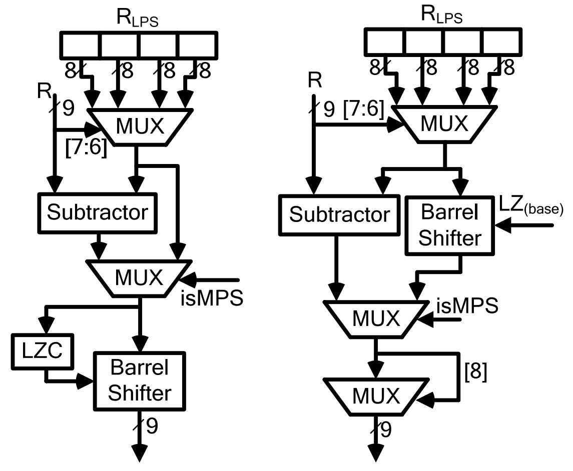
Fig. 6 Derivation of $LZ_{(base)}$.

and $R_{MPS} \geq (1/2) * R$ are always satisfied. This implies that at most one renormalization is needed. The scenario is the same in the final step while R_{LPS} is chosen.

Based on the discussion above and Fig. 5 (b), Fig. 7 (b) depicts the modified structure of R renovation, in which subtraction is performed concurrently with the barrel shifting, and the extra renormalization step, if needed, is accomplished by using a simple multiplexing operation. In Figs. 5 (b) and 7 (b), the notation $LZ_{(base)}$ is used to denote the minimum number of leading zeros for a given *pStateIdx*. With this modification, the resulting critical path delay can then be shortened, thus enhancing the overall operating frequency.

3.5 Parser and Appender

If the range of the interval is lower than the predefined lower bound, renormalization will be done and some leading bits of L will be shifted out to meet the precision of the interval. However, if the shifted bits are "0111...," the result may be later modified to "1000...". This is the so-called carry propagation problem. Since the shifted bits may need to be corrected in the future coding procedure, they cannot be sent until they are asserted correctly. Therefore, a large FIFO may be used to store these un-verified bits. Instead of holding un-verified bits, they can be packed as a token and then stored in FIFO [14]. The packed token format is given in Fig. 8 (a), where *stuf_flag* specifies that the stuffing word is 16'h0000 (*stuf_flag* = 0) or 16'hFFFF (*stuf_flag* = 1), *stuf_words_num* denotes the number of stuffing words and *leading_bitstream* contains the value to be transmitted ahead of the stuffing words. Figures 8 (b) and (c) illustrate the transformation between tokens and transmitted bitstreams.

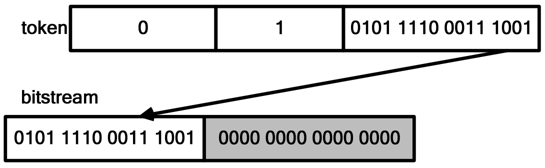


(a) Original structure. (b) Modified structure.

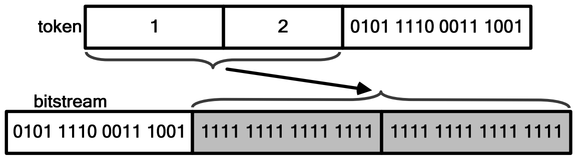
Fig. 7 Structures of *R* renovation.

stuf_flag	*stuf_words_num*	*leading_bistream*

(a) Token format.



(b) Example with *stuf_flag* =0 and *stuf_words_num* =1.



(c) Example with *stuf_flag* =1 and *stuf_words_num* =2.

Fig. 8 Transformation between tokens and bitstream.

Finally, an appender is used to unpack these tokens, produce the actual transmitted bitstream, and deliver them to external circuits. With the packed token format, more unsent bits could be held using the same FIFO size.

4. Experimental Results and Performance Analysis

The proposed CABAC encoder, designed with a 6-stage pipelined architecture, was coded in Verilog hardware description language, simulated extensively based on the test patterns acquired from the reference software, and synthesized using 0.18 μm processes. Tables 2 and 3 respectively list our synthesized results and performance comparisons with related works. Compared with [7], [13] and [18] which do not support multi-symbol encoding, our development can achieve twice their throughput rates with limited hardware overhead (less than 25 %). Note that the much higher area

Table 2 Synthesis and results of CABAC encoders.

	Technology (μm)	Speed (MHz)	Area	Normalized Area
Ours	TSMC 0.18	216	0.432 mm^2/43.2K gates	1
[13]	TSMC 0.18	190	0.355 mm^2	0.82
[18]	TSMC 0.13	200	34.2K gates	0.79
[7]	TSMC 0.18	230	0.534 mm^2/ 53.4K gates	1.23
[16]	AMS 0.35	186	19.4K gates	0.449

Table 3 Performance comparison with other designs.

Design	Speed (MHz)	Processing Rate (bin/cycle)		Throughput (M-bin/s)	
		Average	Max	Average	Max
Ours	216	1.5-1.9	2	324-410	432
[13]	190	1	1	190	190
[18]	200	0.67	1	134	200
[7]	230	0.6	1	138	230
[16]	186	1.9-2.3 (HW)	4	353-428 (HW)	744

requirement in [7] comes from its additional support for CABAC decoding.

The reason why the area requirement in [16] is much less than others is because their hardware design handles only the syntax elements of residual coefficients, and other syntax elements are processed by the embedded processor. Although the processing rate of their hardware design can reach 2.3 bins per cycle, the overall encoding rate is limited by the processor because the optimal processing rate of the processor is 10 cycles per bin as mentioned in their previous work [14]. In order to estimate the throughput of [16], we assume the percentage of bins associated with residual coefficients is 75 % which is the best scenario in Table 1. Therefore, the estimated average throughput of Osorio's HW/SW co-design is $[2.3*0.75+(1/10)*0.25]*186 = 325$ (M-bin/s) when considering the SW performance. The throughput of our design is $[2 * 0.75 + 1 * 0.25] * 216 = 378$ (M-bin/s). Furthermore, if the performance degradation caused by the communication between HW and SW is considered, the estimated average throughput of Osorio's design will become lower. As a result, our design still outperforms Osorio's work.

To gain insight into the effectiveness of the proposed CABAC encoder, experiments were built to simulate video sequences with parameters listed in Table 4, where the Level in AVC/H.264 defines the upper bound of bit rates for transmission. We use the AVC/H.264 reference software to encode the video sequences and enable the rate controller to reach the maximum bit-rate constraint. The syntax elements generated from the reference software are then sent to the proposed CABAC encoder to verify the results and check the number of required cycles to complete the task. From Table 4, one can observe that the proposed design can process Level 3.2 720p HD video sequences at a clock rate of less than 27 MHz. Note that the equivalent clock rate is defined as the total cycles multiplying (60/181) in this case. Based on the experimental results in Table 4, it is expected that the proposed CABAC encoder can handle Level 5.0 (3673 × 1536) video sequences when it is operated at its maximum operation frequency, i.e. 216 MHz.

Table 4 Simulation results for example sequences.

Sequences	mobcal	parkrun	shields	stockholm
Level	3.2			
Resolution	1280×720			
Bitrate (k-bit/sec)	20000			
Frame Rate (frame/sec)	60			
Total Frames	181			
Total Cycles	80197927	77723077	75704950	79442736
Equivalent Clock Rate (MHz)	26.73	25.91	25.23	26.48

5. Conclusions

A new architecture for implementing high-throughput CABAC encoders has been presented in this paper. Using the developed schemes, the resulting CABAC encoding rate can be greatly increased with limited hardware overhead. The main features of our design are summarized as follows. 1) An efficient memory arrangement scheme is applied to reduce the hardware requirement and the time required for memory initialization. 2) Based on a prior analysis of bin distribution, the throughput rate is effectively improved by allowing only certain types of syntax elements to be processed two bins at a time. 3) The operating frequency is increased by adjusting the data flow in the binary arithmetic coder. 4) The procedure of range renovation can be accomplished within one cycle to avoid pipeline stalling. 5) A bit-packaging scheme is adopted to deal with the problem of long carry propagation at the output of CABAC encoder using a limited buffer size. Our design can achieve an encoding rate of up to 410 M-bin/s under the best case scenario with limited hardware overhead.

Acknowledgments

This work was supported in part by the National Science Council of R.O.C. under Contract NSC 96-2220-E-006-008.

References

[1] ITU-T Recommendation H.264, Advanced video coding for generic audiovisual services, May 2003.
[2] W.B. Pennebaker, J.L. Mitchell, G.G. Langdon, Jr., and R.B. Arps, "An overview of the basic principles of the Q-coder adaptive binary arithmetic coder," IBM J. Res. Dev., vol.32, pp.717–726, 1988.
[3] K.M. Marks, "A JBIG/ABIC compression engine for digital document processing," IBM J. Res. Dev., vol.42, pp.753–758, 1998.
[4] S.H. Burroughs and T.R. Lattrell, "Data compression technology in ASIC cores," IBM J. Res. Dev., vol.42, pp.725–731, 1998.
[5] M.J. Slattery and J.L. Mitchell, "The Qx-coder," IBM J. Res. Dev., vol.42, pp.767–784, 1998.
[6] L. Li, Y. Song, T. Ikenaga, and S. Goto, "A CABAC encoding core with dynamic pipeline for AVC/H.264 main profile," Proc. IEEE Asia Pacific Conf. Circuits Syst., pp.760–763, Dec. 2006.

[7] L. Li, Y. Song, T. Ikenaga, and S. Goto, "Hardware architecture design of CABAC codec for AVC/H.264," Proc. IEEE Int. Symp. VLSI Design Autom. Test, pp.1–4, April 2007.
[8] J.L. Nunez-Yanez and V.A. Chouliaras, "High-performance arithmetic coding VLSI macro for the H.264 video compression standard," IEEE Trans. Consum. Electron., vol.51, no.1, pp.144–151, Feb. 2005.
[9] J.L. Nunez-Yanez and V.A. Chouliaras, "Design and implementation of a high-performance and silicon efficient arithmetic coding accelerator for the H.264 advanced video codec," IEEE Int. Conf. Applica. Specific Syst. Architect. Processors, pp.411–416, July 2005.
[10] J.L. Nunez-Yanez, V.A. Chouliaras, D. Alfonso, and F.S. Rovati, "Hardware assisted rate distortion optimization with embedded CABAC accelerator for the H.264 advanced video codec," IEEE Trans. Consum. Electron., vol.52, no.2, pp.590–597, May 2006.
[11] S. Sudharsanan and A. Cohen, "A hardware architecture for a context-adaptive binary arithmetic coder," Proc. SPIE Embed. Process. for Multimedia and Commun., vol.5683, pp.104–112, March 2005.
[12] H. Shojania and S. Sudharsanan, "A high performance CABAC encoder," Proc. IEEE Int. Conf. North-East Workshop Circuits Sys., pp.315–318, June 2005.
[13] H. Shojania and S. Sudharsanan, "A VLSI architecture for high performance CABAC encoding," Proc. SPIE Int. Soc. Optical Eng., vol.5960, no.3, pp.1444–1454, July 2006.
[14] R.R. Osorio and J.D. Bruguera, "Arithmetic coding architecture for H.264/AVC CABAC compression system," Proc. IEEE Euromicro Symp. Digit. Syst. Design, pp.62–69, Sept. 2004.
[15] R.R. Osorio and J.D. Bruguera, "A new architecture for fast arithmetic coding in H.264 advanced video coder," Proc. Euromicro Conf. Digit. Syst. Design, pp.298–305, Aug. 2005.
[16] R.R. Osorio and J.D. Bruguera, "High-throughput architecture for H.264/AVC CABAC compression system," IEEE Trans. Circuits Syst. Video Technol., vol.16, no.11, pp.1376–1384, Nov. 2006.
[17] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," IEEE Trans. Circuits Syst. Video Technol., vol.13, no.7, pp.620–636, July 2003.
[18] P.S. Liu, J.W. Chen, and Y.L. Lin, "A hardwired context-based adaptive binary arithmetic encoder for H.264 advanced video coding," Proc. IEEE Int. Symp. VLSI Design Autom. Test, pp.1–4, April 2007.

Chia-Cheng Lo was born in Tainan, Taiwan, R.O.C., in 1982. He received his B.S. degree in Electrical Engineering from National Cheng Kung University, Tainan, Taiwan, in 2004, where he is now pursuing his Ph.D. degree. His research interests are transform coding, pre- and post video processing, multimedia reconfigurable architecture design, and electronic system level SoC architecture design.

Ying-Jhong Zeng was born in Tainan, Taiwan, R.O.C., in 1982. He received his M.S. degree in Electrical Engineering from National Cheng Kung University, Tainan, Taiwan, in 2006. His research interests include video compression system, VLSI design and testing. He is now with the HimaxDisplay Corporation.

Ming-Der Shieh received the B.S. degree in electrical engineering from National Cheng Kung University, in 1984, the M.S. degree in electronic engineering from National Chiao Tung University, Taiwan, in 1986, and the Ph.D. degree in electrical engineering from Michigan State University, East Lansing, in 1993. From 1988 to 1989, he was an engineer at United Microelectronic Corporation, Taiwan. From 1993 to 2002, he was with the faculty of Department of Electronic Engineering, National Yunlin University of Science & Technology. He received the teaching award from NYUST in 1998 and was the department chairman from 1999 to 2002. Since 2002, he has been with the Department of Electrical Engineering, National Cheng Kung University, where he is currently an associate professor. His research interests include VLSI design and testing, VLSI for signal processing, digital communication, and computer-aided design.