PAPER

A Novel Low-Cost High-Throughput CAVLC Decoder for H.264/AVC

Kyu-Yeul WANG[†a)], **Byung-Soo KIM**[†], **Sang-Seol LEE**[†], ***Nonmembers***, **Dong-Sun KIM**[††], ***Member***, ***and*** **Duck-Jin CHUNG**[†], ***Nonmember***

SUMMARY This paper presents a novel low-cost high-performance CAVLC decoder for H.264/AVC. The proposed CAVLC decoder generates the length of *coeff_token* and *total_zeros* symbols with simple arithmetic operation. So, it can be implemented with reduced look-up table. And we propose multi-symbol *run_before* decoder which has enhanced throughput. It can decode more than 2.5 symbols in a cycle if there are *run_before* symbols to be decoded. The hardware cost is about 12 K gates when synthesized at 125 MHz.

key words: *CAVLC decoder, multi-symbol decoder, VLSI, H.264/AVC*

1. Introduction

There are some needs for a low-cost and high-performance multimedia codec because high-quality multimedia data is used in various mobile devices. To meet the needs, H.264/AVC is developed by Video Coding Expert Group of ITU-T and Moving Picture Expert Group of ISO/IEC. Several new features like Quarter-pixel precision motion estimation, various intra prediction modes, integer transformation, adaptive in-loop filter, and enhanced entropy coding are adopted for higher coding efficiency. Because of these, H.264/AVC has an enhanced compression rate. But the complexity increment of H.264/AVC codec incurs a cost-effectiveness problem of the development of H.264/AVC codec [13]. So, hardware implementation of H.264/AVC codec is inevitable.

Context-based Adaptive Variable Length Coding (CAVLC), which is an entropy coding method of H.264/AVC, is used to encode and decode zig-zag scanned 4×4 or 2×2 residual data. Next decoding step can't be started until current decoding procedure is finished because CALVC consists of variable length symbols. Therefore, each decoding step is processed sequentially. So, CAVLC decoder has to be implemented carefully for the real-time high-quality mobile application system.

This paper presents a low-cost high-throughput CAVLC decoder architecture which exploits CALVC features. The proposed CAVLC decoder has features like efficient decoding methods for Variable Length Code Tables (VLCTs), High throughput multi-symbol *run_before* decoding, and a novel flush unit to renew the bit-stream registers without delay.

The rest of the paper is organized as follows. CAVLC decoding flow and previous works are described minutely in Sect. 2. The proposed CAVLC decoder architecture is presented in Sect. 3. In Sect. 4, verification method and implementation results are depicted. Finally, conclusion is made in Sect. 5.

2. CAVLC Decoding Flow and Previous Works

Zig-zag scanned coding, run-length coding, and CAVLC are adopted to improve coding efficiency of residual data compression in H.264/AVC. CAVLC decoding is built on five sub-decoding steps which are shown in Fig. 1.

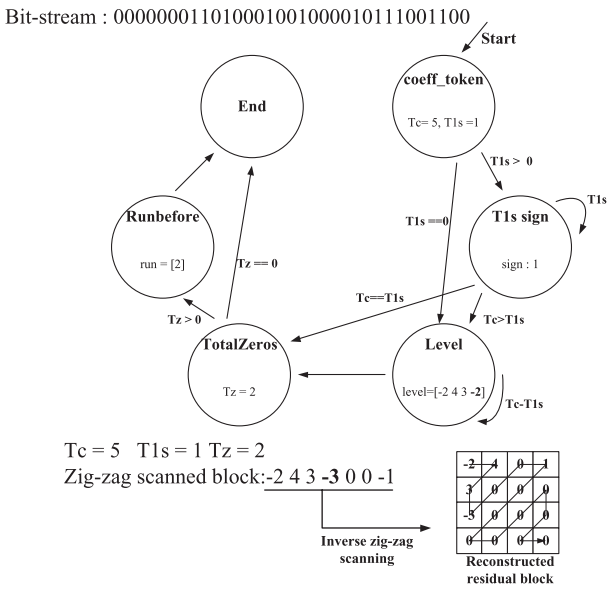
coeff_token decoding as first step of CAVLC decoding is processed to decode the number of non-zero coefficients (Tc) and the number of trailing ones ($T1s$) in the reconstructed residual block as depicted in Fig. 1. The values are used to decide the number of times that the following sub-decoding steps should be processed. In the next step, the signs of trailing ones ($T1s\_sign$) decoding is performed to decode each sign of trailing ones. The trailing ones are last coefficients which has absolute value '1' in the zig-zag scanned block data. Each sign value is decoded with following one bit in the reverse order. That is, the $T1s\_sign$ decoding is processed $T1s$ times. The reverse order means the decoding process is done from the last coefficient or value in the zig-zag scanned block data.

And then *Level* decoding is carried out to decode non-zero coefficients except trailing ones in the zig-zag scanned residual data. *Level* symbols are decoded in the reverse order and the times to be processing are Tc-$T1s$. That is, when Tc is equal to 0 or $T1s$, the decoding process is eliminated. In the example of Fig. 1, the first decoded level's absolute value is incremented by 1. It is conditional exception for reduction of bit-stream length. That is explained in following sub-clause 3.2. The following step is *total_zeros* decoding to decode the number of zeros before the last non-zero coefficients in zig-zag scanned residual data. To decode *total_zeros*, there are two different tables for 4×4 blocks and 2×2 chroma DC blocks. *total_zeros* decoding is not carried out and the value of *total_zeros* is set to zero when Tc is equal to *maxNumCoeff* and *total_zeros* decoding is ignored when Tc is zero. The *maxNumCoeff* is set to sixteen,

Manuscript received July 7, 2010.
Manuscript revised November 16, 2010.

[†]The authors are with the School of Information and Communication, INHA University, Incheon, Korea.
[††]The author is with the Department of DxB Communication Convergence Research Center, Korea Electronics Technology Institute, Korea.
a) E-mail: kyuyeul.wang@inha.edu

DOI: 10.1587/transinf.E94.D.895



Code	Elements	Value	Output array
0000000110	*Coeff_token* (*Tc, T1s*)	*Tc*=5, *T1s*=1	Empty
1	*T1s_sign*	-(reverse)	**-1**
0001,0010, 00010,111	*Level, Level, Level, Level*	-2(-3), +3, +4, -2 (reverse)	**-2**, **4**, **3**, **-3**, -1
0011	*Tz*	2	-2, 4, 3, -3, -1
00	*Run_before*	2 (reverse)	-2, 4, 3, -3,**0**,**0**, -1

Fig. 1 Conventional CAVLC decoding flow and example.

fifteen, or four depending on the type of residual blocks. In *run_before* decoding step, the number of zeros between adjacent coefficients is decoded in the reverse order. For *run_before* decoding, *run_before* decoder use VLCTs which are partitioned by *zeroLeft*. The *zeroLeft* is initialized with *total_zeros* and renewed with *zeroLeft* which is decreased by *run_before*. The *run_before* symbol decoding is processed until the *zeroLeft* is zero or there are no more *run_before* symbol that means *run_before* decoding is processed *Tc*-1 times. Finally, the decoded coefficients in the *level* decoding and *run_before* symbols are merged to reconstruct residual data.

CAVLC symbols except trailing ones' sign (*T*1*s_sign*) are encoded by using Exp-Golomb code which consists of leading zeros, '1', and suffix (*info*). So, it is important to find the number of leading zeros (*leading_zeros*) rapidly, in CAVLC decoding. In the ref. [1], Di proposed an efficient leading zeros detector which is adopted in vast literatures. Look-up table (LUT) and memory architectures considering the number of leading zeros are proposed for *coeff_token*, *total_zeros*, *run_before* decoding in ref. [3]–[10]. In the early work [6], sequential symbol matching process is used from shorter symbol to longer symbol. This scheme is not suitable for high-performance real-time applications because the long matching process time is needed for decoding a long symbol. In the ref. [3]–[5], Moon and Yu proposed some VLCT memory access scheme which can decode a symbol within limited cycles. The proposed architectures have some defects when VLCTs are implemented with a memory. It has unequal processing time depending on the length of the symbol and the memory has the length information of each symbol. Consequently, additional storage area is required. When memory architecture is used for VLCTs, the decoding results are generated in next cycle. So, next decoding step is not determined by skip condition within current decoding process.

To improve CAVLC decoder used memory for VLCT, various VLCT architectures using LUTs are proposed in ref. [2], [6], [8]. CAVLC decoding methods using LUTs require a couple of look-up table access, sequentially. So, they have long critical path and each element in LUT has symbol length information.

total_zeros decoder could be designed with a cognate method employed in *coeff_token* decoding. Because *total_zeros* symbol has similar syntax compared with *coeff_token* symbol. In ref. [9], Moon proposed *total_zeros* decoding method with simple address generation for memory access and some tables are removed with arithmetic decoding for reduced hardware (H/W). *Toal_zeros* decoder in Moon's work generates symbol length with simple arithmetic operations. Therefore, it has more reduced memory size than others.

Run_before decoder has small VLCTs compared with *coeff_token* and *total_zeros* VLCTs. Table removal scheme by using arithmetic operations has been proposed and moon proposed full arithmetic decoding method for *run_before* decoder in ref. [11]. And Yu and Lee proposed multi-symbol *run_before* decoder to decode a couple of *run_before* symbol in ref. [8], [15]. Yu proposed multi-symbol *run_before* by using large memory that contains 86 elements and each element consists of contiguous two symbols. Proposed multi-symbol *run_before* decoder has increased throughput but occupied larger area. Lee adopted Moon's *run_before* decoder and proposed symbol length prediction scheme based on the probability for multi-symbol *run_before* decoder. So, proposed multi-symbol *run_before* decoder achieved 2-fold increase in throughput. Multi-symbol *run_before* decoder generates a couple of level indexes with *run_before* values to update output array.

To improve previous work, we propose a low-cost LUT which is not contain symbol length information. The proposed LUT is accessed by generated address by using the number of leading zeros and bit-stream. We also propose a high throughput multi-symbol *run_before* decoder and a novel flush unit.

3. Proposed CAVLC Decoder

The block diagram of the proposed CAVLC decoder is shown in Fig. 2. The proposed CAVLC decoder shares a *leading_zeros* detector which was existent in *level*,

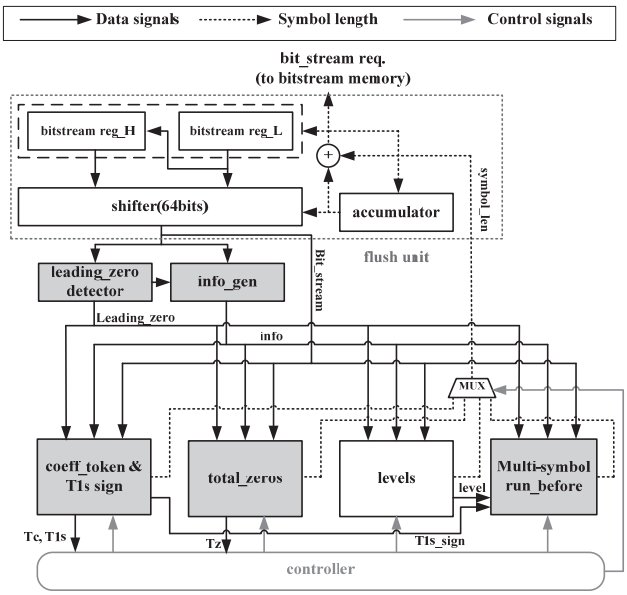


Fig. 2 Block diagram of proposed CAVLC decoder.

run_before, and *coeff_token* sub-decoder, separately, in the previous works [6]–[8]. The proposed CAVLC decoder reduces H/W cost and allows other sub-decoders to use the values of *leading_zeros* in each decoding process. And *info* generator produces the suffix bit-stream by shifting current bit-stream as *leading_zeros* + 1. With shared *leading_zeros* detector and *info* generator, proposed CAVLC decoder has suitable architecture for detecting Exp-Golomb code that consists of *leading_zeros*, '1', and suffix(*info*).

We also proposed a novel flush unit to renew bit-stream registers without additional cycles. The proposed flush unit calculates so far consumed bit-stream length within existing decoding process. In the previous works [1], [3], the flush unit consists of accumulator, shifter, and 32 bits two registers. It can't generate bit-stream request signal (*bit_stream_req*) within current symbol decoding process. So, if the length of bit-stream for residual block is over thirty two, an additional cycle is required for bit-stream register renewal. On the other hand, proposed flush unit has additional adder to generate bit-stream request signal with current symbol length (*symbol_len*) and consumed bit-stream length until last symbol decoding. If the consumed bit-stream length is over thirty two, bit-stream request signal is generated in current decoding process. So, in the following step, the decoding process is accomplished with refreshed bit-stream.

coeff_token & $T1s\_sign$ and *total_zeros* decoders are designed with combinational logic, but *level* and *run_before* decoder should store current state values because they are self-dependent decoding process. In Fig. 2, the shade blocks mean modified blocks that are explained in the following sub-clauses.

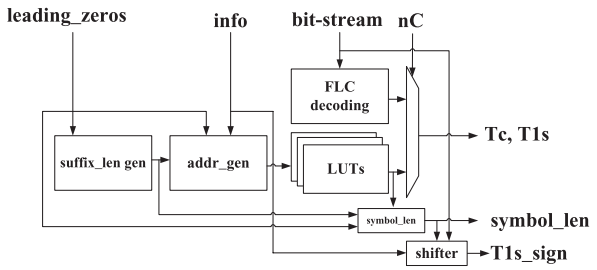


Fig. 3 Block diagram of proposed *coeff_token* & $T1s\_sign$ decoder.

3.1 Proposed *coeff_token* & $T1s\_sign$ Decoder

The proposed *coeff_token* & $T1s$ *sign* decoder depicted in Fig. 3 has four decoding steps to decode total coefficient, trailing ones and the sign of trailing ones. The first step (*suffix_len* decoding) and second step (*addr_gen*) are used for address generation for VLCTs access. In the third step, the symbol length is decoded with leading_zeros, suffix length, and decoded elements of LUTs. Finally, the signs of trailing ones are decoded in the fourth step.

In the first step, suffix length of coeff_token symbol is decoded with logical operation. The operations to calculate suffix length (*suffix_len*) dedicate in Eq. (1)~(3).

$$\textit{suffix\_len} = \begin{cases} 3 & \textit{if } (\&\{a,b,c',d'\} \mid \&\{a,b',c\} \mid \&\{a,b',c,d'\}) \\ 2 & \text{otherwise} \end{cases} \quad \textit{for VLCT0} \tag{1}$$

$$\textit{suffix\_len} = \begin{cases} 3 & \textit{if } (\&\{a,b',c'\} \mid \&\{a',b,c,d\} \mid \\ & \&\{a',b',c,d'\} \mid \&\{a,b',c,d'\} \\ 2 & \text{otherwise} \end{cases} \quad \textit{for VLCT1} \tag{2}$$

$$\textit{suffix\_len} = \begin{cases} 3 & \textit{if } (\&\{a',b'\} \mid \&\{a',b,c'\} \mid \&\{a',b,d'\}) \\ 2 & \text{otherwise} \end{cases} \quad \textit{for VLCT2} \tag{3}$$

In Eq. (1)~(3), & and | indicate bit-wise AND and OR operation, respectively. And brace is used as in Verilog HDL syntax; signals in the brace are concatenated. The small letters, a, b, c, and d, means MSB to LSB of 4-bits width *leading_zeros* signal. Complement of small letters that are small letters with single quotation mark is inversed signal with NOT function. The *suffix_len* has two when *nC* is equal to -1 that is the fixed length code (FLC) decoding.

In second step, address for LUT access is decoded by using the number of leading zeros, suffix bit-stream (*info*) and suffix length which is acquired in previous step. The address decoding operations for VLCT0, VLCT1, VLCT2, and chroma DC are depicted in Eq. (4)~(7), respectively. After address decoding, the decoded address is adjusted depending on suffix length decoded in step 1. If *suffix_len* is three, MSB bit of suffix bit-stream (*info*[0]) is inversed and then added with address value to make final address. If not,

address value acquired in Eq. (4)~(7) is enforced to final address.

$$addr\_tmp0 = \begin{cases} 18 & if\ (\&\{a,b,c,d'\}) \\ leading\_zeros & \text{otherwise} \end{cases}$$

$$addr\_tmp = \begin{cases} 9+(leading\_zeros-9)\ll 1 & if\ ((\&\{a,b',c\})|(\&\{a,b,c'\})) \\ addr\_tmp0 & \text{otherwise} \end{cases} \quad (4)$$

$$addr\_tmp0 = \begin{cases} leading\_zeros & if\ ((\&\{a',b',c'\})|(\&\{a',b',d'\})) \\ leading\_zeros+1 & \text{otherwise} \end{cases} \quad (5)$$

$$addr\_tmp1 = \begin{cases} 17 & if\ (\&\{a,b,c',d'\}) \\ 8+((leading\_zeros-7)\ll 1) & \text{otherwise} \end{cases}$$

$$addr\_tmp = \begin{cases} addr\_tmp1 & if\ (a) \\ addr\_tmp0 & \text{otherwise} \end{cases}$$

$$addr\_tmp = \begin{cases} leading\_zeros+7 & if\ (a) \\ leading\_zeros \ll 1 & \text{otherwise} \end{cases} \quad (6)$$

$$addr\_tmp = \begin{cases} 7 & if\ (a) \\ leading\_zeros & \text{otherwise} \end{cases} \quad (7)$$

Where '≪' means left shifting operation. The proposed address decoder which is used for *coeff_token* VLCT decoding is showed in Fig. 4.

In this paper, we store the four VLCTs in four look-up table. Four elements constituted with Tc and $T1s$ are inserted in a row. Among the elements decoded with address, *coeff_token* (Tc and $T1s$) are made a final decision with the suffix bit-stream (*info*). The suffix bit-stream is used for sub-address to select one element among four elements in a row. A *sub_addr* is selected among suffix bit-stream depending on the value of *suffix_len*. If *suffix_len* is three, second and third bits (*info*[1 : 2]) in suffix bit-stream generated in *info* generator are chosen as *sub_addr* and if not, the first and second bits (*info*[0 : 1]) are selected as *sub_addr*. If valid suffix bit-stream length is shorter than two, the adjacent element is copied to decode a correct element regardless of invalid suffix bit-stream. If suffix length is three, eight elements are stored in two consecutive rows of a look-up table. Because of regularity, intuitive *coeff_token* decoding can be done. Because of duplicated elements, there are some inefficient uses of proposed LUTs shown in Table 1~4 but we can reduce look-up table size about 30 % because symbol length information of each element is not contained compared with ref. [8]. First number of each element in the table is total coefficient and the second one is the number of trailing ones.

In the third step, the symbol length decoding is processed. When we access proposed LUTs, we get four elements which are used to select the number of total coefficient and the number of trailing ones with *addr*. To find the length of valid suffix, we compare the four pairs like followings. There are two comparisons. First one is comparison of two elements which *sub_addr* value is '10' and '01'. And second one is comparison of upper two elements which *sub_addr* is '11' and '10' if the first bit of *sub_addr* (*sub_addr*[0]) is '1'. Otherwise, lower two elements are used for second comparison. The comparison results have '1' if the chosen two values are same. The results have '0' when the selected two pairs are different. The two compared results are defined as compare0 and compare1 in Eq. (8).

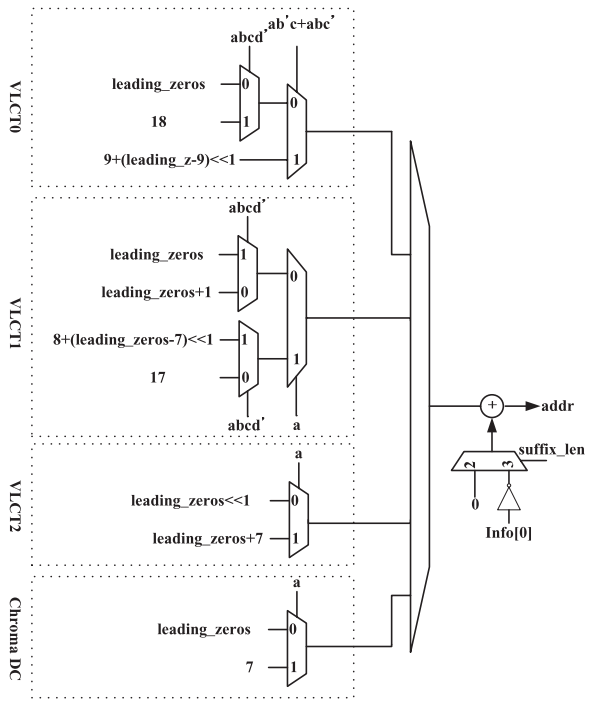


Fig. 4 Proposed address generator for *coeff_token* LUT decoding.

Table 1 Proposed VLCT0 look-up table for *coeff_token* decoding.

Addr	The Num. of leading zeros	*sub_addr*			
		11	10	01	00
0	0	[0,0]	[0,0]	[0,0]	[0,0]
1	1	[1,1]	[1,1]	[1,1]	[1,1]
2	2	[2,2]	[2,2]	[2,2]	[2,2]
3	3	[3,3]	[3,3]	[1,0]	[2,1]
4	4	[4,3]	[4,3]	[3,2]	[5,3]
5	5	[2,0]	[3,1]	[4,2]	[6,3]
6	6	[3,0]	[4,1]	[5,2]	[7,3]
7	7	[4,0]	[5,1]	[6,2]	[8,3]
8	8	[5,0]	[6,1]	[7,2]	[9,3]
9	9	[6,0]	[7,1]	[8,2]	[10,3]
10	9	[7,0]	[8,1]	[9,2]	[8,0]
11	10	[9,0]	[9,1]	[10,2]	[11,3]
12	10	[10,0]	[10,1]	[11,2]	[12,3]
13	11	[11,0]	[11,1]	[12,2]	[13,3]
14	11	[12,0]	[12,1]	[13,2]	[14,3]
15	12	[13,0]	[14,1]	[14,2]	[15,3]
16	12	[14,0]	[15,1]	[15,2]	[16,3]
17	13	[15,0]	[16,1]	[16,2]	[16,0]
18	14	[13,1]	[13,1]	[13,1]	[13,1]

Table 2 Proposed VLCT1 look-up table for *coeff_token* decoding.

Addr	The Num. of leading zeros	*sub_addr*			
		11	10	01	00
0	0	[0,0]	[0,0]	[1,1]	[1,1]
1	1	[2,2]	[2,2]	[3,3]	[4,3]
2	2	[2,1]	[2,1]	[5,3]	[5,3]
3	2	[1,0]	[3,1]	[3,2]	[6,3]
4	3	[2,0]	[4,1]	[4,2]	[7,3]
5	4	[3,0]	[5,1]	[5,2]	[8,3]
6	5	[4,0]	[6,1]	[6,2]	[5,0]
7	6	[6,0]	[7,1]	[7,2]	[9,3]
8	7	[7,0]	[8,1]	[8,2]	[10,3]
9	7	[8,0]	[9,1]	[9,2]	[11,3]
10	8	[9,0]	[10,1]	[10,2]	[12,3]
11	8	[10,0]	[11,1]	[11,2]	[11,0]
12	9	[12,0]	[12,1]	[12,2]	[13,3]
13	9	[13,0]	[13,1]	[13,2]	[14,3]
14	10	[14,0]	[14,0]	[14,2]	[14,2]
15	10	[14,1]	[15,2]	[15,0]	[15,1]
16	11	[16,0]	[16,1]	[16,2]	[16,3]
17	12	[15,3]	[15,3]	[15,3]	[15,3]

Table 3 Proposed VLCT2 look-up table for *coeff_token* decoding.

Addr	The Num. of leading zeros	*sub_addr*			
		11	10	01	00
0	0	[0,0]	[1,1]	[2,2]	[3,3]
1	0	[4,3]	[5,3]	[6,3]	[7,3]
2	1	[2,1]	[3,2]	[8,3]	[3,1]
3	1	[4,2]	[4,1]	[5,2]	[5,1]
4	2	[1,0]	[6,1]	[6,2]	[9,3]
5	2	[2,0]	[7,1]	[7,2]	[3,0]
6	3	[4,0]	[8,1]	[8,2]	[10,3]
7	3	[5,0]	[9,2]	[6,0]	[7,0]
8	4	[8,0]	[9,1]	[10,2]	[11,3]
9	4	[9,0]	[10,1]	[11,2]	[12,3]
10	5	[10,0]	[11,1]	[12,2]	[13,3]
11	5	[11,0]	[12,1]	[13,2]	[12,0]
12	6	[13,1]	[13,1]	[13,0]	[14,1]
13	6	[14,2]	[14,3]	[14,0]	[15,1]
14	7	[15,2]	[15,3]	[15,0]	[16,1]
15	8	[16,2]	[16,2]	[16,3]	[16,3]
16	9	[16,0]	[16,0]	[16,0]	[16,0]

$$symbol\_len = \begin{array}{l} leading\_zeros + 1 + suffix\_len \\ -compare0 - compare1 + T1s \end{array} \quad (8)$$

With proposed *coeff_token* decoding flow, Tc, $T1s$ and symbol length could be obtained, however, there is a irregular symbol ($nC = -1$. bit-stream = 000_0000...) which is remarked with shade elements in the Table 4. To get correct results, the exception is treated by additional logics that modify *leading_zeros* to be used for *symbol_len* calculation.

Finally, $T1s\_sign$ decoding to transfer the sign of trailing ones to level register file is carried out. The symbol

Table 4 Proposed chroma DC look-up table for *coeff_token* decoding.

Addr	The Num. of leading zeros	*sub_addr*			
		11	10	01	00
0	0	[1,1]	[1,1]	[1,1]	[1,1]
1	1	[0,0]	[0,0]	[0,0]	[0,0]
2	2	[2,2]	[2,2]	[2,2]	[2,2]
3	3	[1,0]	[2,1]	[3,3]	[2,0]
4	4	[3,0]	[3,0]	[4,0]	[4,0]
5	5	[3,1]	[3,1]	[3,2]	[3,2]
6	6	[4,1]	[4,1]	[4,2]	[4,2]
7	>6	[4,3]	[4,3]	[4,3]	[4,3]

position of the sign of the trailing ones ($T1s\_sign$) is started at *suffix_len* - *compare*0 - *compare*1 in the suffix bit-stream (*info*). The following bits as $T1s$ are used for trailing ones sign decoding and parsed to be stored in level register file.

Fixed Length Code (FLC) decoding can be defined with arithmetic function in contrast with other VLC decoding. The symbols of FLCT (for $8 \le nC$) have six bit fixed length. FLC decoding is defined as Eq. (9).

$$\begin{cases} Tc = 0 \\ T1s = 0 \end{cases} \quad \text{if } (leading\_zeros == 4 \;\&\; bs[0] == 1)$$
$$\begin{cases} Tc = bs[0:3] + 1 \\ T1s = bs[4:5] \end{cases} \quad \text{otherwise} \quad (9)$$

Where *bs* means current valid bit-stream generated form the 64 bits shifter and the numbers in a square bracket are the bit's position in the bit-stream used for Tc and $T1s$ calculation. In the following equations, the *bs* indicates valid bit-stream parsed from flush unit, continuously.

3.2 *Level* Decoder

Level symbols are decoded with not VLCTs but arithmetic decoding procedure. To decoding *level* symbol, maximum length of level symbol should be analyzed, precisely. The length of level symbol is defined depending on supported profile. If the profile is baseline, main, or extended profile, *prefix* of *level* symbol is below fifteen. And the length of suffix is *prefix* − 3 or less. Therefore, the maximum length of *level* symbol is twenty eight bits. In the other profiles, the length of prefix is 11+*bit_depth* and below. The *bit_depth* is eight more and fourteen less. Level decoder is designed to decode the symbol that its length is less than twenty eight because proposed CAVLC decoder supports up to main profile. Level decoding flow is described in Table 5. In the Table 5, the '∼' means bit-wise not gate operation.

There is a conditional exception in the level decoding procedure that is depicted in Sect. 2. It is applied for enhanced compression rate in H.264/AVC encoding. In the CAVLC encoding, the first non-trailing ones level has reduced absolute value by one when the number of trailing ones is less than three. If $T1s$ is less than three, then the first non-trailing ones level is incremented by one if negative, then decremented by one if positive so that the first

Table 5 level symbol decoding flow.

```
threshold = [3 6 12 24 48 96]
level_prefix = leading_zeros
// suffixLength initialization
if (level_cnt == 0 & Tc > 10 & T1s < 3)// first level decoding
    suffixLength = 1
else if (level_cnt == 0)
    suffixLength = 0
else
    suffixLength = suffixLength
// levelSuffixSize decoding
if(level_prefix == 14 & suffixLength == 0)
    levelSuffixSize = 4
else if (level_prefix == 15)
    levelSuffixSize = level_prefix – 3
else
    levelSuffixSize = suffixLength
// levelCode decoding
level_suffix = info[0:levelsuffixSize-1]
levelCode = level_prefix<<suffixLength + level_suffix
if (levelSuffixSize >= 15 & suffixLength == 0)
    levelCode = levelCode + 15
if (level_cnt == 0 & T1s <3) // first level decoding
    levelCode = levelCode + 2
// level decoding
level = levelCode[0] ? (~levelCode)/2 : levelCode/2 +1
//level_abs decoding
level_abs = level >= 0 ? level : ~level + 1
// suffixLength update
if (suffixLength == 0)
   suffixLength = 1;
if (suffixLength < 6 & level_abs > threshold[suffixLength])
    suffixLength = suffixLength + 1
```

non-trailing ones level closed to zero. By contrast, this exception is expressed by conditional sentence which checks *level_cnt* and $T1s$ in the level decoding process. If *level_cnt* is zero and $T1s$ is less than three, *levelCode* is incremented by 2. As a result, the last non-trailing ones level has incremented absolute value.

In the level decoding flow, the *suffixLength* decided in previous level decoding process is used in current level decoding. So, level decoding is self-dependent decoding process and it is implemented by sequential logic.

3.3 Proposed *total_zeros* Decoder

Proposed *total_zeros* decoder has similar LUTs designed same ways which is used for LUTs in *coeff_token* decoder. Suffix length of *total_zeros* symbol is less than two. So, address adjustment used in second step of *coeff_token* decoding is not required. An identical symbol length decoding method is used in *coeff_token* decoding. But *total_zeros* decoding requires additional decoding process because there are a number of zero sequence symbols which can't be decoded by proposed symbol length decoding method. So, we calculate maximum length of zero sequence before address decoding with Eq. (10) proposed in ref. [9].

$$\max\_zero\_len = \begin{cases} 5 + j - k + 4i & for\ m = 1 \\ 6 & for\ m = 1 \end{cases} \tag{10}$$

In Eq. (10), i is $Tc/8$. k is Tc-6 when i is equal to one and otherwise, k is Tc. m is $k/4$ and $j = (k-2)/4$. / indicate the integer division with truncation of the result toward zero. If obtained *leading_zeros* is larger than maximum length of zero sequence, *leading_zeros* is replaced with *max_zero_len*. In ref. [9], Moon proposed hybrid decoding method. The *total_zeros* decoder used simple arithmetic operation to decode *total_zeros* symbol when Tc is 1, 14, or 15. We extend the arithmetic operation. Proposed arithmetic operation is interpreted in Eq. (11) which is applicable when Tc is equal to 2. We can reduce about 11 % look-up table compare with ref. [9].

$$\begin{aligned} &temp = bs[0:2] + 1 \\ &\begin{cases} total\_zeros =\sim bs[0:2] \\ symbol\_len = 3 \end{cases} \quad if\ (temp \le 4) \\ &\begin{cases} total\_zeros = 10 - bs[0:3] \\ symbol\_len = 4 \end{cases} \quad elseif\ (temp \le 2) \\ &\begin{cases} total\_zeros = bs[3]?13 - bs[0{:}4]{:}14 - bs[0{:}5] \\ symbol\_len = bs[3]?5 : 6 \end{cases} \quad \text{otherwise} \end{aligned} \tag{11}$$

We also replace look-up table for 2×2 chroma residual data to arithmetic decoding operation like Eq. (12).

$$\begin{aligned} &\begin{cases} total\_zeros = 4 - Tc & if\ (leading\_zeros \ge 4 - Tc) \\ total\_zeros = leading\_zeros & \text{otherwise} \end{cases} \\ &\begin{cases} symbol\_len = total\_zeros & if\ (total\_zeros \ge 4 - Tc) \\ symbol\_len = total\_zeros + 1 & \text{otherwise} \end{cases} \end{aligned} \tag{12}$$

3.4 Proposed *run_before* Decoder

The length of almost *run_before* symbols is shorter than three and *run_before* decoder has small H/W size than other sub-decoders. Multi-symbol run_before decoder is studied in various literatures based on the feature of *run_before* symbol [8], [15]. But there are huge increases of H/W size in contrast with its enhanced throughput. Yu proposed separated *run_before* tables for multi-symbol *run_before* decoder in ref. [15]. The multi-symbol *run_before* decoder decodes two *run_before* symbols when *zeroLeft* is less than six. The table has possible combination of two continuous *run_before* symbols. It has enhanced throughput but the table size increased exponentially. In ref. [11], Moon proposed full arithmetic decoding based *run_before* decoder and then Lee improved the Moon's work for efficient H/W implementation. Lee also proposed multi-symbol *run_before* decoder based on statistical analysis between the length of *run_before* symbol and *zeroLeft* in ref. [8]. The

multi-symbol *run_before* decoder predicts the length of current and next *run_before* symbol with current *zeroLeft*. It can decode three *run_before* symbols in a cycle when the length of decoded symbols is equal to prediction results. But, the multi-symbol *run_before* decoder doesn't have high prediction success ratio.

The proposed *run_before* decoder has reduced H/W size by increasing regularity of *run_before* decoding operation. We remove three adders, three 2-input MUX with additional 5 gates. *run_before* decoding is divided four case that are *zeroLeft* = (1 and 2), *zeroLeft* = (3, 4, and 5), *zeroLeft* = 6, and *zeroLeft* > 6 listed in Eq. (13), (14), (15), and (16). Final *run_before* is selected by *zeroLeft*. In Eq. (15), '∧' means exclusive OR gate operation.

$$run\_tmp0 = \sim bs[0]$$
$$run\_tmp1 = bs[0]\ ?\ run\_tmp0 : zeroLeft - bs[0:1]$$
$$run\_before = zeroLeft_{LSB}? \sim bs[0] : run\_tmp1 \quad (13)$$
$$run\_before = zeroLeft + \sim bs[0:1] \le 6?$$
$$\sim bs[0:1] : zeroLeft - bs[0:2] \quad (14)$$
$$run\_tmp0 = bs[2]\ ?\ bs[0:2] : bs[0:2] + 2$$
$$run\_tmp1 = bs[0:1] == 3\ ?\ 0 : bs[0:2] + 1$$
$$run\_before = \wedge bs[0:1]\ ?\ run\_tmp0 : run\_tmp1 \quad (15)$$
$$run\_before = |bs[0:2]? \sim bs[0:2] : leading\_zeros + 4 \quad (16)$$

The proposed multi-symbol *run_before* decoder shown in Fig. 6 uses the proposed *run_before* decode which is depicted in Fig. 5. The proposed CAVLC decoder offers multi-symbol *run_before* decoder the number of leading zeros.

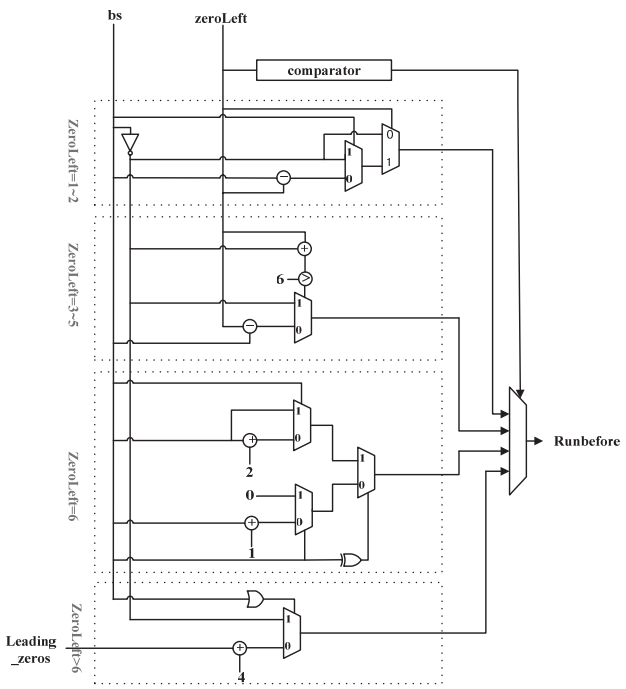


Fig. 5 Proposed run_before decoder.

The number of leading zeros and *zeroLeft* are used to decode the length of the 1st *run_before* symbol before the 1st *run_before* results are generated. As a result, we can decode correct two *run_before* symbols when there are remained *run_before* symbols to be decoded. Each *run_before* decoder in multi-symbol *run_before* decoder is executed when previous *zeroLeft* is larger than zero and the number of *run_before* execution is less than *Tc*-1. The 3rd *run_before* decoder has correct result if the symbol length generated in the 2nd *run_before* decoder is identical to prediction results in the Table 6.

The conditions described in the previous paragraph are checked in a *run_before* controller. The controller generates 3 bit-width *run_en* signal to enable to write the level coefficients stored in level register file with level index generated with *run_before* decoders and level index register. The level index register stores prior wrote *level_index* in the last *run_before* decoding to make level indexes of following level coefficients. The level indexs are used as addresses to store level coefficients in output registers.

The controller, also, generates selection signal for 4 MUXs positioned on right side of run_before_d blocks. If the results generated in a run_before_d block are not equal

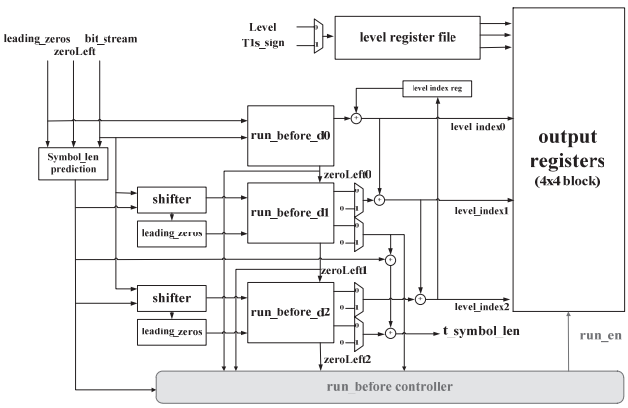


Fig. 6 Proposed multi-symbol run_before decoder.

Table 6 First run_before symbol length pre-decoding and second run_before symbol length prediction.

Case	*ZeroLeft*	*Leading_*zeros	1st symbol Length	2nd symbol Length
1	1	0	1	1
2	2	0	1	1
3		otherwise	2	1
4	3	all	2	2
5	4	≥2	3	1
6		otherwise	2	2
7	5	0	2	2
8		otherwise	3	2
9	6	AND(bs[0],bs[1]) = 1	2	3
10		else	3	2
11	≥6	≤1	3	3
12		otherwise	*leading_zeros* + 1	2

to prediction results or there are no more symbol to be decoded, the *run_before* and *symbol_len* of run_before_d block are ignored and demmy value '0' are selected to get a correct *level_index* for level indexes and total symbol length that is calculated by summation of each *symbol_len* generated in *run_before* decoder blocks.

Finally, multi-symbol *run_before* decoder generates total length of decoded *run_before* symbols (*t_symbol_len*) and selects valid *zeroLeft* by using the number of decoded *run_before* symbols. We also proposed a simple symbol length decoding operation for the 2nd and the 3rd *run_before* decoders in Eq. (17).

$$len\_tmp = zeroLeft + run\_before$$
$$symbol\_len = \begin{cases} 1 & \text{for } len\_tmp \le 2 \\ 2 & \text{for } len\_tmp \le 6 \\ 3 & \text{for } zeroLeft > 6 \,\&\, (|code[0:2] = 1) \\ leading\_zeros + 1 & \text{for otherwise} \end{cases} \tag{17}$$

4. Experimental Results

Test sequences used for experimental results are offered by ITU [16]. Input and output data of CAVLC decoder are generated by using JM 16.0 for the functional verifications. We eliminate misjudgments according to subtle difference of test sequence by using public test sequence.

4.1 Performance Evaluation of Proposed Multi-Symbol *run_before* Decoder

Run_before symbol length pre-decoding is categorized to twelve cases. It is used for first *run_before* symbol length pre-decoding with *zeroLeft* and leading zeros. Second *run_before* symbol length is examined with eight general purpose test sequences offered by ITU when second *run_before* symbol decoding is performed and there are remained *run_before* symbols to be decoded in the third *run_before* decoder. The eight test sequence which has various quantization parameter (Qp), and level are used to avoid local prediction results. In the Table 6, AND($bs[0], bs[1]$) means AND gate operation of the first and second bit in current bit-stream. The researched symbol length results are depicted in Fig. 7.

In Fig. 7, the numbers on the left and right of the underscore in the x label signify the case in the Table 6 and second *run_before* symbol length, respectively. The maximum value of occurrence rate is eight because the executed number of third *run_before* symbol decoding is normalized to one in each test sequence. Second *run_before* symbol length prediction has more than 50 % success ratio in the other cases except case 12. Especially, in case 1, 3, and 5, the prediction success ratio has 100 %.

We verified the performance of the proposed multi-symbol *run_before* decoder with Oh and Lee's work. We decoded eight test sequences and compare the total consumed cycles which are used in *run_before* decoding. The results are shown in Table 7. The *run_before* symbol has a low proportion of total CAVLC decoding in low-quality test sequences like BA3_SVA_C and BA2_SVA_F. In these case, the double-symbol *run_before* decoder using the *run_before* symbol length pre-decoding is comparable with Lee's multi-symbol *run_before* decoder which has three *run_before* decoding path. When there are more level decoding and *run_before* decoding than low-quality test sequence, Lee's multi-symbol *run_before* decoder consumes lower decoding

Table 7 Processing cycle comparison of various run_before decoders.

Sequence	ref. [2] (Oh)	ref. [8] (Lee)	Proposed double	Proposed Multi
BA3_SVA_C	1995	1435	1429	1347
BA2_SVA_F	42183	29250	29509	27175
CVFC1_Sony_C	247455	143478	152454	130388
NLMQ1_JVC_C	285316	133890	157389	121641
NLMQ2_JVC_C	226185	109738	124862	97276

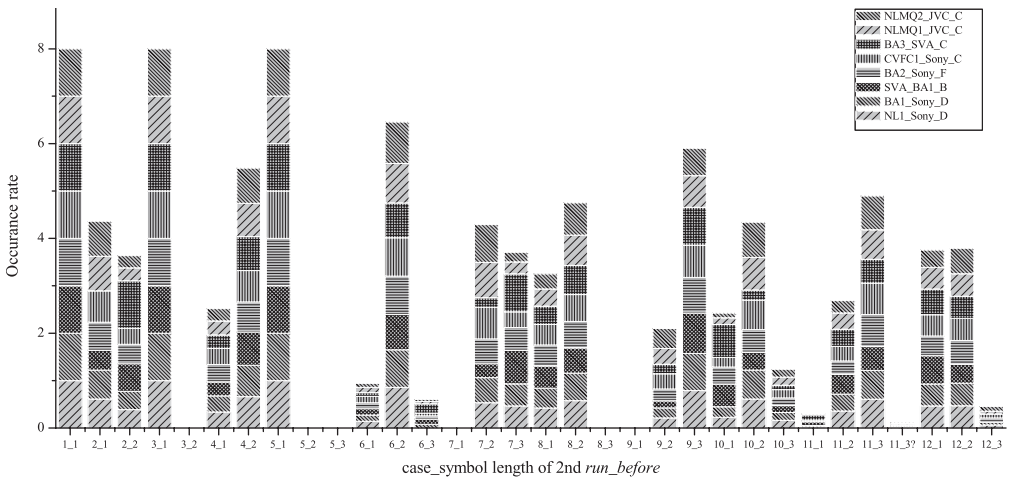


Fig. 7 Occurrence rates of the second *run_before* symbol length.

cycles than the double symbol *run_before* decoder. Because Lee's work utilizes 3 path *run_before* decoding and statistics of *run_before* symbol length for multi-symbol *run_before* decoding. But, proposed multi-symbol *run_before* decoder has about 7∼11 % reduced processing cycles compared with Lee's work based on *run_before* symbol length pre-decoding and high accuracy *run_before* symbol length prediction.

4.2 Performance Evaluation of Proposed CAVLC Decoder

Proposed CAVLC decoder requires much lower cycles for CAVLC decoding due to *coeff_token* & $T1s\_sign$ decoding, high-throughput multi-symbol *run_before* decoder, a novel flush unit, and look-up table based symbol decoding compared with chang and alle's work. In additionally, we can increase the throughput about 4∼9 % compared with Lee's work by using *run_before* symbol length pre-decoding and high accuracy *run_before* symbol length prediction. For the throughput comparison, we calculate the average cycles per macroblock that is conventionally used in CAVLC throughput comparison. The throughput comparison is shown in Table 8 and '-' used to denote the missing data.

4.3 Implementation Results

We designed proposed algorithm with Matlab and the input and output data used in software simulations are generated with JM reference software ver. 16.0. After software simulation, the proposed algorithms are designed in Verilog HDL and than synthesized with Megnachip 0.18 technology library.

Table 8 Throughput comparison of various CAVLC decoder (Average cycles per macroblock).

Sequence	ref. [6] (Chang)	ref. [14] (Alle)	ref. [8] (Lee)	Proposed
NL1_Sony_D	187	142	64	61.6
BA1_Sony_D	187	144	64	61.6
SAV_BA1_B	140	120	36	35.8
BA2_Sony_F	-	-	7.4	7.4
BA3_SVA_C	-	-	3.8	3.7
CVFC_1_Sony_C	-	-	72.1	69.2
NLMQ1_JVC_C	-	-	269.3	247.2
NLMQ2_JVC_C	-	-	176.8	161.7

Table 9 Implementation results of various CAVLC decoders.

	Tech.	Freq. (Mhz)	Gate Count	Memory Size (bits)	Power (mW)
Chang (ref. [14])	UMC 0.13	125	9943	1152	-
Alle (ref. [15])	UMC 0.18	125	17586	5120	-
Yu (ref. [4])	0.18	125	13192	0	-
Lee (ref. [8])	TSMC 0.13	125	15602	0	5.28
Proposed	Megna 0.18	125	11964	0	5.26

Novel look-up tables of proposed CAVLC decoder don't have symbol length information of each symbol. So, it archived 30 % look-up table size reduction compare with Lee's double look-up table architecture which consists of group table and symbol table in ref. [8] as described in Table 9. And we designed hybrid *total_zeros* decoder with extended arithmetic decoding and proposed look-up table. Because of extended simple arithmetic decoding, we can reduce 10 % look-up table area compared with ref. [9]. We also optimize the *run_before* decoder. Therefore, proposed CAVLC decoder can be implemented with 23 % reduced H/W size compared with ref. [8].

5. Conclusion

CAVLC decoding in H.264/AVC has an important role to get high coding efficiency. But the decoding flow should be implemented sequentially because of variable length characteristics of CAVLC symbol. It is not acceptable for high quality and real-time video sequence decoding. To overcome these defects of CALVC decoding, we proposed a low-cost and high-throughput CAVLC decoder.

For the H/W reduction, we proposed simple symbol length generation method for *coeff_token*, *total_zeros*, and *run_before* decoder. In the total_zeros decoder, we also extend the arithmetic decoding for the high efficient hybrid decoding and smaller LUT size. In the previous works, there is individual leading zeros detector to detect *leading_zeros* of Exp-Golomb code in each sub-decoder. On the other hand, we designed CAVLC decoder with a shared leading_zeros detector to reduce H/W size and provide sub-decoders with additional information.

In addition, the proposed CAVLC decoder has increased throughput because of proposed multi-symbol *run_before* decoder and a novel flush unit which does not require additional cycles for bit-stream buffer renewal. As a result, the proposed CAVLC decoder can process high quality video sequence effectively because of its high throughput. It is also applicable for portable media applications because of its cost-efficient design.

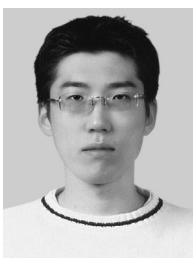
Acknowledgments

This work was supported by IC Design Education Center (IDEC).

References

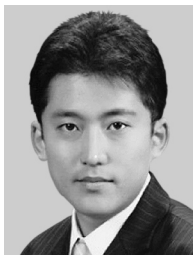
[1] D. Wu, W. Gao, M. Hu, and Z. Ji, "A VLSI architecture design of CAVLC decoder," 5th International Conference on ASIC, vol.2, pp.962–965, 2003.
[2] M. Oh, W. Lee, and J. Kim, "Design of high speed CAVLC decoder for H.264/AVC," IEEE Workshop on Signal Processing Systems, pp.325–330, 2007.

[3] Y.H. Moon, I.K. Eom, and S.W. Ha, "An improved coeff_token variable length decoding method for low power design of H.264/AVC CAVLC decoder," IEEE International Conference on Image Processing, pp.2840–2843, 2008.
[4] Y.-X. Yu, G.-M. Du, D.-L. Zhang, Y.-K. Song, and M.-L. Gao, "An improved decoding method of coeff_token element for H.264 CAVLC decoder," International Conference on Anti-counterfeiting, Security, and Identification in Communication, pp.524–538, 2009.
[5] Y.H. Moon, "A new coeff-token decoding method with efficient memory access in H.264/AVC video coding standard," IEEE Trans. Circuits Syst. Video Technol., vol.17, no.6, pp.729–736, June 2007.
[6] H.-C. Chang, C.-C. Lin, and J.-I. Guo, "A novel low-cost high-performance VLSI architecture for MPEG-4 AVC/H.264 CAVLC decoding," IEEE International Symposium on Circuits and Systems, vol.6, pp.6110–6113, 2005.
[7] B.-S. Choi and J.-Y. Lee, "Implementation of area efficient H.264/AVC CAVLC decoder," IEEE International Conference on IC Design and Technology, pp.135–138, 2009.
[8] G.G. Lee, C.-C. Lo, Y.-C. Chen, H.-Y. Lin, and M.-J. Wang, "High-throughput low-cost VLSI architecture for AVC/H.264 CAVLC decoding," IET Image Processing, vol.4, no.2, pp.81–91, 2010.
[9] Y.H. Moon, "An advanced Total_zeros decoding method based on new memory architecture in H.264/AVC CAVLC," IEEE Trans. Circuits Syst. Video Technol., vol.18, no.9, pp.1312–1317, Sept. 2008.
[10] B.-J. Shieh, Y.-S. Lee, and C.-Y. Lee, "A new approach of group-based VLC codec system with full table programmability," IEEE Trans. Circuits Syst. Video Technol., vol.11, no.2, pp.210–221, Feb. 2001.
[11] Y.H. Moon, G.Y. Kim, and J.H. Kim, "An efficient decoding of CAVLC in H.264/AVC video coding strandard," IEEE Trans. Consum. Electron., vol.51. no.3, pp.933–938, Aug. 2005.
[12] Y.-N. Wen, G.-L. Wu, S.-J. Chen, and Y.-H. Hu, "Multiple-symbol prallel CAvLC decoder for H.264/AVC," IEEE Asia Pacific Conference on Circuits and Systems, pp.1240–1243, 2006.
[13] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro, "H.264/AVC baseline profile decoder complexity analysis," IEEE Trans. Circuits Syst. Video Technol., vol.13, no.7, pp.704–716, 2003.
[14] M. Alle, J. Biswas, and S.K. Nandy, "High performance VLSI architecture design for H.264 CAVLC decoder," Proc. Int. Conf. on Application-Specific Systems, Architectures and Processors, pp.317–322, 2006.
[15] G.-S. Yu and T.-S. Chang, "A zero-skipping multi-symbol CAVLC decoder for MPEG-4 AVC/H.264," Proc. Int. Symposium on ISCAS 2006, pp.5583–5586, 2006.
[16] http://wftp3.itu.int/av-arch/jvt-site/draft_conformance/

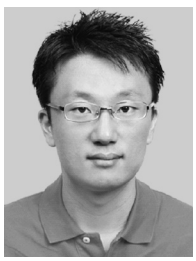


Kyu-Yeul Wang was born in Incheon, South Korea in 1981. He received B.S. and M.S. degrees in the School of Information and Telecommunication Engineering from INHA University, Korea in 2006 and 2008, respectively. He is currently a Ph.D. candidate in the School of Information and Telecommunication Engineering in INHA University. His current research interests are wireless communication, MPEG-4, WLAN, and VLSI & SoC design.

Byung-Soo Kim was born in Seoul, Korea in 1984. He received B.S. and M.S. degrees in the School of Information and Telecommunication Engineering from INHA University, Korea in 2006 and 2008, respectively. He is currently a Ph.D. candidate in the School of Information and Telecommunication Engineering in INHA University. His research interests are wireless communication hardware design, JPEG2000 codec design and VLSI & SoC design.



Sang-Seol Lee was born in Chung-Ju, South Korea in 1981. He received B.S. and M.S. degrees in the School of Information and Telecommunication Engineering from INHA University, Korea in 2007 and 2009, respectively. He is currently a Ph.D. candidate in the School of Information and Telecommunication Engineering in INHA University. His current research interests are wireless communication, DRM, PCMCIA, UWB and VLSI & SoC design.



Dong-Sun Kim was born in Incheon, Korea in 1972. He received B.S. and M.S. degrees in the School of Electronics and Electrical Engineering in 1997 and 1999, respectively, from INHA University, Incheon, Korea. In 2005, he received a Ph.D. degree from the School of Information and Telecommunication Engineering of INHA University, Incheon, Korea. Since 1999, he has been with the Korea Electronics Technology Institute (KETI), Gyeonggi-do, Korea and working on R&D at the DxB·Communication Convergence Research Center, where he currently is a senior researcher and team leader. He is a member of IEEE. His research interests are in the areas of wireless/wired communication systems, wireless sensor networks, VLSI & SoC design, multimedia codec design, computer architecture, and embedded system design.

Duck-Jin Chung was born in Korea on Feb. 8, 1948. He received a B.S. degree in electrical engineering from Seoul National University, Korea in 1970, and a M.S. degree from Utah State University in 1984. In 1988, he received a Ph.D. degree from the University of Utah. He is currently a professor at the Sschool of Information and Communication Engineering of INHA University, Incheon, Korea. He is a member of IEEE. His research interests are VLSI & SoC design, computer architecture, and embedded system design.