Published in IET Image Processing
Received on 2nd April 2008
Revised on 8th October 2009
doi: 10.1049/iet-ipr.2008.0064

ISSN 1751-9659

High-throughput low-cost VLSI architecture for AVC/H.264 CAVLC decoding

G.G. Lee C.-C. Lo Y.-C. Chen H.-Y. Lin M.-J. Wang

Department of Electrical Engineering, National Cheng Kung University, Tainan City, Taiwan
E-mail: nimokos@vlsilab.ee.ncku.edu.tw

Abstract: This study develops a low-cost very-large-scale-integration (VLSI) hardware architecture for entropy coding with increased throughput using the statistical properties of context-based adaptive variable-length coding (CAVLC) in AVC/H.264. Statistical analyses show that better symbol length prediction was achieved by breaking the recursive dependency among codewords for the multi-symbol decoder implementation. The proposed CAVLC decoder easily meets the real-time requirements for high definition (HD) (1920 × 1088) applications. The clock speed is only 13 MHz under the best case scenario.

1 Introduction

The field of information theory was pioneered by Shannon in 1948 [1]. Information theory applies the thermodynamic principle of entropy to a data source as a measure of its information content, which can be expressed by

$$H = -\sum_{i=1}^{m} p_i \log_b (p_i) \tag{1}$$

where m is the number of unique symbols in the alphabet, p_i is the probability of occurrence of each symbol and b is the base of the number system used in the coded representation of the source data. Hereafter, this base is assumed to be 2 as the binary number system is commonly applied to digital information storage and transmission. In information theory, if the symbols are distinct, then the average number of bits needed to encode them is always bounded by their entropy. Hence, the objective of lossless data compression is to approach entropy as closely as possible. Variable-length coding (VLC) is an entropy coding method which provides lossless compression for video and images, such as MPEG-2 [2], JPEG [3] and AVC/H.264 [4] standards. VLC attempts to approach entropy by assigning codeword lengths based on the probability model of the input symbols. For frequently occurring symbols, VLC uses short codeword representations; longer codewords are assigned to symbols that occur less frequently. Therefore the average codeword length is close to entropy.

AVC/H.264 was defined by the Joint Video Team (JVT) in 2003 [4]. It differs from previous standards in that it applies content adaptation in context-based adaptive variable-length coding (CAVLC) as entropy coding. Instead of using {RUN, Level, EOB (end of block)} to encode each non-zero coefficient in MPEG-2 and JPEG standards, CAVLC uses *Coeff_Token*, *TrailingOnes Sign*, *Level*, *TotalZero*, and *RunBefore* symbols to handle the coefficients, whose value is zero or ±1, more efficiently. CAVLC also considers the spatial relationship between the current and previous coded symbols. The spatial correlation significantly improves the compression ratio. However, increasing the compression ratio leads to computational complexity.

As mentioned above, *Coeff_Token*, *TrailingOnes Sign*, *Level*, *TotalZero*, and *RunBefore* are the five major decoding procedures in CAVLC. CAVLC decodes one basic block (usually a 4 × 4 block) by performing the above decoding procedures sequentially. Fig. 1*a* shows an example of CAVLC decoding and the values of related symbols; Fig. 1*b* shows the procedure of CAVLC decoding. CAVLC decodes two symbols, *TotalCoeff* and *TrailingOnes*, after *Coeff_Token* decoding. *TotalCoeff* is the number of total coefficients in a basic block. *TrailingOnes* is the number of coefficients whose absolute values are equal to one at the end of the zig-zag scan order. Note that if there are more than three ±1's, only the last three are treated as *TrailingOnes*; the rest are treated as normal level values. In the *TrailingOnes Sign*

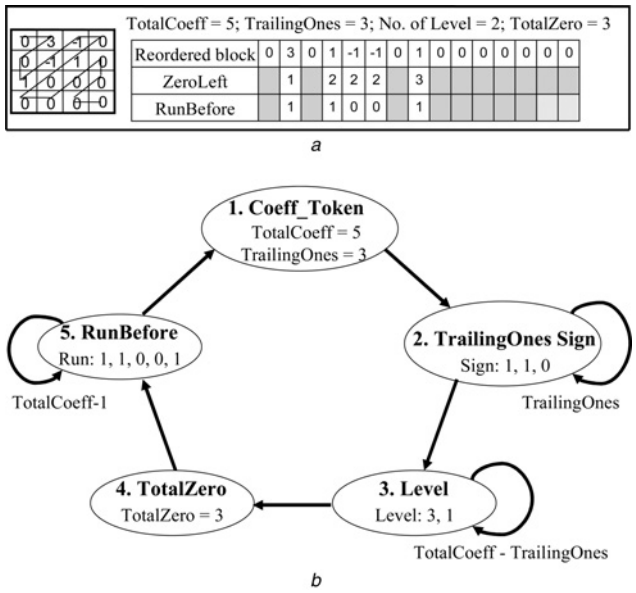


Figure 1 *Symbol definition and decoding flow in CAVLC*

a Example of CAVLC and values of related symbols
b State diagram of CAVLC decoder

decoding stage, the decoder identifies the signs of the *TrailingOnes*. In this step, every symbol is coded with 1 bit. The non-zero coefficients in a 4 × 4 block which do not belong to *TrailingOnes* are then decoded in the *Level* decoding stage. *Level* coding uses seven VLC tables, from VLC0 to VLC6. The last two procedures, *TotalZero* and *RunBefore*, are required to determine the positions of non-zero coefficients to construct a 4 × 4 block. *TotalZero* is the number of all zeros preceding the last non-zero coefficients in the reordered block. There are tables for the 4 × 4 AC block and the 2 × 2 DC block, respectively. In the *RunBefore* phase, the decoder processes the number of zeros preceding each non-zero coefficients in reverse order and inserts zeros between two nCs. Seven tables are used in the *RunBefore* decoding procedure. These tables are separated according to *ZeroLeft*, which denotes the number of remaining zeros before the current coefficient. After *RunBefore* decoding, the number of zeros between non-zero coefficients is known. A 4 × 4 residual block is thus completely reconstructed and passed to the next decoding procedure (inverse quantisation). In our work, we employed a codeword length prediction scheme and two extra *RunBefore* decoders in multi-symbol decoding to achieve multi-symbol decoding. In addition, combining the *Coeff_Token* and *TrailingOnes Sign* processes decreases the total processing time. The experimental results show that the proposed CAVLC decoder can be implemented for Level 4.1 1080HD (1920 × 1088) video applications with a clock speed of under 13 MHz. Our design also has a smaller silicon area than those of other works.

The rest of this paper is organised as follows. Section 2 reviews related studies on CAVLC decoders. Section 3 presents the proposed design. Section 4 discusses the verification and implementation results. The conclusion is given in Section 5.

2 Literature review

There have been many studies on CAVLC decoder design [5–10]. These designs use technology similar to that proposed in previous VLC designs [11–19]. Xue *et al.* [5] developed a software-based decoder and provided a fast group-based decoding algorithm. Based on Lei's work [6], Wu *et al.* [7] proposed a decoder which can decode one symbol per cycle. Moon *et al.* [8] used partial arithmetic operations on *Coeff_Token* and *RunBefore* decoders to reduce memory access times. Although Moon's design has good performance for low bit-rate streams, the method still depends on look-up tables. Kim *et al.* [9] presented an arithmetic decoding operation to replace all tables to overcome conventional table look-up method drawbacks, such as high power consumption and slow decoding time. Chang *et al.* [10] focused on table partitioning according to the codeword length. During the decoding procedure, tables that store short codewords are accessed ahead of tables that store long codewords. Although this hierarchical table accessing can limit power consumption, there is a large performance penalty for symbols with long codewords. All the above-mentioned designs only considered the matching scheme between codeword and symbol.

Since the CAVLC decoder is the first stage of the AVC/H.264 decoder, its throughput significantly affects the overall performance of the video decoder. Thus, a high-throughput architecture for CAVLC decoding is necessary, especially for high resolution applications. To achieve this goal, Alle's design [20], which is based on Chang's architecture [10], adopts an early termination scheme to avoid unnecessary decoding procedures for CAVLC to avoid pipeline stalling to further improve performance. Tseng and Hsieh [21] proposed a pattern-search method, which reconstructs a 4 × 4 (or 2 × 2) block directly without going through CAVLD. However, if there is no pattern match in the table, the block has to be reconstructed using the normal CAVLD decoding procedure. Oh *et al.* [22] developed a lookup table which combines *Coeff_Token* and *TrailingOnes Sign*. Therefore *Coeff_Token* and *TrailingOnes Sign* can be decoded at the same time.

Other high-throughput VLD architectures have been proposed with multi-symbol decoders [23–26]. Chen *et al.* [23] built multi-symbol VLD tables for *Level* and *RunBefore* directly, so that CAVLD can decode consecutive multiple *Level* or *RunBefore* symbols in one cycle. However, the details of their design were not described. Yu's decoder [24] can decode multiple symbols in the *TrailingOnes Sign* and *RunBefore* stages. It divides the *RunBefore* table into six parts to achieve two-symbol decoding if the consecutive symbols use different *ZeroLeft* tables. Nikara *et al.* [25] proposed a parallel multi-symbol decoding scheme for an MPEG-2 decoder. Parallel codeword length detection is

doi: 10.1049/iet-ipr.2008.0064

performed so that multiple symbols can be obtained from the symbol table. Tsai and Fang [26] proposed parallel operations in *Level* decoding to generate two *Level* in one cycle. Tsai's design also detected a series of *RunBefore* symbols whose values are zero. Intuitively, multi-symbol decoding can be achieved using the following two methods:

1. A multi-symbol VLD table can be adopted, but the table size increases exponentially. For example, if there are M possible symbols to be decoded, M^2 possible joint symbols are required for a two-symbol decoding engine.

2. Tables for multi-symbol decoding can be cascaded. Although the table size increases linearly, the long critical path degrades performance.

In previous designs, the codeword dependency was neglected. This dependency significantly limits the throughput of the decoder. The present study develops a codeword length prediction scheme using statistical approaches to decode successive codewords concurrently. A high-throughput rate for the CAVLC decoder is achieved with only limited hardware overhead. A memory-free decoder is also proposed. Finally, the arithmetic operation algorithm is applied to *Level* and *RunBefore* for better decoding performance.

3 Proposed CAVLC decoder

As mentioned previously, there are five major symbols that need to be processed in the CAVLC decoding process. The decoding flows of the example in Fig. 1*a* for Wu's architecture [7] and for the proposed CAVLC design are shown in Figs. 2*a* and *b*, respectively. In Wu's work, each symbol requires one cycle for decoding. Therefore the required number of cycles for *Coeff_Token*, *TrailingOnes Sign*, *Level*, *TotalZero* and *RunBefore* decoding are 1, 3, 2, 1 and 4, respectively. Moreover, five extra cycles are required to insert zeros for the five coefficients to reconstruct a 4 × 4 block. In our work, *TrailingOnes Sign* decoding is reduced to one cycle and combined with *Coeff_Token* decoding. The two symbols can thus be decoded in one cycle. A multi-symbol decoder is then

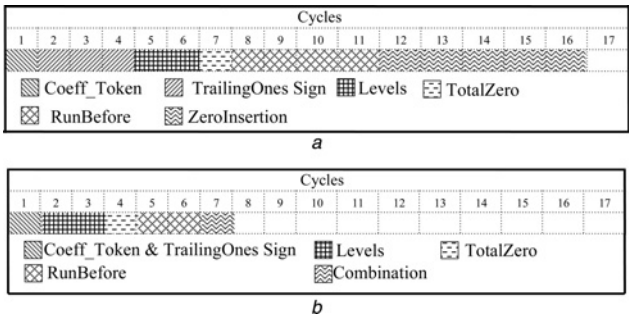


Figure 2 *Example of CAVLC decoding flow*

a Wu's architecture [7]
b Proposed design

employed in *RunBefore* decoding, shortening the processing cycles from 4 to 2. Moreover, the proposed rapid zero insertion scheme is processed in parallel with *RunBefore* decoding so only one cycle is required to reconstruct a 4 × 4 block after all symbols are decoded. As a result, the number of cycles required for processing a 4 × 4 block is reduced from 16 to 7.

Fig. 3 shows the block diagram of the proposed CAVLC decoder, which contains *Coeff_Token* & *TrailingOnes Sign*, *Level*, *TotalZero* and multi-symbol *RunBefore* decoders. Output registers are used to construct a 4 × 4 block according to the decoded symbols. The following subsections provide detailed descriptions of the symbol decoders. Throughout this paper, we use the notation I_i to indicate the ith bit of input bitstream I in binary representation and $(I_i \ldots I_{j+1}I_j)$ to represent a segment of I from the ith bit to the jth bit.

3.1 Coeff_Token and TrailingOnes sign decoder

To decode the symbols *TotalCoeff* and *TrailingOnes* in the *Coeff_Token* step, one fixed-length and four variable-length look-up tables are used depending on the variable nC, specified in AVC/H.264 standard, in the left and upper decoded blocks. The fixed-length code table can be realised using the following numerical operations

$$\begin{cases} \text{TotalCoeff} = (I_5 I_4 I_3 I_2) + 1 \\ \text{TrailingOnes} = (I_1 I_0) \end{cases} \quad (2)$$

The variable-length code decoders are implemented using the group-based structural VLC table. The VLC table is first classified into several groups according to codeword prefix zeros and codeword length, as shown in Table 1. The prefix zeros and '1' serve as the first reference for look-up table searching; the remaining bits act as the designated reference for searching. After the index of the look-up table is identified, the symbols *TotalCoeff* and *TrailingOnes* can be chosen by the multiplexers.

A memory-free *Coeff_Token* decoder which uses arithmetic operations and multiplexers is proposed. Although the

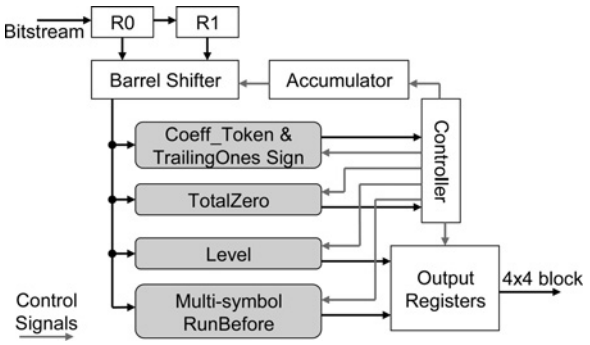


Figure 3 *Block diagram of the proposed CAVLC decoder*

Table 1 Grouping of part of a *Coeff_Token* table

T1s	*TotalCoeff*	Codewords	nC	Prefix code	Remainder	Group	Base	Offset	Length
...	...	...	...	...	...	...	...	...	...
3	9	00000000100	0	000000001	00x	10	56	0–1	11
2	7	00000000101	0	000000001	01x	10	56	2–3	11
1	6	00000000110	0	000000001	10x	10	56	4–5	11
0	5	00000000111	0	000000001	11x	10	56	6–7	11
...	...	...	...	...	...	...	...	...	...

proposed decoder relies on group-based processing, the decoded symbols are directly available without address generation after matching. Moreover, the number of decoding cycles is reduced by combining the decoding of *Coeff_Token* and *TrailingOnes Sign* together, as shown in Fig. 4. The bitstream is input to leading zero detection (LZD). The result is used to determine the base and offset address for the symbol table to derive *TotalCoeff* and *TrailingOnes*. Then, the length of bits consumed by the *Coeff_Token* decoder is passed along to the *TrailingOnes Sign* decoder to decode the *TrailingOnes Sign* symbols from the correct position of the bitstream. The total length of used bits, including the codeword length of *Coeff_Token* and *TrailingOnes Sign*, is transferred to the next stage. As a result, the number of cycles required to decode *Coeff_Token* and *TrailingOnes Sign* can be reduced from 4 (first to fourth cycle in Fig. 2*a*) to 1 (first cycle in Fig. 2*b*). Since the data in the group table and symbol table are fixed and small, we implement the two tables using random logic instead of storing them in memory.

3.2 Level and TotalZero decoders

Because the longest codeword length of *Level* is 28 bits, implementing the *Level* decoder with look-up tables, or even the group-based scheme used in the *Coeff_Token* decoder, is inefficient. The *Level* decoding process can be simplified by using arithmetic operations as specified in AVC/H.264. Each *Level* contains two parts: *LevelPrefix* and *LevelSuffix*. *LevelPrefix* represents how many bits are equal to 0 before the first non-zero bit. This means that *LevelPrefix* can be derived from a leading zero detector. *LevelSuffix* is the variable-length code that follows

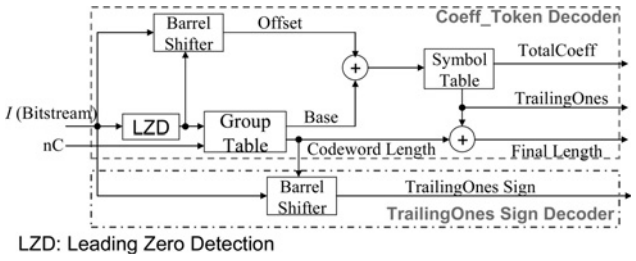


Figure 4 *Architecture of Coeff_Token and TrailingOnes Sign decoders*

LevelPrefix. *LevelSuffix* can be determined using *LevelSuffixSize*, which is the codeword length of *LevelSuffix*. The variable *LevelSuffixSize* can be derived from the variable *SuffixLength*, which is specified in the AVC/H.264 standard, as shown in (3)

$$LevelSuffixSize = \begin{cases} 4, & \text{if}(Level\text{Prefix} = 14) \text{ and } (SuffixLength = 0) \\ 12, & \text{else if}(Level\text{Prefix} = 15) \\ SuffixLength, & \text{otherwise} \end{cases} \tag{3}$$

Therefore the syntax element *LevelSuffix* is decoded as (4). In (4), $(I_{(LevelSuffixSize-1)} \ldots I_2 I_1 I_0)$ is the bitstream I with a length of *LevelSuffixSize* bits

$$LevelSuffix = \begin{cases} 0, \\ \text{if}(LevelSuffixSize = 0) \\ (I_{(LevelSuffixSize-1)} \ldots I_2 I_1 I_0), & \text{otherwise} \end{cases} \tag{4}$$

When *LevelPrefix* and *LevelSuffix* are known, the magnitude of *Level*, called *LevelCode*, can be decoded by the following pseudo code:

```
0  while (i < TotalCoeff − TrailingOnes)
1  {//Get the magnitude of coefficient
2  LevelCode = (Level Prefix ≪ SuffixLength) + LevelSuffix
3  if((LevelPrefix == 15) && (SuffixLength == 0))
4  LevelCode = LevelCode + 15;   //Getlevel
5  if(LevelCode[0] == 0) Level [i] = (LevelCode + 2) ≫ 1;
//positive level
6  else Level [i] = (−LevelCode − 1) ≫ 1;   //negative level
7  //Update suffixLength
8  SuffixLength = (LevelCode > (3 ≪ (SuffixLength − 1)))?
SuffixLength + 1: SuffixLength;
9  i = i + 1;}
```


doi: 10.1049/iet-ipr.2008.0064

We know that there are (*TotalCoeff*−*TrailingOnes*) *Level* decodings (Line 0). Each *LevelCode* is derived from the equation in Line 2. 15 is added to *LevelCode* if *LevelPrefix* = 15 and *SuffixLength* = 0 (Line 3–4). The *i*th *Level* is derived after determining its signed value (Line 5–6). *SuffixLength* must be updated for the next *Level* decoding. Therefore the table look-up method is not necessary because *Level* can be calculated using arithmetic operations.

As for *TotalZero* decoding, a group-based approach, as used in *Coeff_Token* decoding, can be applied. However, since each table of *TotalZero* is very small for 4 × 4 or 2 × 2 blocks, the decoder is implemented with combinational logic, rather than look-up tables, for simplicity.

3.3 RunBefore decoder

RunBefore represents the number of zeros between two non-zero coefficients. Here, we modified the arithmetic operations, proposed in Moon's work [8], to reduce the required number of operations and thus lower the hardware cost

$$RunBefore = \begin{cases} 1 - I_1, & \text{for } ((I_1 I_0) = 0) \text{ and } (ZeroLeft = 2) \\ I_1, & \text{otherwise} \end{cases} \tag{5}$$

$$RunBefore = \begin{cases} 5 - (I_2 I_1 I_0), & \text{for } (Zeroleft = 5) \text{ and } ((I_2 I_1 I_0) \le 2) \\ 4, & \text{for } (Zeroleft = 4) \text{ and } ((I_2 I_1 I_0) = 0) \\ 3 - (I_2 I_1 I_0), & \text{otherwise} \end{cases} \tag{6}$$

$$RunBefore = \begin{cases} 0, & \text{for } (I_2 I_1 I_0) = 6 \\ (I_2 I_1 I_0) + 1, & \text{for } (I_2 I_1 I_0) < 2 \\ 4, & \text{for } (I_2 I_1 I_0) = 2 \\ (I_2 I_1 I_0), & \text{otherwise} \end{cases} \tag{7}$$

$$RunBefore = \begin{cases} 7 - (I_2 I_1 I_0), & \text{for } (I_2 I_1 I_0) > 0 \\ 4 + \mathrm{m}, & \text{otherwise} \end{cases} \tag{8}$$

The optimised results for *ZeroLeft* = (1 and 2), *ZeroLeft* = (3, 4 and 5), *ZeroLeft* = 6, and *ZeroLeft* > 6 are listed in (5), (6), (7) and (8), respectively. In (8), *m* is the number of leading zeros in the bitstream. With these equations, the decoder was implemented using arithmetic operations, with five fewer addition operations and one fewer shift operation compared with those required for Moon's work. The proposed architecture of the *RunBefore* decoder is shown in Fig. 5. In the figure, the comparator in the sub-decoder for *ZeroLeft* = 6 is used to select one of four possible values according to (7). The *RunBefore* decoder is partitioned into four parts based on (5), (6), (7) and (8), respectively. Finally, *RunBefore* is decoded from the output of these sub-decoders according to the value of *ZeroLeft*.

In the overall decoding procedure, only *TrailingOnes Sign*, *Level* and *RunBefore* appear more than once in a basic block. As mentioned before, the implementation of *TrailingOnes*

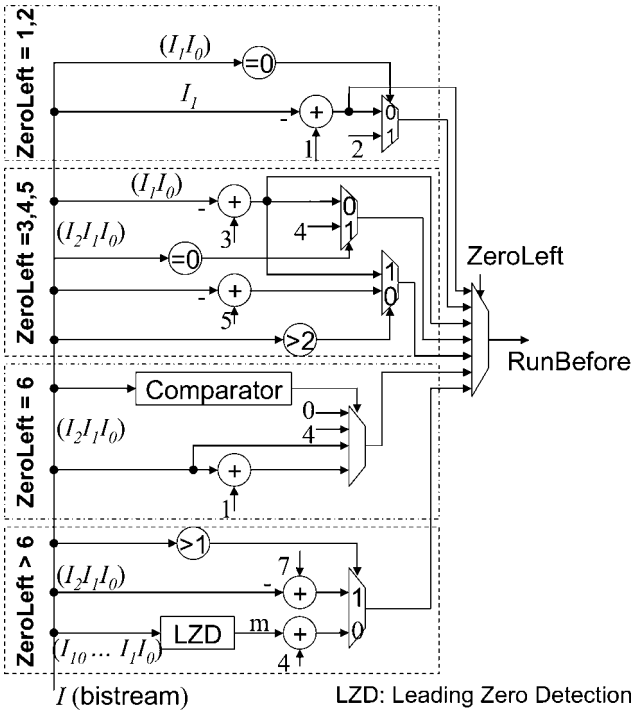
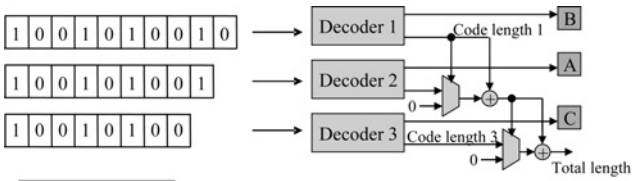


Figure 5 *Architecture of RunBefore decoder*

Sign decoding is relatively easy; all symbols are decoded in one cycle. *Level* decoding consists of a series of arithmetic operations requiring long critical data paths, and hence is not suitable for multi-symbol decoding. In order to increase the throughput of the entire CAVLC decoder, we analysed the statistics of codewords and broke the dependency of the bitstream to design a multi-symbol decoder for *RunBefore*.

3.4 Multi-symbol RunBefore decoder

The main goal of this multi-symbol decoder is to break the recursive dependency between codewords. Each possible codeword is put into its own decoder and the codeword length is returned for the next symbol decoding. Because this algorithm removes the codeword length dependency, a parallel architecture can be used to improve decoding throughput. Fig. 6 shows an example of the multi-symbol decoder and its performance for the input bitstream of the second and third decoders. In Fig. 6, three parallel



Symbol	Code
A	1
B	01
C	001
D	000

Decoder 1 : Output valid, Total Length = 2

Decoder 2 : Output invalid, Total Length = 2

Decoder 3 : Output valid, Total Length = 5

Figure 6 *Example of multi-symbol decoder*

decoders are fed the single-shifted bitstream, but only two symbols are decoded. Because the codeword length of the first symbol (symbol B) is 2 bits, the output of the second decoder is invalid since the input bitstream of the second decoder is 1-bit shifted. This example shows that the input bitstream for the second and third decoders is important. Therefore instead of feeding the parallel decoder with the single-shifted bitstream, a codeword length predictor is employed in our multi-symbol decoder to further improve the decoding efficiency.

We used statistical analyses to find the relationship of codeword length between successive symbols before the *RunBefore* decoder was designed. The statistical analyses were performed on a randomly chosen image sequence; the results are shown in Fig. 7. In this figure, the *X* axis represents all possible combinations of codeword length and *Y* axis represents the number of occurrences during the decoding of 300 I-frames. On the *X* axis, the numbers on the left and right of the hyphen in the label signify the first and second codeword length of the *RunBefore* symbol, respectively. For instance, 2–1 means that the codeword lengths of two successive *RunBefore* symbols are 2 bits and 1 bit, respectively. In Fig. 7, when QP is equal to 20, the number of occurrences of 1–1 is about 280 000. This is a much higher number of occurrences than those for 1–2 and 2–1. Although the variation of QP influences the number of occurrences, successive codeword lengths have a similar distribution regardless of the value of QP.

Conditions such as sequence resolution, sequence content and the quantisation parameter were also taken into consideration. Fig. 8 shows the statistical results of *RunBefore* decoding. As shown in Fig. 8*a*, all sequences (akiyo, foreman, hall, news, silent, coastguard, container, stefan and table_tennis) have similar distributions despite the sequence content. The reason can be deduced from Table 9–10 in the AVC/H.264 standard [4]. When *ZeroLeft* is equal to 2, the codeword length is 1 or 2. Since a short codeword length implies a high frequency of appearance, 1–1 (highlighted by dotted blocks) is the most probable codeword length pair. Unfortunately, the codeword length distribution is always the same, except in the case of high QP (QP = 36). However, 1–1 is still on average about 27% of the entire codeword length pairs. When QP increases, the number of *RunBefore* symbols decreases, as shown in Fig. 8*a*. Therefore although the different distributions at high QP increase the ratio of prediction error, the effect can be neglected when compared with other values of QP. Figs. 8*b* and *c* show the statistics of codeword length for *ZeroLeft* equal to 3 and 4, respectively. Based on the same statistical analysis, 2–2

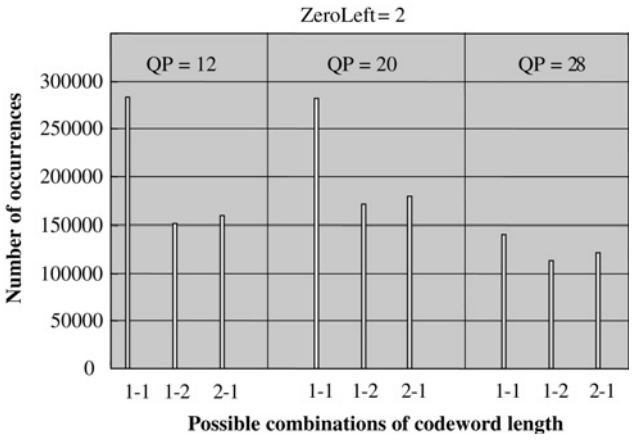


Figure 7 *Statistical results of RunBefore for mobile sequence with 4 : 2 : 0 CIF and 300 I-frames*

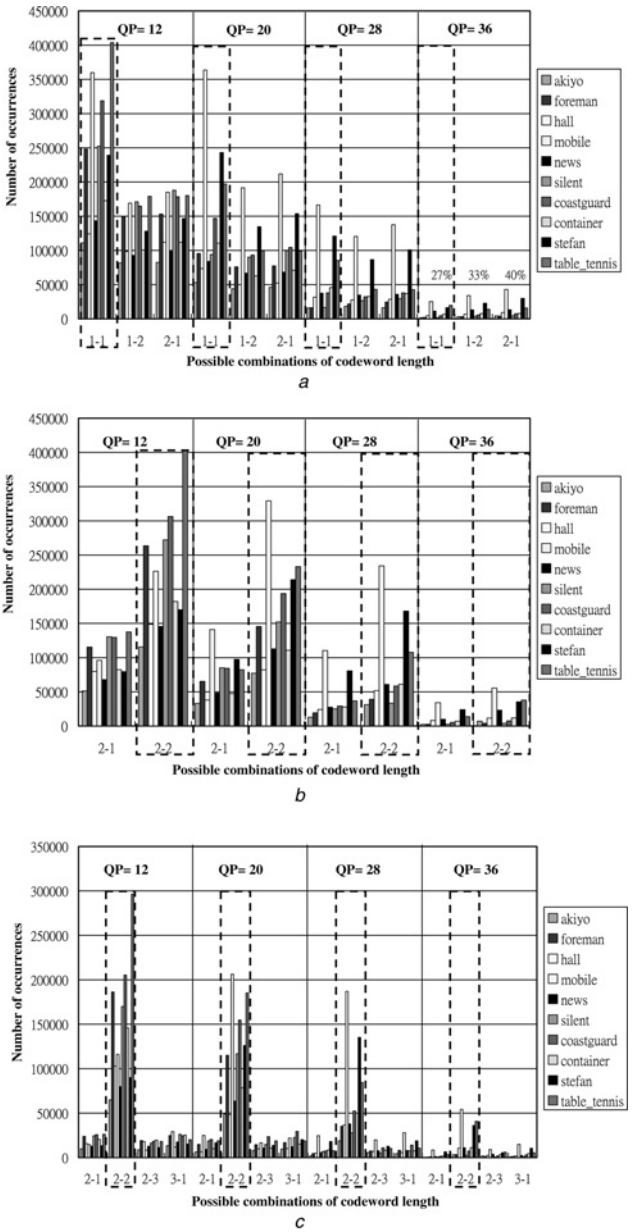


Figure 8 *Some statistical results of RunBefore (4 : 2 : 0 CIF and 300 I-frames)*

a Statistics of codeword length when *ZeroLeft* = 2
b Statistics of codeword length when *ZeroLeft* = 3
c Statistics of codeword length when *ZeroLeft* = 4

doi: 10.1049/iet-ipr.2008.0064

Table 2 Prediction table for *RunBefore*

ZeroLeft	Prediction	
	First codeword length	Second codeword length
1	1	1
2	1	1
3	2	2
4	2	2
5	3	2
6	3	2
else	3	3

(dotted blocks in the two figures) has a much higher number of occurrences than those of any other cases regardless of the sequence content and quantisation parameter.

From the histogram in Fig. 8, the variation of codeword length distribution only depends on *ZeroLeft*. This helps us implement the decoder in a more flexible manner for various *ZeroLeft* situations. The statistical data were analysed under various *ZeroLeft* situations; the results are shown in Table 2. An accurate codeword length prediction can be made before the next *RunBefore* decoding. Hence, we can break the recursive dependency among codewords to decode successive codewords concurrently and achieve a high throughput rate in the *RunBefore* decoder.

The proposed multi-symbol *RunBefore* decoder shown in Fig. 9 was designed using the codeword length prediction scheme shown in Table 2. The tradeoff between the number of decoders used and the length of the critical path needs to be considered. A good balance between the throughput and the clock speed should be sought. Since the critical data path of the entire CAVLC decoder is in the *Level* decoder (about 8 ns), we chose three decoders for the multi-symbol decoder. The critical path of a multi-symbol decoder is nearly the same as that of the *Level* decoder. The details of the tradeoff between the number of decoders and throughput are discussed in Section 4.1. In Fig. 9, the operations of the upper two barrel shifters are based on Table 2. The shifted bit-stream is passed along to

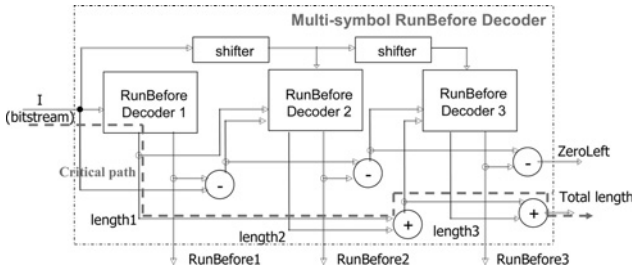


Figure 9 *Architecture of multi-symbol RunBefore decoder*

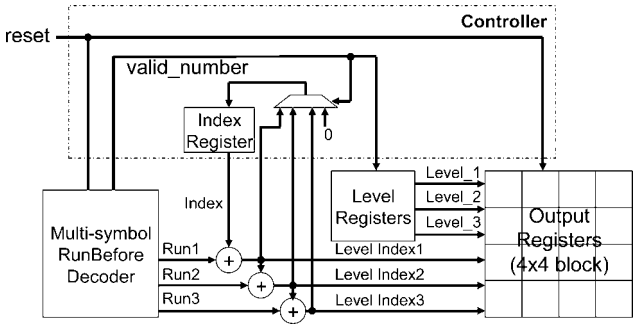


Figure 10 *Architecture of rapid zero insertion*

the next two *RunBefore* decoders and the decoded codeword length becomes the selection of the next sub-decoder. The controller detects the number of valid symbols using the total codeword length since the number of valid symbols is not fixed due to prediction error in our multi-symbol *RunBefore* decoder.

3.5 Rapid zero insertion

When *RunBefore* has been decoded, zeros have to be inserted between non-zero coefficients. Fig. 10 shows the architecture of rapid zero insertion. Rapid zero insertion can be achieved as follows. First, all coefficients in a 4 × 4 block (output registers in Fig. 10) are reset to zero before a block element is decoded. Then, all the values of decoded non-zero coefficients are stored in *Level* Registers before the *RunBefore* symbol is decoded. After decoding each *RunBefore* symbol, the *RunBefore* symbol is added to the previous index and the results are passed along to the next adder. The index denotes where the *Level* should be inserted. Non-zero coefficients which are stored in the *Level* registers are then stored in the output register according to the computed index. Since our *RunBefore* decoder provides multiple *RunBefore* symbols, *Level* registers in Fig. 10 must output one to three symbols according to the number of decoded *RunBefore* symbols. If the last *RunBefore* has been decoded, all remaining non-zero coefficients are simultaneously passed along to the corresponding register. As a result, zero insertion can be done in parallel with *RunBefore* decoding. Therefore only one cycle is required for final zero insertion after all *RunBefore* symbols have been decoded, as shown in Fig. 2*b*.

4 Experimental results

4.1 Performance evaluation of proposed multi-symbol RunBefore decoder

As mentioned in Section 3.4, we chose three decoders for the multi-symbol decoder. Table 3 shows some of our statistics for average *RunBefore* symbols in a 4 × 4 block. As shown in the table, the average number of *RunBefore* symbols is about three in a 4 × 4 block. This means that the design does not benefit greatly by using more than three *RunBefore* decoders. As a

Table 3 Average *RunBefore* symbols in a 4 × 4 block

Sequence	Foreman		Akiyo		News		Mobile	
	Luma	Chroma	Luma	Chroma	Luma	Chroma	Luma	Chroma
QP = 12	4.26	3.01	2.48	1.96	2.95	2.19	3.35	4.00
QP = 20	3.30	1.66	2.15	1.34	2.76	1.35	4.32	3.56
QP = 28	1.95	0.90	1.51	1.01	2.23	0.77	3.92	2.23

result, we designed a multi-symbol decoder for decoding *RunBefore* that can decode three symbols in one cycle.

The number of operational clock cycles reduction for various kinds of *RunBefore* decoder is tabulated in Table 4. Type I does not use any acceleration scheme, Type II uses a generic multi-symbol decoder as in Fig. 6 and Type III is the proposed multi-symbol *RunBefore* decoder. The three circuits were fed the same input data to evaluate the total number of decoding cycles for *RunBefore* symbol. As shown in Table 4, the performance of Type II is based on the percentage of 1-bit symbols, as shown in the last row. Because 1-bit symbols only occur if *ZeroLeft* is equal to 1 or 2, the percentage of 1-bit symbols decreases with increasing QP. This occurs because higher QP leads to more runs between non-zero coefficients. However, a large QP makes non-zero coefficients only appear at low frequency, which diminishes the probability of (*ZeroLeft* > 2). As a result, the percentage of 1-bit symbols may slightly increase, as shown in the last column in Table 4. However, the proposed design can achieve a 50% cycle reduction regardless of the value of QP. The throughput of the proposed *RunBefore* decoder is higher than those of other designs due to the accurate prediction in our decoding scheme. Moreover, Table 5 shows the number of *RunBefore* symbols and its percentage of total symbols. From Table 5, the *RunBefore* symbol is about 30% of the total symbols. As mentioned above, the proposed multi-symbol *RunBefore* decoder can reduce the number of processing cycles by half compared with single symbol decoding. This means that the proposed multi-symbol *RunBefore* decoder can improve CAVLC decoding by (30% × 50%) = 15%.

4.2 Functional verification and performance evaluation

To achieve maximum compression, coded data are normally sent in a continuous stream of bits without specific guard-bits assigned to separate two consecutive symbols. If an error occurs during decoding, it propagates to the decoding of the following codeword. This chain reaction crashes the decoding of the following symbols. Therefore a robust verification of the entropy decoder is important. First, the CAVLC decoder is partitioned into five major decoding sub-systems according to the output symbol type. Since each sub-decoder is data dependent, sub-block verification is important for later integration. Various input patterns are used, and the output responses are extracted for each sub-decoder from the Golden C model. These patterns can be used to verify RTL and gate-level models for each sub-system.

Table 4 Cycle reduction with various kinds of *RunBefore* decoder (4 : 2 : 0 CIF and 300 I-frames)

	Foreman							
	QP 12		QP 20		QP 28		QP 36	
	Cycles	Reduction	Cycles	Reduction	Cycles	Reduction	Cycles	Reduction
I	3 278 784	–	1 791 140	–	637 701	–	82 861	–
II	1 853 982	43.4%	1 222 939	31.72%	510 273	19.98%	64 370	22.31%
III	1 013 344	69.1%	663 156	62.97%	280 160	56.06%	41 320	50.13%
no. of 1-bit symbols	1 495 010 (45.5%)		649 138 (36.2%)		174 018 (27.3)		26 352 (31.8%)	

Table 5 Percentage of *RunBefore* symbol (4 : 2 : 0 CIF and 300 I-frames) in total number of symbols

	No. symbols (Foreman sequence)		
	QP 12	QP 20	QP 28
total	7 841 956	4 987 972	2 493 086
RunBefore	3 278 784 (43.8%)	1 791 140 (35.9%)	637 701 (25.6%)

IET Image Process., 2010, Vol. 4, Iss. 2, pp. 81–91
doi: 10.1049/iet-ipr.2008.0064

After each sub-block is verified, the design is verified after integration. Although there are only five sub-decoders in the CAVLC decoder, there are several exceptions in the data transfer of the decoder. Some sub-decoders may be repeated several times while some may remain idle during the decoding of a block element. In order to verify the entire CAVLC system, we generated corresponding test patterns based on the diagram of the CAVLC decoder shown in Fig. 1*b*. After fully testing the various kinds of test pattern, we used AVC/H.264 reference software to encode real sequences, including: akiyo, foreman, hall, mobile, news, silent, coastguard, container, stefan and table_tennis. The compressed data were then decoded to generate the test patterns for our design. The generated test patterns make the functional verification more complete.

The proposed design was implemented in Verilog HDL and synthesised with the TSMC 130 nm CMOS standard cell-based design library. The proposed decoder was compared with other published designs. For a fair comparison, we used the same bitstreams provided by JVT [27] and other common sequences encoded with various QP. Table 6 shows the average required cycles for decoding a macroblock. Since these designs do not provide all the simulation results of these sequences, '-' is used to denote missing data. Simulation results show that the number of processing cycles required for CAVLC decoding of the proposed design is much lower than those for the other designs. This is due to:

1. *Coeff_Token* and *TrailingOnes Sign* processes being combined.

2. Multi-symbol decoding on *RunBefore*, which neglects the codeword length dependency.

3. Rapid zero insertion schemes with *RunBefore* decoding.

However, the percentage of *RunBefore* symbols is no more than 50%. The number of required processing cycles was reduced by around 3-fold compared to Chang's design because the number of required processing cycles of each kind of symbol may require more than one cycle in Chang's work. In the proposed design, the decoding of *Coeff_Token* or *Level* only needs one cycle. For *TrailingOnes Sign* and *RunBefore* decoding, 1–3 symbols are decoded in a single cycle. It is possible that the processing cycles can be reduced by around 3-fold compared to Chang's design. The proposed architecture design reduces processing cycles by 23% compared to Yu's design with little area cost, because a codeword length prediction circuit and three *RunBefore* decoders are adopted in our design.

In the previous section, we used cycles per macroblock to estimate the decoder's performance. In order to make sure that the proposed design can meet the requirement of real-time video applications, we follow the Level limitations of the AVC/H.264 standard, which include bit-rate, frame size,

Table 6 Comparison of processing cycles per macroblock with related hardware

Sequence		Chang [10]	Alle [20]	Yu [24]	Proposed
NL1_Sony_D [23]		187	142	–	64
BA1_Sony_D [23]		187	144	–	64
SAV_BA1_B [23]		140	120	–	36
Stefan	QP 28	319	–	124	97
	QP 20	441	–	200	154
	QP 12	563	–	269	204
mobile	QP 28	395	–	174	135
	QP 20	570	–	279	211
	QP 12	704	–	353	264
news	QP 28	196	–	58	49
	QP 20	282	–	108	87
	QP 12	404	–	176	138
technology		UMC.13	UMC.13	0.18 μm	TSMC.13
frequency, MHz		125	125	125	125
gate count		9943	17 586	13 192	15 602
memory size, bits		1152	5120	0	0
power, mW		n/a	n/a	n/a	5.28

Table 7 Simulation results for HD sequences

Sequence	Blue_sky	Rolling	Riverbed	Traffic	Sunflower
level	4.1				
resolution	1920 × 1088				
bitrate (kbits/s)	50 000				
frame rate (frames/s)	30.1				
total frames	148				
total cycles	56 128 109	62 236 829	63 322 174	60 466 499	59 025 129
clock speed, MHz	11.37	12.61	12.83	12.256	11.964

Table 8 Performance and gate count comparison of CAVLC decoders

	Chang [10]	Alle [20]	Oh [22]	Yu [24]	Tsai [26]	Proposed
frequency, MHz	76.4	79	30.8	47.3	33.5	13
gate count (0.18 mm)	16 855	17 202	13 230	13 192	13 189	10 405

and the macroblock processing rate, for evaluation. The Level limitation defines the upper bound of bitrates for transmission. Although a bitstream with low QP value may increase the number of processing cycles which leads to higher operating frequency in a CAVLC decoder, the received bitstream does not meet the Level limitation in the AVC/H.264 standards. Therefore the reference software from AVC/H.264 was used to encode the video sequences by enabling the rate controller to reach the upper bound bit-rate constraint. The generated bitstreams were decoded in our evaluation to meet the actual condition. Table 7 shows the Level limits and our simulation results. The proposed design can process Level 4.1 1080HD (1920 × 1088) video with a clock speed of under 13 MHz. The proposed CAVLC decoder has a high throughput and hence achieves high performance for AVC/H.264. Table 8 shows the minimum required operating frequency for various decoder designs to meet the real-time constraint for 1080HD resolution. The area of the proposed CAVLC decoder can be reduced by releasing the timing constraint while synthesising our circuit. The table shows that the proposed design operates at the lowest frequency and has the lowest gate count.

5 Conclusion

CAVLC uses context-based adaptation to remove statistical redundancy. However, these adaptations increase data dependency. Breaking the codeword dependency allows high-throughput VLSI architectures to be adopted to implement a multi-symbol decoder. Since CAVLC eliminates statistical redundancy to obtain better compression ratios, utilisation of statistical information is the key to high-throughput design. Using the codeword length distribution to obtain better prediction, the recursive dependency between codewords was broken, making the use of a parallel architecture possible. The multi-symbol decoder was proven to increase performance. In addition, combining the *Coeff_Token* and *TrailingOnes Sign* processes uses cycle time efficiently and decreases the total processing time. Results show that the proposed CAVLC decoder can be implemented for Level 4.1 1080HD (1920 × 1088) video applications with a clock speed of under 13 MHz.

6 Acknowledgments

The authors thank Professor Ming-Der Shieh, the director of the VLSI laboratory in National Cheng Kung University, for his helpful advice.

7 References

[1] SHANNON C.E.: 'A mathematical theory of communication', *Bell Syst. Tech. J.*, 1948, **27**, pp. 379–423, 623–656

[2] ISO/IEC 'Information technology-generic coding of moving pictures and associated audio information – part 2: video'. Standard ISO/IEC 13818–2, 1995

[3] WALLACE G.K.: 'The JPEG still picture compression standard', *Commun. ACM*, 1991, **34**, (4), pp. 30–44

[4] TEAM J.V.: 'Draft ITU-T recommendation and final draft international standard of joint video specification'. ITU-T Rec. H.264 and ISO/IEC 14496–10 AVC, May 2003

[5] XUE Q., LIU J., WANG S., ZHAO J.: 'AVC/H.264 baseline profile decoder optimization on independent platform'. Int. Conf.

doi: 10.1049/iet-ipr.2008.0064

on Wireless Communications, Networking and Mobile Computing, 2005, vol. 2, pp. 1253–1256

[6] LEI S.M., YANG M.T., TZOU K.H.: 'An entropy coding system for digital HDTV applications', *IEEE Trans. Circuits Syst. Video Technol.*, 1991, **1**, (1), pp. 147–155

[7] WU D., GAO W., HU M., JI Z.: 'A VLSI architecture design of CAVLC decoder'. Proc. ASIC Fifth Int. Conf., October 2003, vol. 2, pp. 962–965

[8] MOON Y.H., KIM G.Y., KIM J.H.: 'An efficient decoding of CAVLC in AVC/H.264 video coding standard', *IEEE Trans. Consum. Electron.*, 2005, **51**, pp. 933–938

[9] KIM Y.H., YOO Y.J., SHIN J., CHOI B., PAIK J.: 'Memory-efficient AVC/H.264 CAVLC for fast decoding', *IEEE Trans. Consum. Electron.*, 2006, **52**, (3), pp. 943–952

[10] CHANG H.C., LIN C.C., GUO J.I.: 'A novel low-cost high-performance VLSI architecture for MPEG-4 AVC/H.264 CAVLC decoding'. IEEE Int. Symp. on Circuits and Systems, May 2005, vol. 6, pp. 6110–6113

[11] SHIEH B.J., HSU T.Y., LEE C.Y.: 'A new approach of group-based VLC codec system'. IEEE Int. Symp. Circuits and Systems, 2000, vol. 4, pp. IV-609–IV-612

[12] HASHEMIAN R.: 'High speed search and memory efficient Huffman coding'. IEEE Int. Symp. on Circuits and Systems, 1993, vol. 1, pp. 287–290

[13] HASHEMIAN R.: 'Design and hardware implementation of a memory efficient Huffman decoding', *IEEE Trans. Consum. Electron.*, 1994, **40**, (3), pp. 345–352

[14] CHOI S.B., LEE M.H.: 'High speed pattern matching for a fast Huffman decoder', *IEEE Trans. Consum. Electron.*, 1995, **41**, pp. 97–103

[15] HSIEH C.T., KIM S.P.: 'A concurrent memory-efficient VLC decoder for MPEG applications', *IEEE Trans. Consum. Electron.*, 1996, **42**, pp. 439–446

[16] MUKHERJEE A., RANGANATHAN N., BASSIOUNI M.: 'Efficient VLSI design for data transformations of tree-based codes', *IEEE Trans. Circuits Syst.*, 1991, **38**, pp. 306–314

[17] PARK H., PRASANNA V.K.: 'Area efficient VLSI architectures for Huffman coding', *IEEE Trans. Circuits Syst.*, 1993, **40**, pp. 568–575

[18] LEE Y.S., SHIEH B.J., LEE C.Y.: 'A generalized prediction method for modified memory-based high throughput VLC decoder design', *IEEE Trans. Circuits Syst.*, 1999, **46**, (6), pp. 742–754

[19] SIMA M., COTOFANA S., VASSELIADIS S., VAN EIJNDHOVEN J.T.J., VISSERS K.: 'MPEG macroblock parsing and pel reconstruction on an FPGA-augmented TriMedia processor'. Int. Conf. on Computer Design, September 2001, pp. 425–430

[20] ALLE M., BISWAS J., NANDY S.K.: 'High performance VLSI architecture design for H.264 CAVLC decoder'. Proc. Int. Conf. on Application-Specific Systems, Architectures and Processors, 2006, pp. 317–322

[21] TSENG S.Y., HSIEH T.W.: 'A pattern-search method for AVC/H.264 CAVLC decoding'. IEEE Int. Conf. on Multimedia and Expo, July 2006, pp. 1073–1076

[22] OH M., LEE W., JUNG Y., KIM J.: 'Design of high-speed CAVLC decoder architecture for AVC/H.264', *ETRI J.*, 2008, **30**, (1), pp. 167–169

[23] CHEN T.W., HUANG Y.W., CHEN T.C., CHEN Y.H., TSAI C.Y., CHEN L.G.: 'Architecture design of AVC/H.264 decoder with hybrid task pipelining for high definition videos'. IEEE Int. Symp. on Circuits and Systems, May 2005, vol. 3, pp. 2931–2934

[24] YU G.S., CHANG T.S.: 'A zero-skipping multi-symbol CAVLC decoder for MPEG-4 AVC/H.264'. IEEE Int. Symp. on Circuits and Systems, May 2006, pp. 5583–5586

[25] NIKARA J., VASSILIADIS S., TAKALA J., LIUHA P.: 'Multiple-symbol parallel decoding for variable length codes', *IEEE Trans. VLSI Syst.*, 2004, **12**, (7), pp. 676–685

[26] TSAI T.H., FANG D.L.: 'A novel design of CAVLC decoder with low power consideration'. IEEE Asian Conf. on Solid-State Circuits, 2007, pp. 196–199

[27] http://ftp3.itu.ch/av-arch/jvt_site/draft_conformance, accessed 2006