High Throughput Parallel Decoding Method for H.264/AVC CAVLC

Donghoon Yeo and Hyunchul Shin

A high throughput parallel decoding method is developed for context-based adaptive variable length codes. In this paper, several new design ideas are devised and implemented for scalable parallel processing, a reduction in area, and a reduction in power requirements. First, simplified logical operations instead of memory lookups are used for parallel processing. Second, the codes are grouped based on their lengths for efficient logical operation. Third, up to *M* bits of the input stream can be analyzed simultaneously. For comparison, we designed a logical-operation-based parallel decoder for *M*=8 and a conventional parallel decoder. High-speed parallel decoding becomes possible with our method. In addition, for similar decoding rates (1.57 codes/cycle for *M*=8), our new approach uses 46% less chip area than the conventional method.

Keywords: Parallel decoding, context-based adaptive variable length coding (CAVLC), CAVLD, H.264/AVC.

I. Introduction

The increasing use of multimedia services and mobile terminals has resulted in requirements for higher coding efficiency. To meet this need, video coding standard H.264/AVC [1] was established by the ITU-T/ISO/IEC Joint Video Team. One of the techniques to increase efficiency is variable length coding (VLC). VLC achieves code compression by assigning short codewords to input symbols of high probability and longer codewords to those of low probability. H.264/MPEG-4 AVC includes two advanced VLC techniques, context-based adaptive variable length coding (CAVLC) and context-based adaptive binary adaptive coding (CABAC). CABAC is more complicated in computation but shows a higher compression rate than CAVLC. The H.264/AVC baseline profile only includes CAVLC [1]. When entropy_coding_mode is set to 0, residual block data is coded using CAVLC, and other variable-length coded units are coded using Exp-Golomb codes. Exp-Golomb codes are variable length codes with a regular construction. CAVLC residual coding consists of 5 syntax types: Coeff_token, Trailing_ones, Level, Total_zeros, and Run_before. Conventionally, CAVLC decoders use lookup tables that frequently require multiple memory accesses until the desired codeword is found. This heavy memory access results in high power consumption and delay in operations of multimedia terminals, such as digital multimedia broadcasting (DMB) players, portable media players (PMP), personal digital assistants (PDAs), and mobile phones with video capabilities [2], [3].

In [3], techniques were reported to decode Run_before without lookup tables based on the observation that some codewords have systematic patterns. For Coeff_token, the codewords with high frequency are directly decoded by integer arithmetic operations. In [2], memory accesses were further

Manuscript received Feb. 17, 2009; revised June 14, 2009; accepted June 29, 2009.

This research was supported by the Ministry of Knowledge Economy (MKE), Rep. of Korea, under the Information Technology Research Center (ITRC) support program supervised by the Institute for Information Technology Advancement (IITA) (IITA2008 (C1090080400009)).

Donghoon Yeo (phone: +82 31 400 4673, email: dhyeo@digital.hanyang.ac.kr) and Hyunchul Shin (phone: +82 31 400 5176, email: shin@hanyang.ac.kr) are with the School of Electrical and Computer Engineering, Hanyang University, Ansan, Rep. of Korea.

doi:10.4218/etrij.09.0109.0110

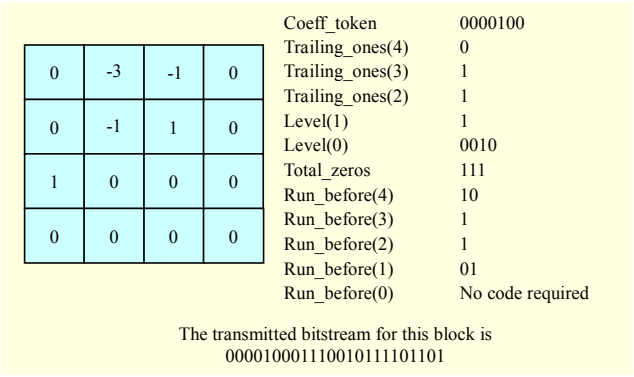

Fig. 1. CAVLC for a 4×4 block.

reduced by using arithmetic operations. However, this method still depends on a large number of table lookups.

In this paper, a logical-operation-based decoding scheme is described and used for all syntax types. Logical operation is simpler to perform than arithmetic operation and therefore consumes less power. The rest of this paper is organized as follows. In section II, typical designs using memory lookups, arithmetic operations, and logical operations are comparatively explained. In section III, parallel CAVLC decoding is described. In section IV, experimental results are shown. Finally, conclusions are summarized in section V.

II. Decoder Design Methods

In this section, two typical design methods and our method are described: memory lookups [4], arithmetic decoding [3], and logical operation techniques. Logical-operation-based decoding is the new method we have developed.

An example shown in Fig. 1 is used to explain the features of the three methods. In the figure, 4×4 DCT coefficients and the codes of Coeff_token, Trailing_ones, Level, Total_zeros, and Run_before are shown [5]. Zigzag scanning produces the sequence 0, -3, 0, 1, -1, -1, 0, 1. There are 5 Non-zero coefficients, 3 Trailing_ones, and 3 Total_zeros. Run_before is repeated until Zero_left becomes 0. The transmitted bitstream is shown at the bottom of Fig. 1.

1. Memory Lookup

In memory lookup methods, a codeword is used as an address for the lookup table. Since the number of entries can be as large as 2^{code_length}, straightforward implementation is not practical. Therefore, the lookup table is usually partitioned into several smaller ones. High-frequency codewords are put in lookup tables with small indices. If no codeword is found in the current lookup table, the next-layer lookup table is searched to find the current codeword [4]. For the example

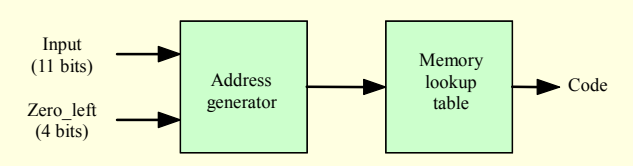

Fig. 2. Run_before decoding by lookup table.

shown in Fig. 1, Run_before values are found by referring to lookup tables as shown in Fig. 2. However, this method is difficult to parallelize.

2. Arithmetic Decoding

CAVLC decoding using arithmetic operations was developed to reduce memory accesses and thus reduce power consumption. The codes are grouped using the correlation of VLC codes, and arithmetic operations can replace the table lookup process [2]. This is an improved software decoding method. As an example, the Run_before codes in Fig. 1 are decoded by the following arithmetic operations. For simplicity, grouping and state checking are not shown.

$$\text{Run\_before(4): Zero\_left} - \text{Run\_before(4)[1:0]} = 3 - 10_2 = 1$$

$$\text{Run\_before(3): (2–Run\_before(3)[1:0])}\cdot\text{(1–Run\_before(3)[1])} = (2 - 1_2)\cdot(1 - 1_2) = 0$$

$$\text{Run\_before(2): (2–Run\_before(2)[1:0])}\cdot\text{(1-Run\_before(2)[1])} = (2 - 1_2)\cdot(1 - 1_2) = 0$$

$$\text{Run\_before(1): } 1 - \text{Run\_before(1)[1]} = 1 - 0_2 = 1$$

The weakness of this arithmetic method is that arithmetic operations are only able to replace memory lookups, which makes parallelization difficult.

3. Logical-Operation-Based Decoding

For efficient hardware implementation, we use logical operations instead of memory lookup tables or arithmetic operations. As an example, the Run_before codes in Fig. 1 are decoded by using the simple logical operations shown in Fig. 3. The logical operations are simple (fast) and thus consume less power.

4. Multiple-Symbol Parallel Variable Length Decoding Technique

Due to the sequential characteristics of variable length coding, parallelization is difficult. In [6], a multiple-symbol parallel variable length decoding method was reported for MPEG 2. All possible codewords starting from every bit were searched. The first matched code was selected, and then the codeword starting from the next bit was selected. This selection

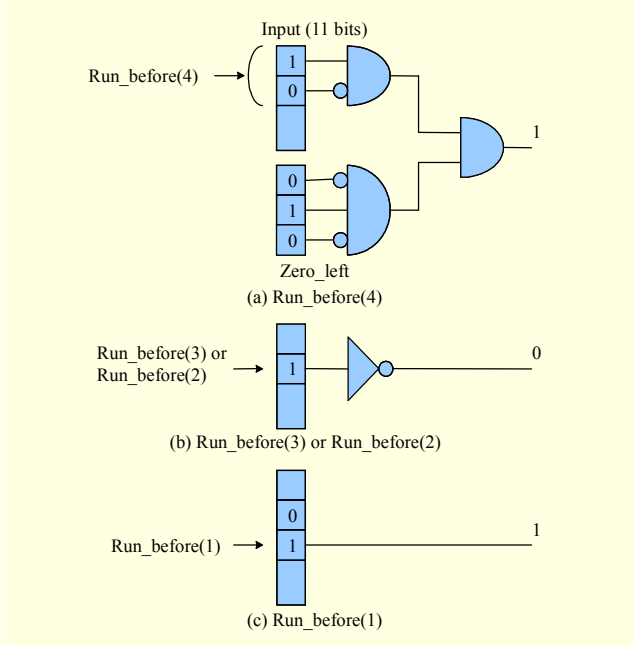

Fig. 3. Run_before decoding by logical operation.

step can be repeated to decode the N-bit input. In other words, all possible codewords are found in parallel, and the correct codewords are selected. We implemented this method for H.264 for comparisons of results.

III. Parallel Decoder Design

We now explain our new parallel decoding scheme based on logical operations. CAVLC decoding is sequentially performed in 5 steps in order to decode 5 syntax elements. Coeff_token and Total_zeros appear only once per macroblock, while Trailing_ones can appear up to three times. However, the code length is not more than 3 bits, so parallel processing is not necessary. Several iterations can be repeated in Level and Run_before. Therefore, parallel processing is performed only for Level and Run_before. In parallel processing, at least one codeword is decoded every cycle, and all codewords whose sum of lengths is less than or equal to M are decoded in one cycle. When M is large, many codewords can be decoded in a cycle at the cost of a larger area. We designed the parallel CAVLC decoder with M=8.

Let $C_1, C_2, C_3, \cdots, C_i, \cdots$ be the input bit stream and l_i be the code length of C_i.

Case 1. $l_1 \geq M$.

Only C_1 is decoded by a normal decoder.

Case 2. $l_1 < M$.

We find the maximum integer k such that $\sum_{i=1}^{k} l_i \leq M$.

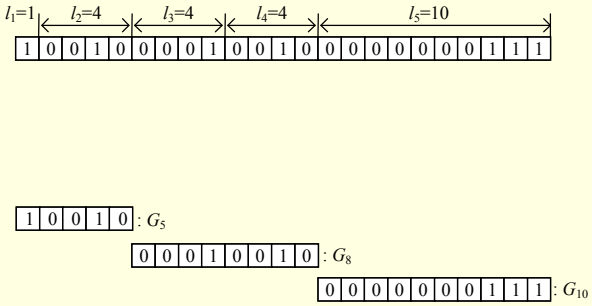

Fig. 4. Example of parallel decoding when M=8.

Then k codewords are decoded in a cycle.

When $M = 8$, since $\sum_{i=1}^{2} l_i = 5 < 8$ and $\sum_{i=1}^{3} l_i = 9 > 8$, two codewords, C_1 and C_2, are decoded in a cycle, and the bit stream is shifted by 5 bits. In the next cycle, C_3 and C_4 are decoded since $l_3 + l_4 = 8$ as shown in Fig. 4.

1. Proposed Parallel Decoding of CAVLC

CAVLC decoding is processed by using the five steps of Coeff_token, Trailing_ones, Level, Total_zeros, and Run_before. A step can be started when the previous step is completed. This sequential nature makes parallel processing among the steps difficult. The steps Coeff_token and Total_zeros occur only once during the decoding of each macroblock, while the remaining steps (Trailing_ones, Level, and Run_before) can occur several times.

The decoding of the data stream within a step is also not straightforward because decoding of a given code is dependent on the decoding of the preceding codes. However, our proposed parallel algorithm can decode in parallel by considering the dependencies. All the possible combinations of codes are considered for correct parallel decoding.

In our decoder, the five steps and tables are identified by the State_select variable as shown in Table 1. When the most significant bit of State_select is 1, parallel decoding can then be performed. Otherwise, sequential decoding is performed. Now, we explain the proposed parallel decoding method in detail. Let us consider when M = 8. In this case, the codewords within the 8 bits are decoded.

There are 256 kinds of data streams when M = 8, from 00000000 to 11111111. Because a significant portion of these are not used as codewords, the eight-bit input data stream has many "don't care" cases, which helps logic optimization.

A. Coeff_token

Coeff_token is the first step in macroblock decoding and is executed only once. Total_coeff and Trailing_ones are decoded in Coeff_token. This step uses four tables: Num_VLC0,

Table 1. State_select for determining steps and tables.

Steps	Tables	State_select
Step 1. (Coeff_token)	Num_VLC0	0000
	Num_VLC1	0001
	Num_VLC2	0010
	Num_VLC_DC	0011
Step 2. (Trailing_ones)		0100
Step 3. (Level)	Level_VLC0	1000
	Level_VLC1	1001
	Level_VLC2	1010
	Level_VLC3	1011
	Level_VLC4	1100
	Level_VLC5	1101
	Level_VLC6	1110
Step 4. (Total_zeros)	Total_zeros	0110
	Total_zeros_DC	0111
Step 5. (Run_before)	run_before	1111

Table 2. Choice of lookup table for Coeff_token.

N	Tables for coeff_token
0, 1	Num_VLC0
2, 3	Num_VLC1
4, 5, 6, 7	Num_VLC2
8 or above	Num_VLC_DC

Table 3. Thresholds for determining Level_VLC tables.

Current lookup table	Thresholds
Level_VLC0	0
Level_VLC1	3
Level_VLC2	6
Level_VLC3	12
Level_VLC4	24
Level_VLC5	48
Level_VLC6	N/A

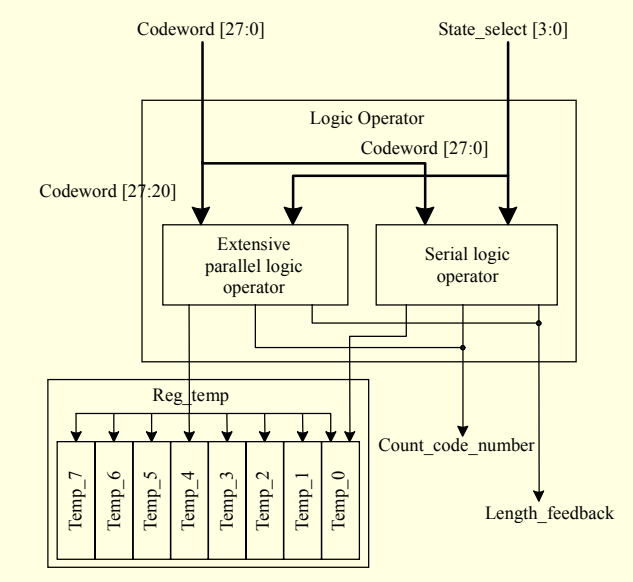

Fig. 5. Structure of level block.

Num_VLC1, Num_VLC2, and Num_VLC_DC.

Let N_u be the number of nonzero coefficients of the upper block (of the current block being decoded), and let N_l be the number of nonzero coefficients of the left block. Then, $N=(N_u+N_l)/2$. Based on the value of N, one of the tables is selected using Table 2. Parallelization is not necessary since this step is executed only once.

B. Trailing_ones

Trailing_ones found in the previous step represent the number of Trailing_ones (1 or -1) in the macroblock. Because each Trailing_one is coded by one bit, Trailing_ones can be decoded at once.

C. Level

The Level step uses 7 lookup tables, namely, Level_VLC0, ···, Level_VLC6, as shown in Table 3. At the beginning of the Level step, Level_VLC0 is usually accessed. One exceptional case is when Total_coeff is greater than 10 and Trailing_ones is less than 3, in which case, Level_VLC1 is accessed. When the decoded Level is greater than the threshold values, the next Level_VLC table is used. If the decoded Level is less than or equal to the threshold, then the current Level_VLC table is used. Therefore, if the Level_VLC(N) table is used, then the next table used is either Level_VLC(N) or Level_VLC(N+1), depending on the decoded value. Because the number of branches is limited (2 in this case), simple and efficient parallel processing is feasible by decoding all the possible cases.

The decoding steps can be explained by using the example shown in Fig. 1. Because Total_coeff = 3, the initial table to access is Level_VLC0. The absolute value of the decoded Level (1) coefficient is 1 and thus is greater than the threshold 0. Therefore, the Level_VLC1 table is to be used for the next decoding. Level (0) = 0010 and the absolute value of decoded Level (0) is 3. Level_VLC1 is to be used next. This decoding process is also executed in parallel by efficient logical

Table 4. Level_VLC lookup tables.

(a) Level_VLC0

Code	Level
1	1
01	-1
001	2
0001	-2
00001	3
000001	-3
0000001	4
00000001	-4
⋮	

(b) Level_VLC1

Code	Level
1x	±1
01x	±2
001x	±3
0001x	±4
00001x	±5
000001x	±6
0000001x	±7
⋮	

(c) Level_VLC2

Code	Level
1xx	±1 to ±2
01xx	±3 to ±4
001xx	±5 to ±6
0001xx	±7 to ±8
00001xx	±9 to ±10
000001xx	±11 to ±12
⋮	

(d) Level_VLC3

Code	Level
1xxx	±1 to ±4
01xxx	±5 to ±8
001xxx	±9 to ±12
0001xxx	±13 to ±15
00001xxx	±17 to ±19
⋮	

(e) Level_VLC4

Code	Level
1xxxx	±1 to ±8
01xxxx	±9 to ±16
001xxxx	±17 to ±24
0001xxxx	±25 to ±32
⋮	

(f) Level_VLC5

Code	Level
1xxxxx	±1 to ±16
01xxxxx	±17 to ±32
001xxxxx	±33 to ±64
⋮	

(g) Level_VLC6

Code	Level
1xxxxxx	±1 to ±32
01xxxxxx	±33 to ±64
⋮	

operations that consider all the possible cases.

Figure 5 shows a block diagram of the Level step process when $M = 8$. Since all the possible 256 cases are considered in the proposed parallel logic operator shown in Fig. 5, all the codewords in 8 bits can be decoded at once. The State_select signal determines the step and the initial table. Consider a case when a codeword [27:20] = 00100111. Here, we describe four of the seven cases.

Case 1. State_select = Level_VLC0.

Two codes, 001 and 0011, are decoded in parallel. From table Level_VLC0, 001 is decoded, and the decoded value 2 is stored in Temp_0. From table Level_VLC1, 0011 is decoded and the value of -3 is stored in Temp_1. Since there are only two codewords decoded, the results are

Temp_0 = 2,
Temp_1 = -3,
Temp_2 = Temp_3 = ⋯ = Temp_7 = 0, (0 means null or no codeword).

The proposed parallel logic operation is synthesized to produce the above outputs when the input is 0010011d (d means don't care) in this case.

Case 2. State_select = Level_VLC1

Two codes, 0010 and 011, are decoded in parallel, and we get 3 and -2 using table Level_VLC1. Thus, the results are

Temp_0 = 3,
Temp_1 = -2,
Temp_2 = ⋯ = Temp_7 = 0.

Case 3. State_select = Level_VLC2

By using the table of Level_VLC2, the following results are obtained:

Temp_0 = 5,
Temp_1 = -2,
Temp_2 = ⋯ = Temp_7 = 0.

Case 4. State_select = Level_VLC3

One codeword, 001001, is decoded, producing the following results:

Temp_0 = -6
Temp_1 = ⋯ = Temp_7 = 0.

D. Total_zeros

Total_zeros represents the number of zero coefficients and are decoded only once. Total_zeros and Trailing_ones are decoded by the serial logic operator show in Fig. 5.

E. Run_before

Decoding in the Run_before step is repeated until Zero_left becomes 0. The initial value of Zero_left is the Total_zeros

Table 5. Run_before table.

Run_ before	Zero_left						
	1	2	3	4	5	6	More than 7
0	1	1	11	11	11	11	111
1	0	01	10	10	10	000	110
2		00	01	01	011	001	101
3			00	001	010	011	100
4				000	001	010	011
5					000	101	010
6						100	001
7							0001
8							00001
9							000001
10							0000001
11							00000001
12							000000001
13							0000000001
14							00000000001

found in the previous step. As in the decoding in the Level step, the decoding condition for the next code is determined by the current decoded codeword in the Run_before step. We also parallelized the decoding in the Run_before step by considering all the possible cases (256 cases when $M = 8$).

Table 5 is used to update Zero_left after decoding each codeword. When a codeword is decoded, the appropriate Run_before value in Table 5 is found, and the value is subtracted from the current Zero_left. For example, let the input data stream be 10010000 and let Zero_left be 7. Then, the first Run_before code is '100' and the Run_before value is 3 from Table 5. Now, Zero_left is updated to 4 (=7–3). The second code is '10' and the Run_before value is 1 from Table 5. Then, Zero_left is updated to 3 (=4–1). The third code is '00' and the Run_before value is 3 from the table. Since Zero_left becomes 0 (=3–3), the Run_before step is completed. In our extensive parallel approach, these three codewords ('100', '10', and '00') are simultaneously decoded in parallel, since all the possible combinations are considered in the proposed parallel decoding step. Consider the example in Fig. 1. Run_before(4) to Run_before(1) codewords are '10', '1', '1', and '01' (or '101101'). In our method, these four codewords are decoded at once in one cycle, while the sequential decoding method [2] takes four cycles to decode them. Since many short codewords appear in the Run_before step when compared to those in the level step, significant speed up is possible by parallelizing the decoding in the Run_before step.

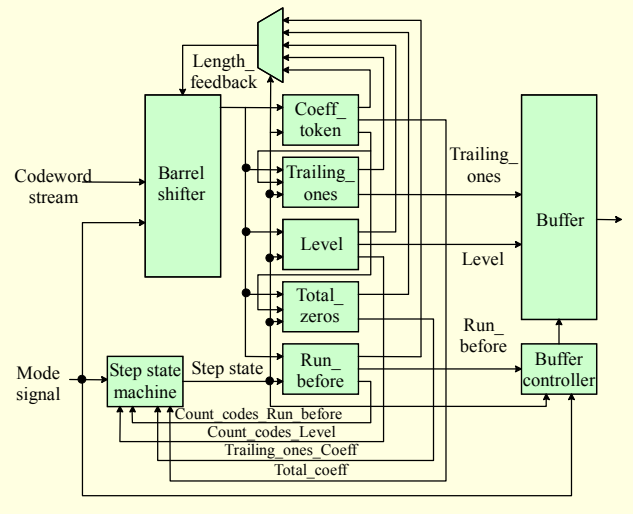

Fig. 6. Logical-operation-based parallel CAVLC decoder.

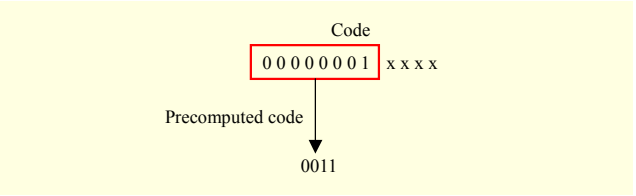

Fig. 7. Output of prefix precomputaion.

Figure 6 shows a block diagram of our CAVLC decoder.

2. Further Optimization by Using Prefix Precomputation

In memory-lookup-based decoding methods, memory partition techniques are widely used. These decode the prefix parts separately to reduce the total memory size [2], [3]. However, we use logic circuits instead of memory. Therefore, if the logic synthesis tool is ideal in logic optimization, precomputation is unnecessary. In practice, logic synthesis tools are not ideal, and the quality of the results is dependent on the input description. In other words, if the input format capitalizes the structures of the logical operations, better solutions can be obtained from a logic synthesizer.

Inspired by memory partitioning, we have developed prefix precomputation techniques to reduce the decoder area. Figure 7 shows an example codeword which is 12 bits long and whose prefix is 8 bits long. If we precompute this prefix as 0011, then the input to the logic operator can be reduced from 12 bits to 8 bits, as shown in Fig. 8. Figure 8(a) shows the original logic circuit, and Fig. 8(b) shows the circuit optimized by using prefix precomputation. When we synthesize the circuits by using a well-known commercial logic synthesizer, the original circuit takes 33 instances (gates). However, the circuit optimized by using prefix precomputation takes only 23

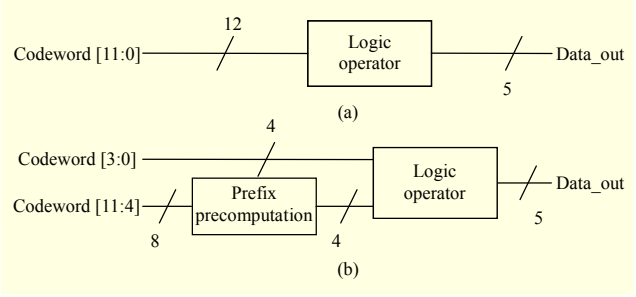

Fig. 8. Comparison of design with symbol logic operation.

instances (gates). Furthermore, the optimized circuit uses smaller gates due to the simpler circuit structure. Table 6 shows the precomputation table we implemented for prefix values from 00001 to 0000000000000001. Using this precomputation, the circuit area can be further reduced by 11% when compared to the original circuit.

IV. Experimental Results

CAVLC is decoded in 5 steps (states). We use logical operations instead of memory lookup tables or arithmetic operations. Proposed parallel decoding is used for Level and Run_before states. Since Run_before statistically contains many short codes, parallel processing is more effective for Run_before than for Level.

We also designed a parallel CAVLC decoder based on multiple-symbol parallel decoding [6] to compare its performance and area with those of the proposed decoder. The input stream was obtained by using JM 10.2 with QP = 24 from the Foreman video sequence data. Tables 7 and 8 show performance and area comparisons, respectively. Both of the decoders targeted to the baseline profile were designed with the Synopsys Design Analyzer using the Hynix 0.25 μm library. The decoders were synthesized for a 50 MHz clock rate. In the figure, CD indicates the number of codeword detectors used.

In the current implementation, M = 8 and 8 bits of input stream are analyzed simultaneously. At least one codeword is decoded in a cycle and multiple codewords can be decoded if the sum of their code lengths is not greater than M. For M=8, an average of 1.57 codewords are decoded in a cycle for the Foreman sequence. The results for the Mobile sequence are very similar. The performance (throughput) of our decoder is similar to that of the multiple-symbol parallel decoder with 6 codeword detectors (CD #6). However, our decoder uses 40% less area when compared to the multiple-symbol parallel decoder [6]. The decoder described in [4] can decode 0.05 or 0.04 codes/cycle and shows significantly lower decoding rates than those of our proposed decoder and that proposed in [6].

Table 6. Precomputed symbols.

Prefix	Precomputed symbols	
00001	A	0000
000001	B	0001
0000001	C	0010
00000001	D	0011
000000001	E	0100
0000000001	F	0101
00000000001	G	0110
000000000001	H	0111
0000000000001	I	1000
00000000000001	J	1001
000000000000001	K	1010
0000000000000001	L	1011

Table 7. Performance comparison (QP=24).

		Throughput	
		Foreman	Mobile
Multiple-symbol parallel decoding [6]	CD #6	1.56 code/cycle	1.62 code/cycle
	CD #7	1.59 code/cycle	1.68 code/cycle
	CD #8	1.63 code/cycle	1.75 code/cycle
	CD #32	1.98 code/cycle	2.13 code/cycle
Method in [4]		0.05 code/cycle	0.04 code/cycle
Our method (extensive parallel) M = 8		1.57 code/cycle	1.64 code/cycle

Table 8 shows a comparison in terms of area for the methods proposed in [4] and [6], our original method, and our improved method. In [4], gate counts excluding buffers are shown, and the area is not shown. Therefore, area cannot be directly compared with the method in [4]. When compared to our original method, our improved method using prefix precomputation requires 11% less area. When compared to the multi-symbol parallel decoding method in [6], our improved decoder uses 46% less chip area while maintaining similar performance (CD #6).

V. Conclusion

For conventional CAVLC decoding methods, such as memory lookup and arithmetic methods, parallel decoding is difficult due to its sequential nature. In this paper, a new logical-operation-based CAVLC decoding scheme was proposed as well as a parallel decoding architecture to efficiently process Level and Run_before steps. Among the

Table 8. Area and gate-count comparison.

		Area (μm^2)	Gate-counts
Multiple-symbol parallel decoding [6]	CD #6	233,928	10,172
	CD #7	263,019	11,435
	CD #8	287,384	12,564
	CD #32	615,867	24,576
Method in [4]		Not available	4,720*
Our method without precomputaion		139,252	9,623
Our method with precomputaion		124,847	7,456

* excluding buffers

syntax elements, Coeff_token and Total_zeros appear only once per macroblock, so parallel processing is not necessary. Trailing_ones can appear up to three times with code lengths up to 3 bits, so the effect of parallel processing is minor.

For parallel decoding up to M bits, all the various 2^M input cases can be simultaneously analyzed. However, there are many "don't care" cases resulting in simpler decoder logic circuits. If the code length is greater than M, only one codeword is decoded. Otherwise, all the codewords within the M bits are simultaneously decoded. We implemented the decoder for M=8. When compared to multiple-symbol parallel decoding [6], our decoder uses 46% less area to achieve the same performance in throughput.

References

[1] Joint Final Committee Draft (JFCD) of Joint Video Specification (ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC), *Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG*, 4th Meeting, Klagenfurt, Austria, 22-26 July 2002.

[2] Y.H. Kim et al., " Memory-Efficient H.264/AVC CAVLC for Fast Decoding, " *IEEE Trans. Consumer Electronics*, vol. 52, no. 3, Aug. 2006, pp. 943-952.

[3] Y.H. Moon, G.Y. Kim, and J.H. Kim, "An Efficient Decoding of CAVLC in H.264/AVC Video Coding Standard," *IEEE Trans. Consumer Electronics*, vol. 51, no. 3, Aug. 2005, pp. 933-938.

[4] H. Chang, C. Lin, and J.I. Guo, "A Novel Low-Cost High-Performance VLSI Architecture for MPEG-4 AVC/H.264 CAVLC Decoding," *IEEE Int. Symp. Circuits and Systems*, vol. 6, 2005, pp. 6110-6113.

[5] E.G Iain, *H.264 and MPEG-4 Video Compression*, Wiley, 2003.

[6] J. Nikara et al., "Multiple-Symbol Parallel Decoding for Variable Length Codes," *IEEE Trans. Very Large Scale Integration Systems*, vol. 12, no. 7, July 2004, pp. 676-685.

Donghoon Yeo received the BS degree in electrical and computer engineering from Hanyang University, Ansan, Korea, in February 2006. He received his MS degree in mechatronics engineering from Hanyang University in August 2008. His research interests include H.264 and VLSI system design.

Hyunchul Shin received the BS degree in electronics engineering from Seoul National University in 1978, and the MS degree in electrical engineering from the Korea Advanced Institute of Science and Technology in 1980. He received the PhD in electrical engineering and computer sciences from the University of California at Berkeley in 1987. From 1980 to 1983, he was with the Department of Electronics Engineering at the Kumoh National Institute of Technology, Korea. In 1983, he received a Fullbright scholarship. From 1987 to 1989, he was a member of the technical staff at AT&T Bell Laboratories, Murray Hill, NJ. Since 1989, he has been a professor with the School of Electrical and Computer Engineering of Hanyang University, Korea. He is the president of the Multi-core Design Methodology (MDM) Research Center. His research interests include the design and synthesis of integrated circuits and systems.