Area Efficient H.264/AVC CAVLC Decoder Architecture

Byung-Sik Choi and Jong-Yeol Lee
College of Engineering, Division of Electronics, Chonbuk National University
664-14 1ga Deokjin-dong, Jeonju, Jeonbuk, South Korea
E-mail: choi4702@chonbuk.ac.kr, jong@chonbuk.ac.kr

Abstract

In this paper, we propose an area-efficient VLSI architecture of H.264/AVC CAVLC decoder. In the proposed architecture, we reduce the decoder area by rearranging the lookup tables. We also save the bus area and cycles by delaying T1s decoding to the final reordering step which is performed in output buffer. To remove the overlapped logics we combine a controller and a barrel shifter. By using the proposed architecture, we can reduce the area by about 30% compared with previous work. We design the proposed decoder using Verilog HDL and synthesize using 0.35μm standard cell library. We verify the proposed architecture by simulation that the designed decoder can run at the frequency of 50Mhz.

1. Introduction

H.264/AVC is adopted for the standard of video CODEC. It is made by jointly ITU-T and MPEG for the purpose of high compression rate and network-friendly service. The efficiency of H.264/AVC is over two times better than previous CODECs. Furthermore, it shows better quality than previous CODECs. However, since it is 2.5 times[1] and four times[2] more complex than H.263 and MPEG-4, respectively, we need a hardware-based design of H.264/AVC to decode images in real-time.

Previous CAVLC decoders use forward zig-zag scan and fixed variable length coding (VLC) for variable coefficient coding. However, H.264/AVC uses inverse zig-zag scanned run-length coding and context-adaptive VLC in order to improve a compression rate. We use CAVLC decoding method to decode coefficients, and use Exp-Golomb coding method to decode syntaxes.

Recently, usage of mobile devices such as MP3P, PMP, and Navigator where H.264/AVC is used a video CODEC, is getting higher and the power consumption in those devices also increases as power-hungry applications like watching TV or moves prevail. To reduce power consumption in CAVLC decoder, we propose the CAVLC decoder architecture that occupies small area and consumes less power.

This paper is organized as follows. We present the CAVLC decoding algorithm and the previous works in Section 2. After we describe the proposed architecture in Section 3, Section 4 shows H/W cost comparison with previous works. Section 5 concludes this paper.

2. Algorithm and Previous works

In this Section, we present the CAVLC decoding algorithm in H.264/AVC and the previous works.

2.1 CAVLC decoding algorithm

Entropy decoder decodes each syntax element by using UVLC decoder, and decode the residual data with CAVLC decoder. In other words, CAVLC decoding is the process that decodes the temporal redundant data and frequency redundant data. To decode the residual data, it needs some variables such as Slice types, maxCoeffnum, CBP(Coded Block Pattern). Decoding is performed based on Macroblock. Macroblock is composed of Luma DC blocks, Luma AC blocks, Chroma AC blocks based on 4x4 blocks, and Chroma DC blocks based on 2x2 blocks as shown in Fig 1.

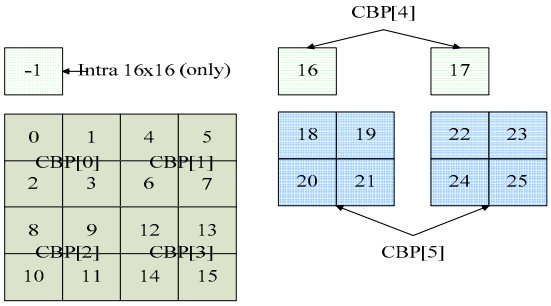

Fig. 1. Configuration of MB and Processing order

The processing of Luma DC is followed by Luma AC, Chroma DC, and Chroma AC. Slice type indcates the current slice is intra slice or inter slice. maxCoeffnum represents the size of a current Macroblock. CBP represents which 8x8 block is active or not. CAVLC decoding consists of 5 steps as follows:

Step1. coeff_token: *coeff_token* is the first decoded parameter including both the total number of non-zero coefficients (TotalCoeff) and the number of trailing ones without sign (T1s). TotalCoeff should be any value from 0 to 16 and T1s is from 0 to 3. There are five choices of look-up table which is used to decode *coeff_token*, and the parameter nC should be computed for which look-up table should be selected before the *coeff_token* decoding.

Step2. sign of T1s: For each T1 signalled by *coeff_token*, the sign is decoded with a single bit in reverse order, 0 is for +1 and 1 is for -1.

Step3. level: The codeword of each level consists of a prefix (*level_prefix*) and a suffix (*level_suffix*), and every level of non-zero coefficient is decoded in inverse order. For an example, a *level* should be encoded as 0…01x…xs. The "0…01" is so-called prefix, the "x…x" is suffix, and "s" is the sign of the level. The length of *suffixLength* is from 0 to 12 bits, and should be preset to either 0 or 1. With a level decoding, the *suffixLength* will be adjusted according to the magnitude of each decoded level.

Step4. total_zeros: In this step, the sum of all zeros, *total_zeros*, leading the first non-zero coefficient in reverse zigzag order is decoded with VLC tables. There are 15

tables for 4x4 blocks and 3 tables for 2x2 blocks. Based on the number of TotalCoeff, the corresponding table is selected to decode the correct *total_zeros*.

Step5. run_before: In the last process, *run_before*, the number of zeros which is leading every non-zero coefficient decoded in reverse zigzag order. There are seven tables whose selection is depended on *zeroLeft*. The parameter *zeroLeft* is obtained by subtract previous one from *zeroLeft* and initialized with *total_zeros*.

2.2 Previous CAVLC decoder architecture

The previous architectures of hardware-based CAVLC can be divided into two groups[3]. First method is a bit-serial method, which is not suitable for a real-time CAVLC decoder because it needs many cycles to decode each symbol. Second one is a bit-parallel method. It can decode each syntax and coefficient at high-speed that is suitable for decoding symbols in real-time processing.

In previous works, to reduce the processing cycle, multi-symbol decoding method[4][8], frequent pattern prediction[6], decoding step changing[8][12] and a zero-skipping method[10] are exploited. In [3], H/W cost is reduced by using lookup tables that are divided by prefixs and suffixs exploiting first_one_detector[3]. [5] uses arithmetic relations instead of lookup tables in order to reduce H/W cost. In pipelined architectures [12][13], they use own modified suffixlength detector to find out the level suffixlength. Since level decoding takes the longest time among 5 steps, if the critical path of level decoding is shrunken, we can decode it in higher frequency. For lower power consumption [7] and [9] proposed a combined lookup table composed of similar frequent entries used in most decoders.

3. Proposed CAVLC decoder architecture

In the proposed architecture, we modify entry searching method for decoding each symbol in LUTs by searching similar suffix patterns first. T1s decoding is moved into the final steps to reduce T1s buses and controller and barrel shifter is merged to reduce overlapped hardware.

3.1 Coeff_token decoding

In this step, we decode Total_coeff and a number of T1s based on lookup tables. If we implement all entries of the lookup tables specified in standard, H/W cost becomes huge. By diving the tables into prefix and suffix as shown in Fig. 2 and storing only suffixs into lookup table, we can reduce the H/W cost.

Coeff \ T1s	0	1
0	1	
1	000101	01
2	00000111	000100
3	00000011 1	00000110
4	0000000111	000000110
5	00000000111	0000000110
6	0000000001011	00000000110

=>

Coeff \ T1s	0	1
0	1	
1	101(3)	1(1)
2	111(5)	100(3)
3	111(6)	110(5)
4	111(7)	110(6)
5	111(8)	110(7)
6	1011(9)	110(8)

Fig. 2 Coeff_token lookup table consisted of prefixs and suffixes. The number in () is the number of prefixs

If we rearrange these entries shown in Fig. 3, we can find three groups of frequent patterns, which is underlined entries in Fig. 3, according to a prefix and nC. These three groups exist for all nC except for the case with nC<=8.

Unlike the previous works where they match nC first and then a prefix and a suffix to decode Total_Coeff and T1s, in proposed method, we match a suffix first and a prefix and nC next. By using this method, the number of bits when matching a suffix can be reduced. The proposed matching works as follows.

First, we check the first bit of a suffix according to a prefix and nC. If there is no entry that matches input stream, check following bit in a suffix. It is repeated until last bit of suffix. By using this method, we can reduce H/W cost of Coeff_token LUT. Lookup table is implemented by combinational logics.

Coeff \ T1s	0	1	2	3
0	1(0)			
1	101(3)	1(1)		
2	111(5)	100(3)	1(2)	
3	111(6)	110(5)	101(4)	11(3)
4	111(7)	110(6)	101(5)	11(4)
5	111(8)	110(7)	101(6)	100(4)
6	1111(9)	110(8)	101(7)	100(5)
7	1011(9)	1110(9)	101(8)	100(6)
8	1000(9)	1010(9)	1101(9)	100(7)
9	1111(10)	1110(10)	1001(9)	100(8)
10	1011(10)	1010(10)	1101(10)	1100(9)

Fig. 3 Part of the modified Coeff_token lookup table(0<=nC<2)

Group 1	Group 2	Group 3
1111	111	11
1110	110	10
1101	101	
1100	100	
1011		
1010		
1001		
1000		

Table \ Group	1	2	3	Etc.
0<=nC<2	9,10,11,12	3,4,5,6,7,8,13	N/A	0,1,2
2<=nC<4	2,7,8,9,10	1,3,4,5,6,11	0	N/A
4<=nC<8	0,1,2,3,4,5,6	7	8	9
8<=nC	N/A	N/A	N/A	N/A
nC=-1	N/A	3	4,5,6	0,1,2,7
nC=-2	3	6,7,8,9,10	N/A	0,1,2,4,5

Fig. 4 Frequent groups in Coeff_token LUT

3.2 T1s decoding

We decode T1s according to the bit of T1s. In previous works, it is decoded either in Coeff_token decoder or Level decoder to reduce processing cycle in one clock cycle. The decoded result is transferred to a reordering unit using three 13-bit buses. In order to reduce the width of bus and processing cycle, we move T1s decoding logic into output buffer, which reorders the non-zero coefficients. According to the proposed method, we can reduce width of dedicated buses and processing cycle for T1s.

3.3 Level decoding

The number of Level is derived from Coeff_token decoder by subtracting T1s from Total_coeff. In this step, we decode Level value using a prefix and a suffix. After decoding, Level value is stored into the output buffer using proposed method in [13]

3.4 Total_zeros decoding

Total_zeros is the total number of 0's before the last non-zero coefficient. In this step, we also use an LUT to decode Total_zeros and use Total_coeff by the index of LUT as shown in Fig. 5.

Total_ zeros	Total_Coeff				
	1	2	3	4	…
0	1	111	0101	00011	…
1	011	110	111	111	…
2	010	101	110	0101	…
3	0011	100	101	0100	…
4	0010	011	0100	110	…
…	…	…	…	…	…

Fig. 5 Total_zeros LUT

By investigating the entries of the LUT in Fig. 5, we can find similar two entries which are depicted as the underlined entries in Fig. 5 except for the last bit. Based on this investigation we use one entry instead of two entries by using MUXs as in Fig. 6, which can lead to 47% H/W cost reduction compared with the previous architectures.

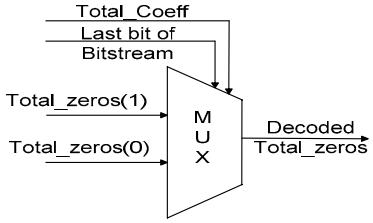

Fig. 6 Proposed Total_zeros decoding method

3.5 Run_before decoding

Run_before is the number of 0's between non-zero coefficients. We insert zeros instead of the non-zero coefficient according to Run_before. In this step, because the entires of the Run_before LUT have two similar patterns excluding the last bit, we can reduce an LUT in the same way with Total_zeros decoder by using MUXs.

3.6 Controller

In previous works, whenever dealing with total decoded bits, controller is designed to accumulate decoded bits in one cycle after finishing each symbol decoding. If total decoded bits are over pre-defined bits, the controller loads next bit-stream from external memory and rearrange bit-stream. [11] proposed 64-bit width shift register architecture instead of 32-bit width shift register to reduce the CAVLC decoding processing cycle when dealing with total decoded bits. In this architecture, total decoded bits are over pre-defined threshold bits, load signal is transmitted to barrel shifter and rearrange temporal bit-stream. In proposed architecture, we also use a 64-bit width Barrel shifter that is included in the controller to reduce overlapped logics. If we use the Barrel shifter at the controller, we can reduce the gate count of the controller and the Barrel shifter. Table 1. shows comparison results to [13].

Table 1 Barrel Shifter Area Comparison, The number in () means cell area for a unit element

Barrel Shifter	Combi. Logic	Non-Combi. Logic	Total
[13]	-	-	1133
64-bit Barrel Shifter	1998(2.02)	108	2108
32-bit Barrel Shifter	1050(1.95)	61	1111
Proposed (Controller + 64-bit Barrel Shifter	764(2.46)	220	984

[13] uses 32-bit width shift register. It needs less non-combinational logic than 64-bit width shift register[11]. But it consumes lots of combinational logic. If we use a 64-bit width Barrel shifter, it needs more non-combinational logic than 32-bit width shift register. It needs, however, less combinational logic than a 32-bit width Barrel shifter. Moreover, By merging the Barrel shifter into the controller as shown in Fig. 7, the combinational logic gates and total gate counts are less than the previous works. The reason is that the common logics in both the controller and shifter such as MUXs for total decoded bits in [11] is reduced. Although unit element is bigger in proposed architecture than others, we can reduce the size of the controller and the Barrel shifter.

3.7 Output buffer

In the final decoding step, coefficients are reordered and stored into output buffer. In proposed architecture, we use one stack to store the Level decoding value. T1s is decoded in this block as well. When decoding Total_zeros, Level and T1s values are reordered. Whenever Run_before is inserted, each coefficient is moved to the corresponding position. The proposed output buffer is depicted in Fig. 7.

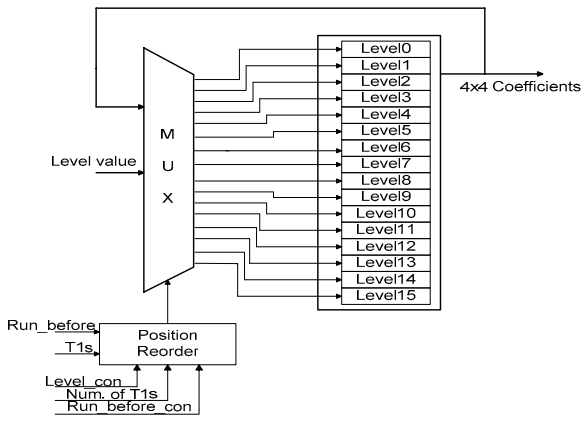

Fig. 7 Proposed Output Buffer Architecture

The proposed architecture is represented in Fig. 8.

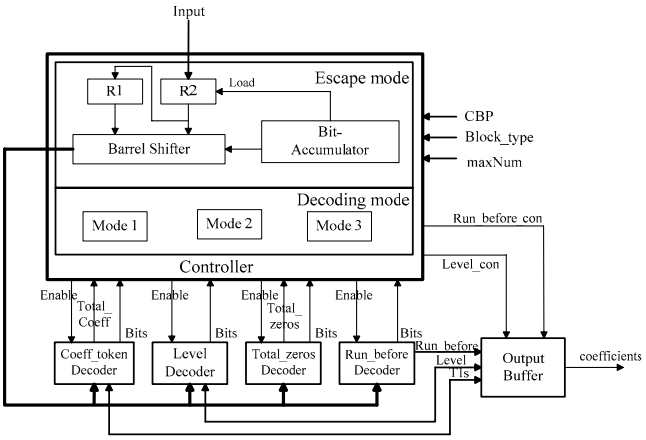

Fig. 8 Proposed CAVLC decoder architecture

4. Verification and Performance Comparison

In order to compare H/W costs, we compare the H/W cost of each block in addition to the H/W cost of overall CAVLC decoder with previous works. In previous works, Level decoder and Run_before decoder use each stack to store Level and Run_before separately. But we combine these stacks into one output stack. We reduce the area of lookup tables which used in Coeff_token decoder, Total_zeros decoder and Run_before decoder as explained in Secion 3. Table. 2 shows the comparison results with [7][13].

The proposed CAVLC decoder is designed and verified in Verilog HDL. We synthesize the proposed architecture by using 0.35μm CMOS standard cell library. The synthesized decoder operates at a frequency of almost 50MHz. Average processing cycle is 79 cycles per MB in case of Akiyo test sequence in CIF image when QP is 28 and only Intra picture is used. If we use Inter and Intra picture together, it needs avg. 66 cycles to decode Akiyo test sequence.

Table 2. H/W cost comparison(gate counts)

	Original	[7]	[13]	Proposed
Coeff_token	901	1037	1825	561
Total_zeros	433	843		235
Run_before	153	227		96
T1s	98	103	102	0
Level	598	662	665	597
Cotroller + Shifter	1982	1884	1786	1002
Output Buffer	3700	3718	2503	2435
CAVLD	7866	8477	6883	4908

[7] proposed the adaptive lookup table to reduce power consumption. [13] proposed Level pipeline architecture and used a combined LUT. As you can see in Table 2, even if we use combined LUT, we can reduce LUT area a little bit. But if we use the proposed LUT, we can reduce gate counts about 58% and 52% compared with [7] and [13], respectively.

In previous works, T1s decoder is used to decode T1s. But we decode T1s in output buffer by using some MUXs. In the proposed architecture, we can reduce the area of T1s decoder. In case of Level decoder, [13] uses modified suffixlength detector which requires more H/W cost.

5. Conclusion

We minimize the H/W cost by using reduced lookup table, performing T1s decoding in the output buffer and combining the controller and the barrel shifter. It needs avg. 79 cycles per MB in CIF image. It also needs 938,520 cycles in 30fps@sec for CIF image and 19,339,200 cycles for 1080HD image. It means that we can decode 1080HD image at 20Mhz frequency.

References

[1]Video Coding for Low Bit Rate Communication, ITU-T Recommendation H.263, Feb 1998

[2]Information Technology-Coding of Audio-Visual Objects-Part 2: Visual, ISO/IEC 14496-2, 1999

[3]Wu Dii, Gao Wen, Hu Mingzeng, Ji Zhenzhou, "A VLSI Architecture Design of CAVLC Decoder", Proc. Intl. Conf. ASIC. pp. 962-965, 2003

[4] Ya-Nan Wen, Guan-Liu Wu, Sao-Jie Chen, and Yu-Hen Hu, "Multiple-Symbol Parallel CAVLC Decoder for H.264/AVC", IEEE ASCCAS, pp.1240-1243, 2006

[5] Yong Ho Moon, Gyu Yeong Kim, Jae Ho Kim, "An Efficient Decoding of CAVLC in H.264/AVC Video Coding Standard", IEEE Transaction, pp. 933-938, 2005

[6]Shau-Yin Tseng, Tien-Wei Hsieh, "A Pattern-Search Method for H.264/AVC CAVLC Decoding", IEEE ICME, pp. 1073-1076, 2006

[7]Heng-Yao Lin, Ying-Hong Lu, Bin-Da Liu, Jarr-Ferr Yang, "Low Power Design of H.264 CAVLC Decoder" IEEE ISCAS, pp.2689-2692, 2006

[8]Hsui-Cheng Chang, Chien-Chang Lin, Jiun-In Guo, "A Novel Low-Cost High-Performance VLSI Architecture for MPEG-4 AVC/H.264 CAVLC Decoding", in Proc. IEEE ISCAS, pp. 6110-6112, 2005

[9]Tsung-Hen Tsa, De-Lung Fang, Yu-Nan Pan, "A HYBRID CAVLD ARCHITECTUR DESIGN WITH LOW COMPLEXITY AND LOW POWER CONSIDERATIONS", in Proc. IEEE ICME, pp. 1910-1913, 2007

[10]Guo-Shiuan Yu, Tian-Sheuan Chang "A Zero-Skipping Multi-symbol CAVLC Decoder for MPEG-4 AVC/H.264"in Proc. IEEE ISCAS, pp. 5583-5586. 2006

[11]Myungseok Oh, Wonjae Lee, Jaeseok Kim, "DESIGN OF HIGH SPEED CAVLC DECODER FOR H.264/AVC",in Proc. IEEE SiPS, pp 325-330, 2007

[12]Mythri Alle, J Biswas, S. K. Nandy, "High Performance VLSI Architecture Design for H.264 CAVLC Decoder", in Proc. IEEE ASAP, pp. 317-322, 2006

[13]Heng-Yao Lin, Ying-Hong Lu, Bin-Da Liu, Jar-Ferr Yang, "A Highly Efficient VLSI Architecture for H.264/AVC CAVLC Decoder", in Proc. IEEE Transaction on multimedia, vol. 10, spp. 31-42, 2008