國　立　交　通　大　學

電子工程學系　電子研究所碩士班

碩　士　論　文

應用於數位電視雙模背景適應性可變長度之編解碼

Context Adaptive Variable Length Coding of Dual Standards for Digital TV

研究生：楊俊彥

指導教授：李鎮宜　博士

中華民國九十五年七月

應用於數位電視雙模背景適應性可變長度之編解碼

研究生：楊俊彥　　　　　　　　　　　　　　指導教授：李鎮宜博士

國立交通大學電子工程學系電子研究所

摘要

在此論文中，首先我們研究了背景適應性可變長度編解碼之特性根據霍夫曼編解碼。因此，根據霍夫曼編解碼的特性，我們提出了一個利用前綴零之個數表格分割法以及藉由算術方式之表格實現，應用於兩種標準的可變長度解碼器，此兩種標準分別為 MPEG-2 以及 H.264/AVC。此一被提出之設計減少了功率消耗以及硬體花費。

再者，考量系統觀點及需求的生產率，我們使用了改良式的以群組為基礎之可變長度編解碼器演算法來實現所提出的可變長度編解碼器系統。除了以群組為基礎的演算法外，我們提出了層次有效率之編解碼、前置零有效率之編解碼、符號構成法以及省略前置零等使用於我們所提出的可變長度編解碼設計之方法。藉由平行化的輸入位元流，我們所提出的可變長度編解碼系統能夠執行即時的編碼及解碼依據所發表的方法。因此，此一被提出的設計能夠滿足訂於 H.264/AVC 主要輪廓中之生產率的需求。

Context Adaptive Variable Length Coding of Dual Standards for Digital TV Applications.

Student : Jiun-Yan Yang Advisor : Dr. Chen-Yi Lee

College of Electric & Computer Engineering

National Chiao Tung University

Abstract

In this dissertation, first we will research the features of CAVLC which is based on Huffman coding. Therefore, based on the features of Huffman coding, we present a VLC decoder for dual standards, MPEG-2 and H.264/AVC, with the PZTP and tables realization with arithmetic method. The proposed design reduces the power consumption and the hardware cost.

Again, from the system view and the requirement of the throughput, we use the improved group-based VLC codec algorithm to realize the proposed VLC codec system. In addition to group-based algorithm, we present other approaches which are level efficient coding, run_before efficient coding, symbols construction and run_before zero-skipping used in the proposed VLC codec design. With the parallel input bitstream, the proposed VLC codec system can execute the real time encoding and decoding based on the proposed approaches. Therefore, the presented design can satisfy the requirement of the throughput specified in H.264/AVC main profile.

Acknowledgments

This dissertation could not have been written without Prof. Chen-Yi Lee who not only served as my supervisor but also encouraged and challenged me throughout my academic program. He and the group leader, Mr. Tsu-Ming Liu patiently guided me through the dissertation process, never accepting less than my best efforts. Besides, Yi-Hong Huang and Kang-Cheng Hou gave me precious advices through this work, and I thank them all.

Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1. Overview of H.264/AVC System

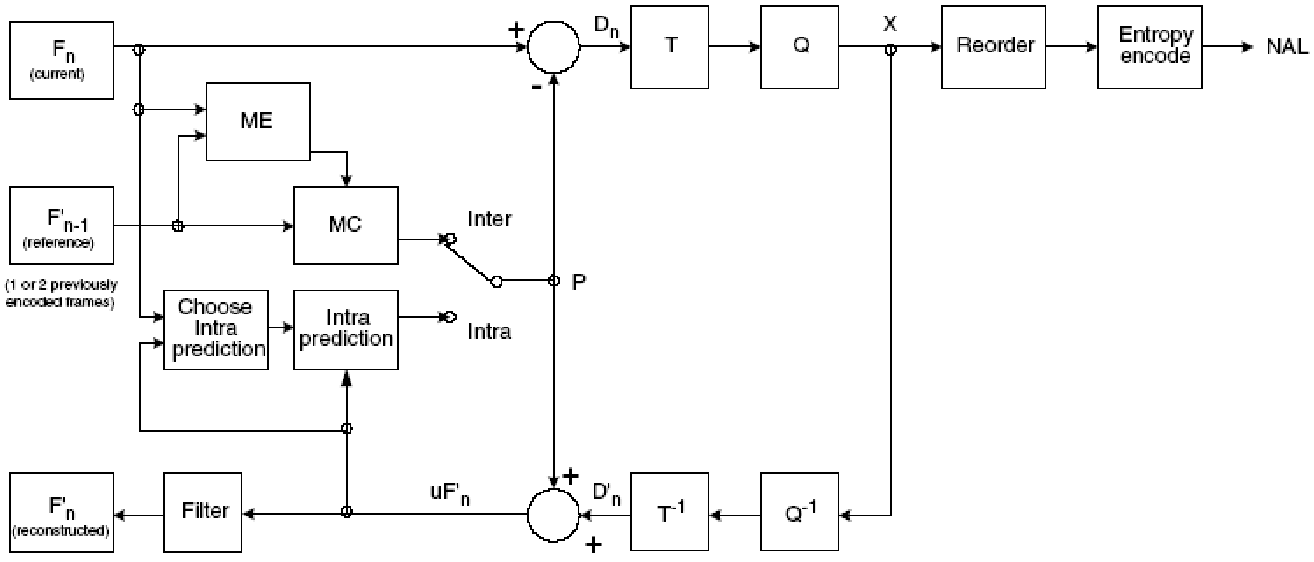


Figure 1-1 : The block diagram of H.264/AVC encoder

Figure 1-1 shows the block diagram of H.264/AVC encoder. When one frame is inputted, first the encoder will do prediction and choose intra or inter prediction according to the input frame type. After the prediction, the original input will subtract the predicted result to get residual data and the residual data will experience discrete-time cosine transform (DCT) and quantization to compress the data transmitted. Finally, entropy encoder will encode the DCT coefficients to bitstream and send the bitstream. Another step to produce F'$_n$ is to make the reference for motion estimation (ME), because in the H.264/AVC decoder this step is to generate the encoded frame. If we want to get the same result in the decoder, we have to use the same reference both in the encoder and decoder. Therefore, we use F'$_{n-1}$ as the reference for ME not F$_{n-1}$.

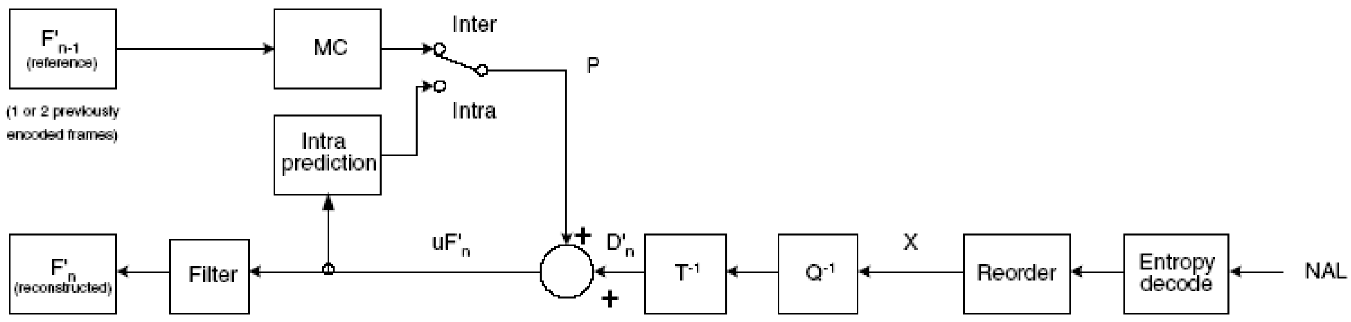


Figure 1-2 : The block diagram of H.264/AVC decoder

Figure 1-2 shows the block diagram of H.264/AVC decoder. As we can see, the architecture of H.264/AVC decoder is much simpler than encoder, because H.264/AVC encoder also has to do decoding process. In H.264/AVC decoder, the input bitstream first is decoded by entropy decoder and the outputs of the entropy decoder is DCT coefficients. Through de-quantization and inverse DCT (IDCT), we can fetch the residual data and finally we add the residual data and the result of MC or intra prediction to get one frame.

Table 1-1 shows the profiles of H.264/AVC standard. These three profiles are basic profiles of H.264/AVC standard. Applications of H.264/AVC cover digital storage media, television broadcasting, and real-time communications. For example, baseline profile targets applications of low bit rates such as multimedia communication and applies portable multimedia players because of its low computation complexity; main profile meets the demand of HDTV due to backup of interlaced content; extended profile contains error resilient tools for the IPTV or multimedia on demand (MOD). However, in those profiles small size of blocks and fixed quantization matrix can't totally hold the image information in high frequency, so H.264/AVC adds Fidelity Range Extensions which contains high profile, high 10 profile, high 4:2:2 profile, and high 4:4:4 profile based on main profile for high definition multimedia applications.

Coding Tools	Profiles		
	Baseline	Main	Extended
I slice	○	○	○
P slice	○	○	○
CAVLC	○	○	○
Slice Group and Adaptive Slice Ordering	○		○
Redundant Slice	○		○
Weighted Prediction		○	○
Interlace		○	
CABAC		○	
SI and SP slice			○
Data Partition			○
B slice		○	○

Table 1-1 : The basic profiles of H.264/AVC standard

From Table 1-1, we can see there are two coding approaches for entropy coding, one is context adaptive variable length coding and the other is context adaptive binary arithmetical coding. Although CABAC has better compression rate than CAVLC, CABAC has extremely more complex structure witch limits the throughput of CABAC than CAVLC. Besides, CAVLC is suitable for all profiles in H.264/AVC system and it has more flexibility for different applications. Therefore, we will further discuss CAVLC in the following sections.

1.2. CAVLC Algorithm

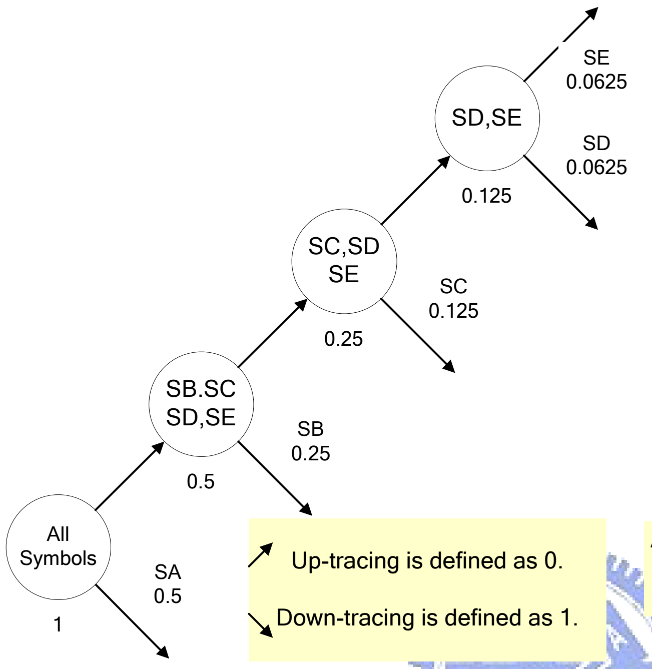
1.2.1. Huffman Coding

Huffman coding uses a specific method for choosing the representations for each symbol, resulting in a prefix-free code (that is, no bit string of any symbol is a prefix of the bit string of any other symbol) that expresses the most common characters in the shortest way possible. It has been proven that Huffman coding is the most effective compression method of this type; i.e. no other mapping of source symbols to strings of bits will produce a smaller output when the actual symbol frequencies agree with those used to create the code. However, for a set of symbols whose cardinality is a power of two and a uniform probability distribution, Huffman coding is equivalent to simple binary block encoding. A Huffman code can be built in the following manner:

- Rank all symbols in order of probability of occurrence.
- Successively combine the two symbols of the lowest probability to form a new composite symbol; eventually we will build a binary tree where each node is the probability of all nodes beneath it.
- Trace a path to each leaf, noticing the direction at each node and define the code for each tracing direction. For example, a ’0’ represents following the left child and a ‘1’ represents following the right child.

An example of building a Huffman tree using binary code is shown in Figure 1-3. We can see that there are 5 symbols, namely SA, SB, SC, SD, and SE. Occurring probability for each symbol is 0.5, 0.25, 0.125, 0.0625, and 0.0625. From the probability of the source symbols, the two smallest probabilities are grouped together and their sum is the substituted probability representing for the original smallest two.

If the branch traces up, it is given the binary code 0. Otherwise, it is given the binary code 1. According to the label (0 or 1) of each branch, we can obtain the variable length codeword of every symbol.



Symbol	Probability	Codeword
SA	0.5	1
SB	0.25	01
SC	0.125	001
SD	0.0625	0001
SE	0.0625	0000

Average bits = 0.5x1 + 0.25x2 + 0.125x3 + 0.0625x4 + 0.0625x4 = 1.875

Figure 1-3 : An example of VLC code construction

For a given frequency distribution, there are many possible Huffman codes, but the total compressed length will be the same. We can see Figure 1-4 for this situation. The example in Figure 1-3 can also be represented by several alternative binary trees. It is possible to define a ‘canonical’ Huffman tree, and that is, pick one of these alternative trees. Such a canonical tree can then be represented very compactly, by transmitting only the bit length of each code. This technique is used in most archives such as PKZIP, LHA, ZOO, ARJ, etc. Huffman coding is optimal when the probability of each input symbol is a power of two. Prefix-free codes tend to have slight inefficiency on small alphabets, where probabilities often fall between powers of two. Expanding the alphabet size by coalescing multiple symbols into “words” before Huffman coding can help a bit.

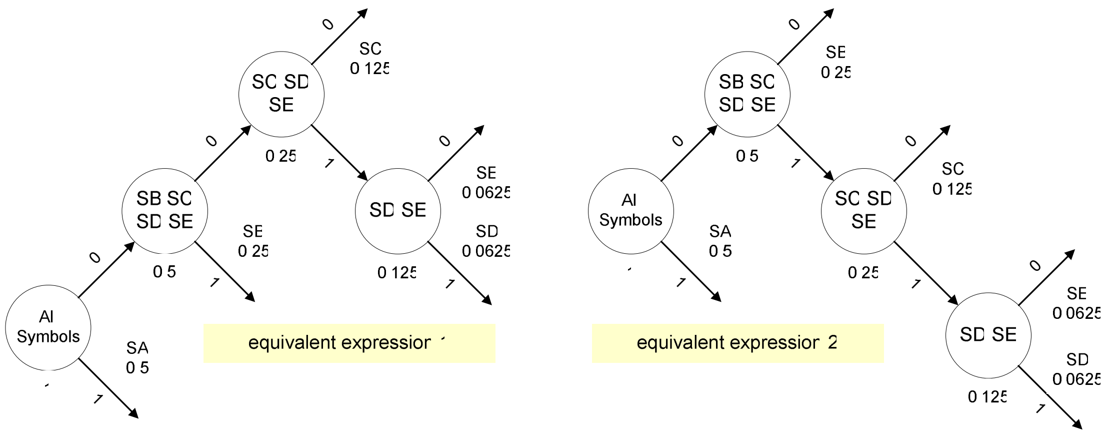


Figure 1-4 : An example of equivalent Huffman trees for Figure 1-3

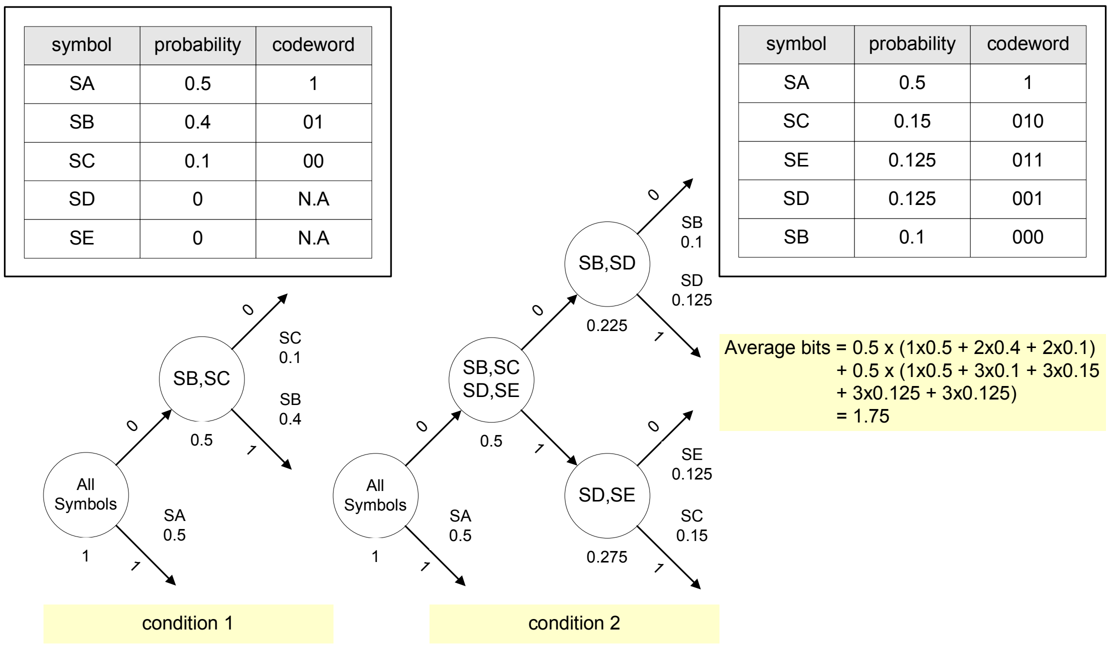
Encoding symbols into bitstream is very simple. We just concatenate the codewords associated with the symbols. For example, if we want to encode SA.SB.SE.SD using the lookup table in Figure 1-3, we just pick the codewords of SA, SB, SD, and SE, which are 1, 01, 0001, and 0000; then concatenate them into 10100000001. If we want to decode 10100000001 back to symbols, we just gave to traverse the binary tree in Figure 1-3 bit by bit through branches to leaf nodes. If a node is encountered, then use the rest of bitstream to traverse from the root of the tree. Keep traversing until there’s no bit left in the bitstream. Traversing the tree, we can decode 10100000001 to be SA, SB, SE, and SD.

1.2.2. Context Adaptive Variable Length Coding

Huffman coding is generally used in various multimedia standards such as MPEG series and JPEG series. CAVLC also adopts Huffman coding as a coding approach but it adds one skill on Huffman coding base. This skill is called “context adaptive” which can bring higher compression ratio than traditional VLC. In above section, the way to calculate the occurring probability of all symbols is under all cases. However, some symbols usually appear under some conditions and seldom appear under other conditions. Therefore, we will build different Huffman codes of one symbol by the occurring probabilities under different conditions. A CAVLC can be built in the following steps:

- ➢ Separate different conditions and get the occurring probabilities of all symbols under all conditions.
- ➢ Rank all symbols in order of probability of occurrence in each condition.
- ➢ Successively combine the two symbols of the lowest probability to form a new composite symbol; eventually we will build a binary tree where each node is the probability of all nodes beneath it.
- ➢ Trace a path to each leaf, noticing the direction at each node and define the code for each tracing direction. For example, a ’0’ represents following the left child and a ‘1’ represents following the right child.

In addition to the first step, the other steps are the same as Huffman code. The purpose of CAVLC is to divide the occurring probability of one symbol in different condition and we can get better compression ratio than traditional VLC. It is sure that more particular description of probability can bring higher code efficiency.



symbol	probability	codeword
SA	0.5	1
SB	0.4	01
SC	0.1	00
SD	0	N.A
SE	0	N.A

symbol	probability	codeword
SA	0.5	1
SC	0.15	010
SE	0.125	011
SD	0.125	001
SB	0.1	000

Average bits = 0.5 x (1x0.5 + 2x0.4 + 2x0.1)
+ 0.5 x (1x0.5 + 3x0.1 + 3x0.15
+ 3x0.125 + 3x0.125)
= 1.75

Figure 1-5 : An example of CAVLC code construction

Figure 1-5 shows an example of CAVLC code construction. The total occurring probabilities of all symbols are the same as the example of Figure 1-3, so the occurring probability of condition 1 and condition 2 is both 50%. Under these two conditions, all symbols have two occurring probabilities, so we will get two code tables to map each symbol. As we mentioned, the only distinction between CAVLC and traditional VLC is the step to divide the conditions, and in each condition there is still Huffman code process. Generally, the way to compare the performance of different coding approaches is to compare the average number of bits. The example of Figure 1-3 gets that the average number of bits is 1.875 and here the average number of bits is 1.75. Although CAVLC has more complex code construction and more VLC tables than traditional VLC, we will achieve the significant improvement of compression rate.

1.3. Designs of CAVLC Encoders and Decoders

1.3.1. Designs of CAVLC Encoders

CAVLC is a lossless coding so the design of CAVLC encoder can’t change the quality of one frame. Therefore, the target of CAVLC encoder design focuses on the performance such as throughput and hardware cost. Table 1-2 shows the maximum throughput requirement of H.264/AVC main profile. Level means the layer of each profile and the range of level in H.264/AVC main profile is 4 to 5.1. Level 4 is the basic demand of main profile and this level can support HD1080i when the frame rate is 30 frames per second. From Figure 1-1 we can observe that the encoding speed of entropy encoder affects the throughput of the entire system greatly. For this reason, the present papers about CAVLC encoder solve the problem of throughput.

Level	4	4.1	4.2/Lo	4.2/Hi	5	5.1
MB/sec	245760	245760	491520	522240	589824	983040

Table 1-2 : Maximum throughput requirement of H.264/AVC main profile

The major two parts of CAVLC encoder are coefficients scanning and symbols encoding. The direct approach to design a CAVLC encoder is to input a set of coefficients and to do the encoding steps serially. Repeating the mentioned steps can easily get the wanted results. However, the maximum number of input coefficients is 16 and encoding symbols has five steps and needs one cycle at least. If we do it serially, the throughput of this simple CAVLC encoder should not meet the requirement.

One way to solve this problem is to deal with scanning coefficients and encoding symbols parallel [1], because there is no dependency between encoding symbols of one block and scanning coefficients of the following block. Therefore, we can execute these two steps parallel and we can improve the encoding throughput.

Another way is to reduce the cycles of encoding symbols because each step of encoding symbols often has multiple cycles. If we send multiple inputs to one step and this step encodes these inputs in one cycle [2]. This method gets better performance than the above one.

1.3.2. Designs of CAVLC Decoders

The discussion about CAVLC decoder is more than encoder because CAVLC decoder has to handle all the bitstream transmitted from H.264/AVC encoder. Great data variation must result much power consumption so power saving of CAVLC decoder is an important issue. Another major issue is the throughput of CAVLC decoder and Table 1-2 shows the throughput requirement of H.264/AVC main profile. Because the input bitstream of CAVLC decoder has dependency on the decoded information, we need some efforts to accelerate the decoding speed of CAVLC decoder.

The major part of CAVLC decoder is also VLC tables and most papers realize those tables by finite state machine (FSM). Build the FSM according to the codeword in the VLC tables and looking up these tables will get the symbols decoded [3]. But directly using the codeword of VLC tables to build the FSM is not efficient in hardware cost and throughput. Furthermore, we have to improve the size of FSM. Separate the VLC tables according the length of the codeword and look up the dividing tables serially and we can build the FSM with the same entries as the VLC tables [4]. This approach achieves lower hardware cost and improves the throughput to support level 4.1. However, if we use some skills such as zero-skipping and multi-symbol, we can get better performance about the throughput [5]. Above papers do not discuss the problem of power consumption. If we make good table partition to control the table switch, we can save the power consumption significantly. In fact many papers proposed many approaches to realize VLC like RAM-based methods [6], [7], but present papers about CAVLC decoder only use ROM-based methods. In fact, we can try more approaches to design CAVLC encoder and decoder.

1.4. Motivation

Recently, human life has been changed greatly by various multimedia applications such as cellular phones, digital cameras, DVD and digital television. But some new technologies like high-definition television (HDTV), blue-ray (BD), and high-definition DVD (HD DVD) appear and will be popular in the future. Therefore, a novel video compression standard, H.264/AVC, can be invoked for these uses because of its high compression rate.

For different demands we have to generate different devices. Therefore when we design a decoder for mobile devices the most important thing is power reduction. The advent of H.264/AVC provides high compression ratio, but there is no backward compatibility to the prevalent MPEG-x and H.264x video coding standards. MPEG-2 and H.264/AVC processors have been reported at ISSCC. However, these solutions used separate modules and only processed a single type of video content in each module. To support different system requirements such as DVB-H or HD-DVD, a scalable pipeline is exploited to efficiently integrate both MPEG-2 and H.264/AVC in a single chip. Besides, we think we can do different table partition from that mentioned above and add other approaches to get more power reduction. Therefore, we propose a VLD with new table partition and realize some tables with arithmetic method.

Furthermore, when our entire system [8], [9] want to provide higher throughput for some applications such as HD 1080, we suffer the entropy decoder can't meet the throughput requirements of H.264/AVC main profile. We have to generate a VLD which can support MPEG-2 and H.264/AVC with enough throughput, and if this VLD can be integrated with context adaptive binary arithmetic decoder (CABAD), that is all we need. We find that CABAD has to use much SRAM for context model and this is a direction to integrate these two entropy decoders. These three decoders, CAVLD, MPEG-2 VLD, and CABAD, in our system should not work at the same time, so we have to make the SRAM with programmability. This approach has been proposed [6], [7], but the approach to divide the groups is not efficient about memory usage and group mapping. From Figure 1-1 and Figure 1-2, we can see that the H.264/AVC encoder also has most parts of H.264/AVC decoder. If we add entropy decoder to the decoding part of the encoder, that is complete H.264/AVC decoder. Therefore, we propose a new group-based VLC codec system adding efficient-coding and zero-skipping to improve the throughput and memory usage.

1.5. Organization of This Thesis

In this thesis, we propose a new low power, table partition VLD for dual standards, a new group-based, high throughput VLC codec system with full programmability for dual standards, and a new soft VLD to handle the error resilient problem. The organization of this thesis is as follows. The overview of CAVLC and the new low power, table partition VLD for dual standards is presented in Chapter 2. The algorithm and architectures of the proposed group-based, high throughput VLC codec system with full programmability for dual standards are described in Chapter 3. The proposed error resilient CAVLD is introduced in Chapter 4. Finally, conclusions and future works are made in Chapter 5.

Chapter 2. A Low Power VLC decoder design

2.1. Overview of CAVLC Encoder and Decoder

2.1.1. Encoding Process Flow

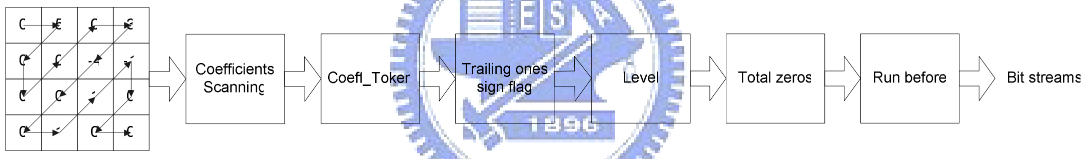
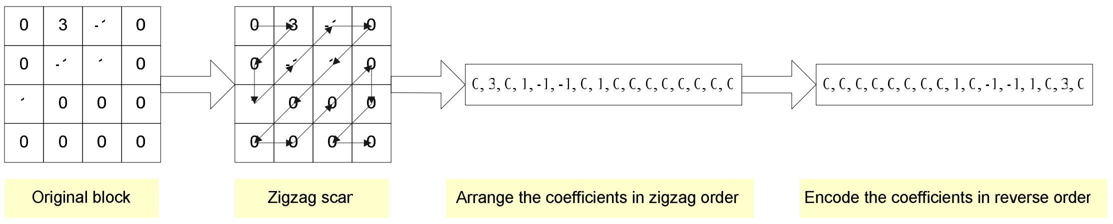


Figure 2-1 : The encoding process flow of CAVLC

Figure 2-8 shows the encoding process flow and the detailed steps are as follows.

➢ When receiving a 2x2 or 4x4 block, the procedure of scanning coefficients will record the symbols to be encoded. There are six symbols which are TotalCoeff, TrailingOnes, trailing_ones_sign_flag, level, total_zeros, and run_before. TotalCoeff is the total number of non-zero coefficients; TrailingOnes is the number of trailing +/-1 and its value should be smaller than four, level is the value of non-zero coefficient; total_zeros is the number of all zeros before the last non-zero coefficient in zigzag-scan order; run_before is the number of zeros before last one non-zero coefficient in zigzag-scan order. Figure 2-2 shows the results derived in coefficients-scanning procedure.

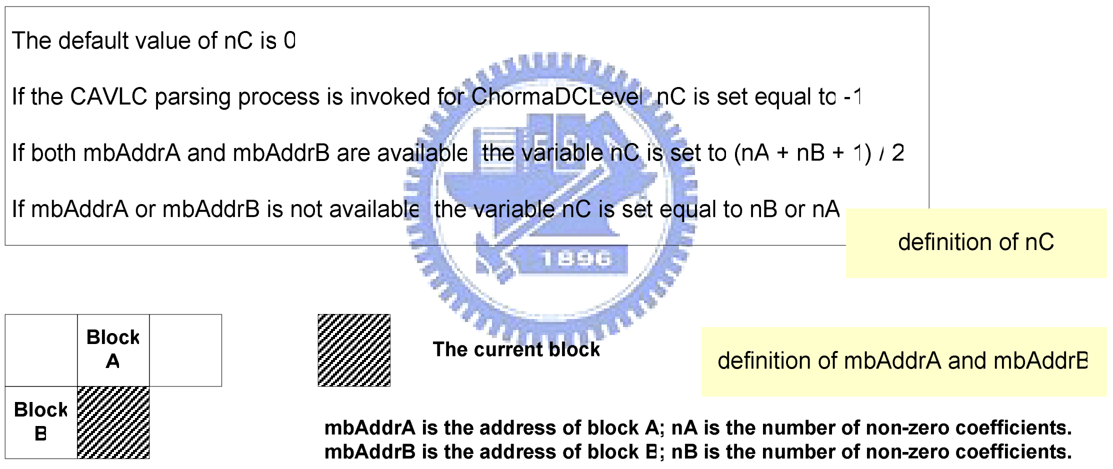


0 3 -´ 0
0 -´ ´ 0
´ 0 0 0
0 0 0 0
Original block
Zigzag scar
C, 3, C, 1, -1, -1, C, 1, C, C, C, C, C, C, C, C
Arrange the coefficients in zigzag order
C, C, C, C, C, C, C, C, 1, C, -1, -1, 1, C, 3, C
Encode the coefficients in reverse order

TotalCoeff	TrailingOnes	trailing_ones_sign _flag	level	total_zeros	run_before
5	3	+ - -	´ 3	3	´ 0 0 ´ ´

Figure 2-2 : An example of CAVLC coefficients scanning

➢ Encode TotalCoeff and TrailingOnes (coeff_token). There are 5 choice of look-up table to use for encoding coeff_token. The choice of table depends on a variable named nC and Figure 2-3 shows how to calculate the value of nC.



The default value of nC is 0

If the CAVLC parsing process is invoked for ChormaDCLevel nC is set equal to -1

If both mbAddrA and mbAddrB are available the variable nC is set to (nA + nB + 1) / 2

If mbAddrA or mbAddrB is not available the variable nC is set equal to nB or nA

mbAddrA is the address of block A; nA is the number of non-zero coefficients.
mbAddrB is the address of block B; nB is the number of non-zero coefficients.

Figure 2-3 : How to calculate the value of nC

➢ Encode the sign of each trailing one in reverse order.

➢ Encode level in reverse order and there are 7 VLC tables to choose from, Level_VLC0 to Level_VLC6.

➢ Encode total_zeros.

➢ Encode run_before.

Table 2-1 lists the result of encoding the example in Figure 2-2 and the transmitted bitstream for this block is 000010001110010111101101.

Element	Value	Code
coeff_token	TotalCoeff = 5, TrailingOnes = 3	0000100
T1 sign (4)	+	0
T1 sign (3)	-	1
T1 sign (2)	-	1
Level (1)	+1	1
Level (0)	+3	0010
total_zeros	3	111
run_before (4)	zerosLeft = 3; run_before = 1	10
run_before (3)	zerosLeft = 2; run_before = 0	1
run_before (2)	zerosLeft = 2; run_before = 0	1
run_before (1)	zerosLeft = 2; run_before = 1	01
run_before (0)	zerosLeft = 1; run_before = 1	No code required; last coefficient

Table 2-1 : The result of encoding the example in Figure 2-2

2.1.2. Decoding Process Flow

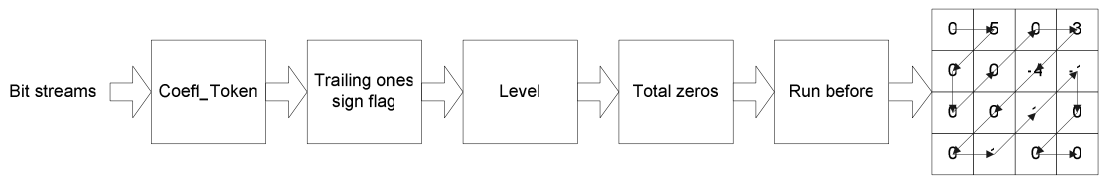


Figure 2-4 : The decoding process of CAVLC

Figure 2-4 shows the decoding process flow of CAVLC and we can see that the decoding procedures are similar to the encoding steps. The only difference is decoding process does not do coefficients scanning and the other steps do decoding bitstream instead of encoding symbols. Table 2-2 shows an example of CAVLC decoding and the final output array is 0, 3, 0, 1, -1, -1, 0, 1.

Code	Element	Value	Output array
0000100	coeff_token	TotalCoeff = 5, TrailingOnes = 3	Empty
0	T1 sign	+	1
1	T1 sign	-	-1,1
1	T1 sign	-	-1,-1,1
1	level	+1	1,-1,-1,1
0010	level	+3	3,1,-1,-1,1
111	total_zeros	3	3,1,-1,-1,1
10	run_before	1	3,1,-1,-1,0,1
1	run_before	0	3,1,-1,-1,0,1
1	run_before	0	3,1,-1,-1,0,1
01	run_before	1	3,0,1,-1,-1,0,1

Table 2-2 : An example of CAVLC decoding from the result of Table 2-1

2.2. Overview of the Proposed Architecture

Figure 2-5 shows the functional diagram of the proposed architecture of the CAVLC decoder. As introduced in section 2.1.2, there are five major parts to decode the symbols. In order to support MPEG-2 VLC decoding, we construct the MPEG-2 VLC tables in coeff_token part, because the two decoding procedures have similar decoding manner. This part will be described in later section. The prefix-zero buffer and the bitstream buffer are used for the table partition and table realization with arithmetic method. The coeffNum is to calculate the right position in the coefficient buffer of the present level in level buffer. For power reduction issue, all function units are controlled by enable signals, because they must not work at the same time. There is also a hold signal for prefix-zero buffer to avoid counting the zeros not belong to prefix zeros. If there is no enable signal or hold signal to control the function unit, it should result the power dissipation.

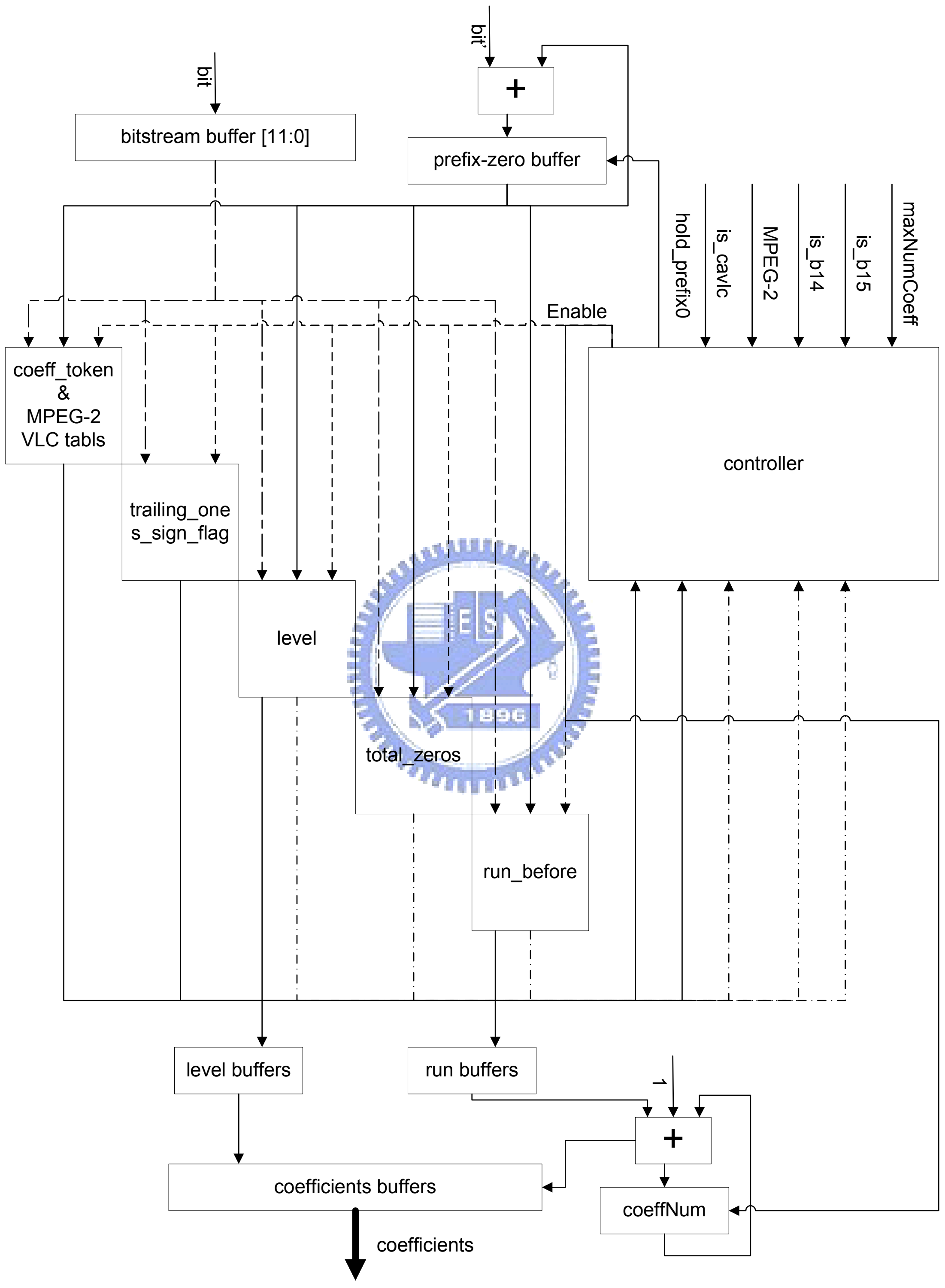
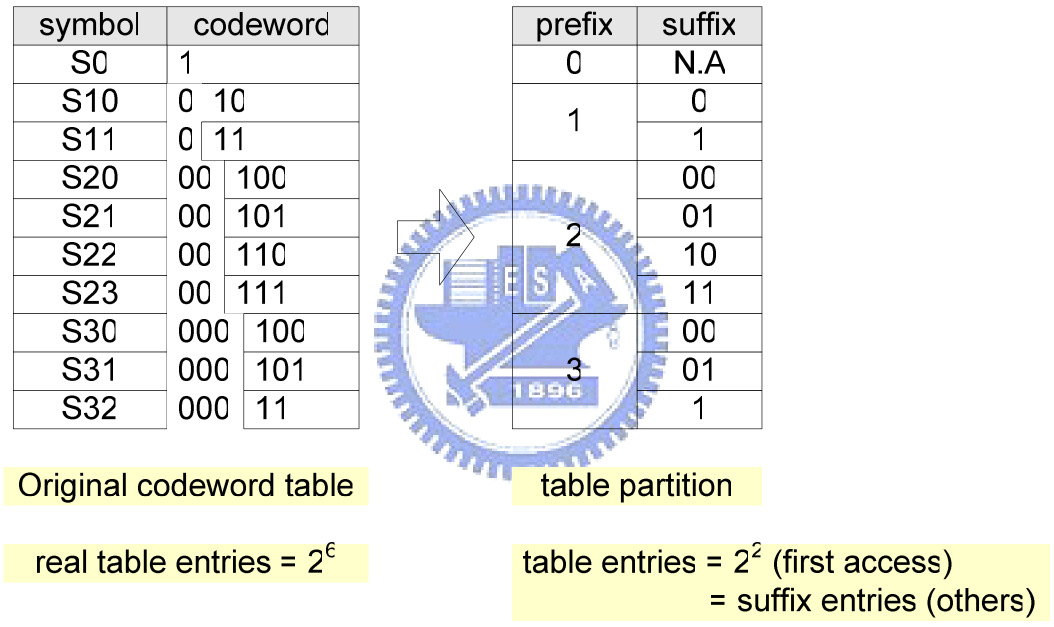


Figure 2-5 : Overview of the proposed low power architecture

2.3. Table Partition

In VLSI design, the efficient method to reduce dynamic power consumption is to decrease the data switching. However, most designs of the CAVLC decoder use FSM to look up the VLC tables. As long as the input bitstream to access the look-up table changes frequently, that must cause much power dissipation. Besides, the alteration in large look-up table must dissipate more power than the same one in small look-up table. Therefore, good table partition will reduce the size of look-up table and the data switching to decrease power consumption.



symbol	codeword
S0	1
S10	0 10
S11	0 11
S20	00 100
S21	00 101
S22	00 110
S23	00 111
S30	000 100
S31	000 101
S32	000 11

prefix	suffix
0	N.A
1	0
	1
2	00
	01
	10
	11
3	00
	01
	1

Original codeword table
table partition
real table entries = 2^6
table entries = 2^2 (first access)
= suffix entries (others)

Figure 2-6 : An example of proposed table partition

Figure 2-6 shows an example of the proposed table partition. Although the original codeword table has only 10 entries, the longest length of the codeword is 6, so we have to build a look-up table with 32 entries for this codeword table by FSM method. That is, the longest length of the codeword dominates the entries of the codeword table not the real entries. However, if we adopt the proposed table partition method to build the look-up table, the entries of the first time to access the table are 4, and other entries are equal to the relative suffix entries. Because this approach divide the tables according to the prefix zeros, we call it prefix-zero table partition (PZTP).

When we access the look-up table with PZTP every cycle, the searching entries are much smaller than the original entries. If the longest length of the codeword is larger, the difference between the searching entries with PZTP and the original entries is greater.

The way to build the look-up table is as follows:

- ➢ According to the leading zeros we call prefix, build the first layer of look-up table like prefix item in Figure 2-6.
- ➢ Build the second layer of look-up table by suffix which is the codeword except the leading zeros and the first 1.

The steps to look up the VLC tables are as follows:

- ➢ We count the leading zeros until the first 1 appears, and choose the relative suffix table by prefix.
- ➢ We look up the suffix table by the input bitstream, and find symbols needed.

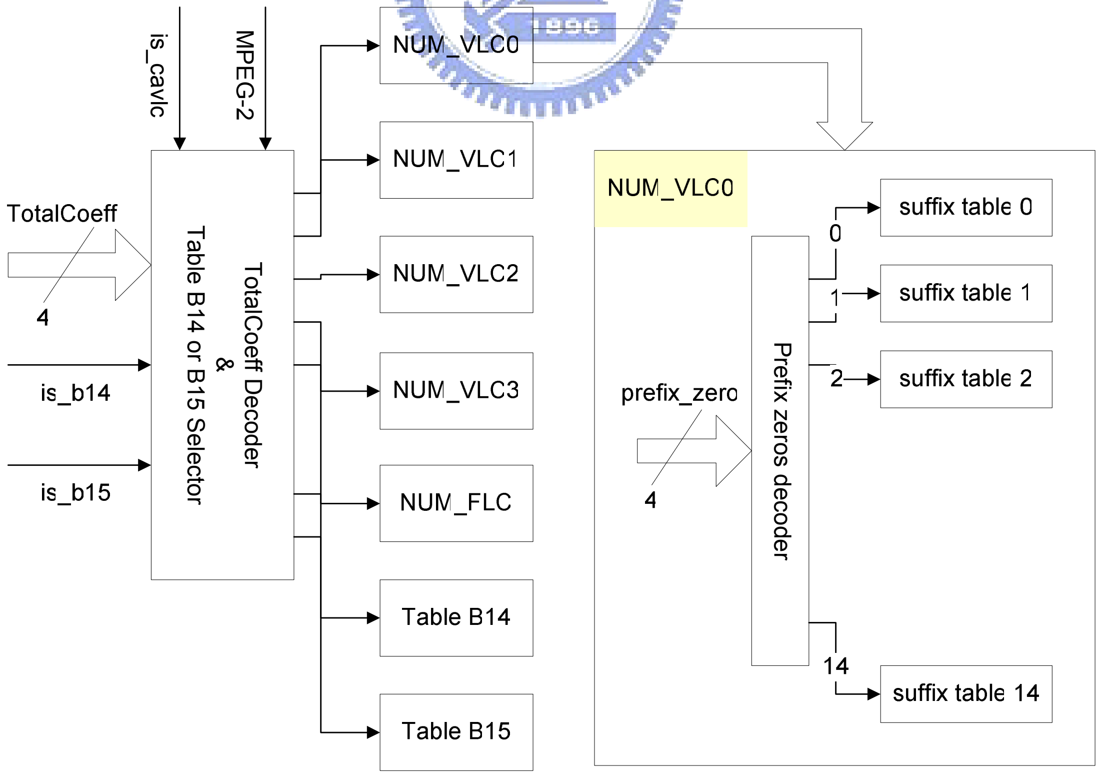


Figure 2-7 : The PZTP VLC decoder architecture of coeff_token

Figure 2-7 shows the PZTP VLC decoder (VLD) architecture of coeff_token. There are five tables of CAVLD, NUM_VLC0, NUM_VLC1, NUM_VLC2, NUM_VLC3, and NUM_FLC and the other two tables, Table B14 and Table B15, belong to MPEG-2 VLD. The implementation of NUM_FLC will be introduced in the next section. First, if both the two enable signals, is_cavlc and MPEG-2, are not active, the entire PZTP VLD will be shut down to avoid the dynamic power dissipation due to the data switching. If either of them is active, the controller (TotalCoeff Decoder & Table B14 or B15) will open only one of those tables for power issue. Of course, the two signals should not be active at the same time.

Assume that we are executing H.264/AVC decoding. Even if the present decoding procedure is coeff_token, the enable signal, is_cavlc, will not be active at the beginning. To avoid unnecessary power consumption, we set the enable signal to be active, only when we receive the first one of the codeword or the boundary of prefix. Therefore, when receiving prefix, only accumulator consumes power. When executing MPEG-2 VLD, we do the same thing.

From Figure 2-5, we put the value of prefix in prefix-zero buffer. When we begin receiving suffix of codeword and looking up the suffix table, the value of prefix is fixed. Therefore, we can consider the output of prefix zeros decoder in Figure 2-7 as an enable signal of the relative suffix table in the process of looking up the suffix table. At this time, the searching entries of the entire codeword table are equal to the entries of the suffix table. The most entries of coeff_token are 8 and those of MPEG-2 VLD are 16.

PZTP takes advantage of the feature of Huffman coding to decrease the data switching when accessing the look-up table, and the hardware cost of the VLC tables. Besides, another advantage is easy to implement, so total_zeros and run_before also adopt this method to implement in the proposed CAVLD.

2.4. Table Realization with Arithmetic Method

2.4.1. NUM_FLC of coeff_token

The length of all the codeword in this look-up table is 6, and the total entries of this table are 62. If we build the table by FSM method, this idea seems good. However, if we analyze the relationship between the codeword and the symbols, we will find some arithmetic rules.

TotalCoeff	TrailingOnes	codeword
0	0	0000 11
0	1	0000 00
1	1	0000 01
0	2	0001 00
1	2	0001 01
2	2	0001 10
0	3	0010 00
1	3	0010 01
2	3	0010 10
3	3	0010 11

TotalCoeff	codeword[5 2]	TrailingOnes	codeword[1 0]
0	0000	0	11
0	0000	1	00
1	0000	1	01
0	0001	2	00
1	0001	2	01
2	0001	2	10
0	0010	3	00
1	0010	3	01
2	0010	3	10
3	0010	3	11

Figure 2-8 : An example of NUM_FLC

Figure 2-8 shows an example of NUM_FLC. The left table is the original table of NUM_FLC and we can derive the right table after we separate the codeword. We can find the following arithmetic relationship except the first row, and this formula exists in NUM_FLC distinctly. Although the first row of NUM_FLC doesn't fit this rule, only prefix of the codeword map to the symbols is 4.

$$TotalCoeff = codeword[5:2]+1$$
$$TrailingOnes = codeword[1:0]$$

Figure 2-9 shows the proposed architecture of NUM_FLC. Due to the power consideration, we only access this part when we receive the sixth bit of the codeword. Based on this method, we can easily change the look-up table into and reduce much hardware cost and power consumption.

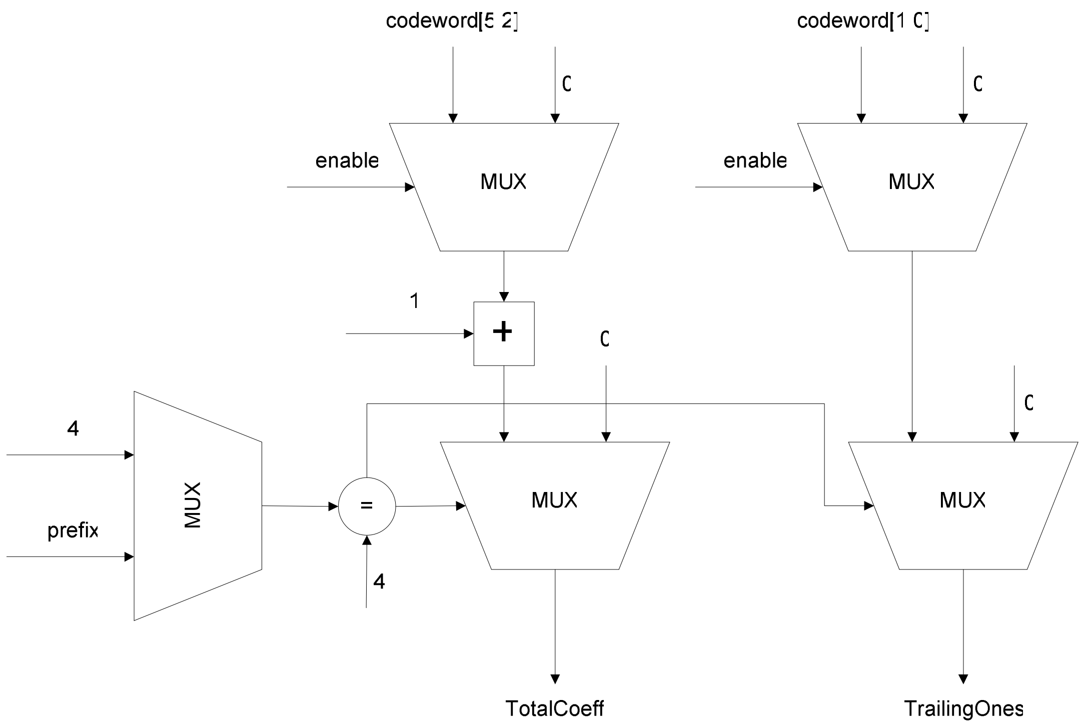


Figure 2-9 : The architecture of proposed NUM_FLC

2.4.2. Level Decoding

Basically, level coding is constructed by seven VLC tables which are VLC0 to VLC6. However, if we implement the level decoder with VLC tables, it costs much hardware and power. The reason is the longest length of codeword is 28, prefix is 16 and suffix is 12. Even if we use PZTP to construction the VLC tables of level decoder, they are still huge VLC tables. For the low power demand, we have to use another method to realize the level decoder, and here we implement it by arithmetic approach which algorithm is specified in [10].

Figure 2-10 shows the algorithm of level decoding. In fact, suffixLength is to decide the VLC tables to choose from. According to this algorithm, if we pipeline the level decoding and suffixLength well, we can use the minimum number of function units to decode level. However, we can get good performance about the power and hardware cost.

```
level_prefix
levelCode = (level_prefix << suffixLength)
if (suffixLength > 0 || level_prefix >= 14)
{
        level_suffix
        levelCode += level_suffix
}
if (level_prefix == 15 && suffixLength == 0)
        levelCode += 15
if (first_level && TrailingOnes < 3)
        levelCode += 2
if (levelCode % 2 == 0)
        level = (levelCode + 2) >> 1
else
        level = (-levelCode - 1) >> 1
```

level decoding

```
if (TotalCoeff > 10 && TrailingOnes < 3)
        suffixLength = 1
else
        suffixLength = 0
```

Decoding level

```
if (suffixLength == 0)
        suffixLength = 1;
if (|level| > (3 << (suffixLength - 1)) && suffixLength < 6)
        suffixLength++
```

suffixLength

```
level_prefix = leading 0s

level_suffix = bitstream [levelSuffixSize-1 : 0]
```

```
if (level_prefix == 15)
        levelSuffixSize = 12
else if (level_prefix == 14 && suffixLength == 0)
        levelSuffixSize = 4
else
        levelSuffixSize = suffixLength
```

Figure 2-10 : Algorithm of level decoding

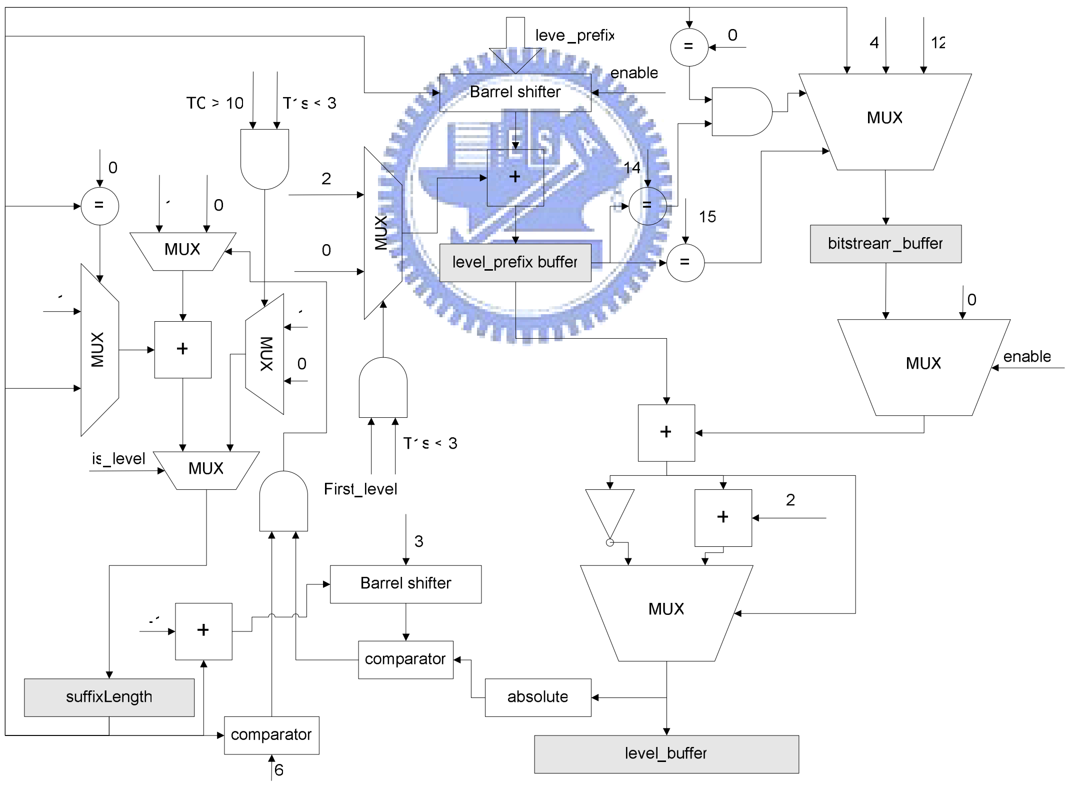


Figure 2-11 : The proposed architecture of level decoding

Figure 2-11 shows the proposed architecture of level decoding. There are two major parts, the left part is to calculate the suffixLength and the right part is to decode the codeword of level. The gray rectangles represent the registers. The size of level_prefix buffer is 10 bits, bitstream buffer uses 12 bits which is shared by all modules, and suffixLength needs 3 bits to save the value. The level_prefix is the number of leading zeros derived by the leading zeros counter shown in Figure 2-5 which is shared by four decoding modules, coeff_token, level, total_zeros, and run_before. The barrel shifter to rearrange the level_prefix works, only when we receive the first one of the codeword of level. Besides, it also handles the special case when level_prefix is 15 and suffixLength is 0. That helps us not to add additional 15 to levelCode, so it shortens the critical path of level decoding and reduces the hardware cost. The whole architecture of level decoding is also controlled by an enable signal which turns off level decoding when we execute another procedure. That inverter is to do the step, (-levelCode - 1), and according to 2's complement -levelCode is equal to (~ levelCode + 1), so the formula, -levelCode – 1, is equal to (~ levelCode + 1 - 1), that is ~levelCode.

The part to calculate suffixLength is also needed even if we implement level decoding with look-up table. As we mentioned above, the method of table searching depends on suffixLength to choose the correct VLC table, so this part is not omissible in any approach of level decoding. Therefore, our contribution is to simplify the VLC tables with arithmetic method, and the effect is pretty good.

2.5. Summary

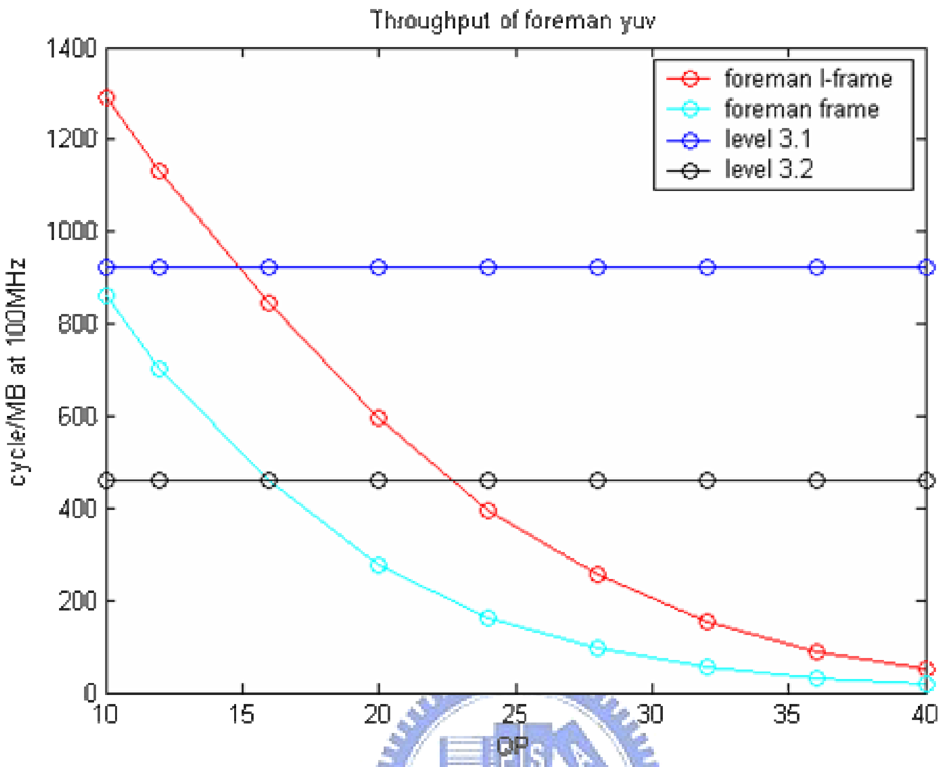


Figure 2-12 : The throughput of foreman.yuv with the proposed VLD

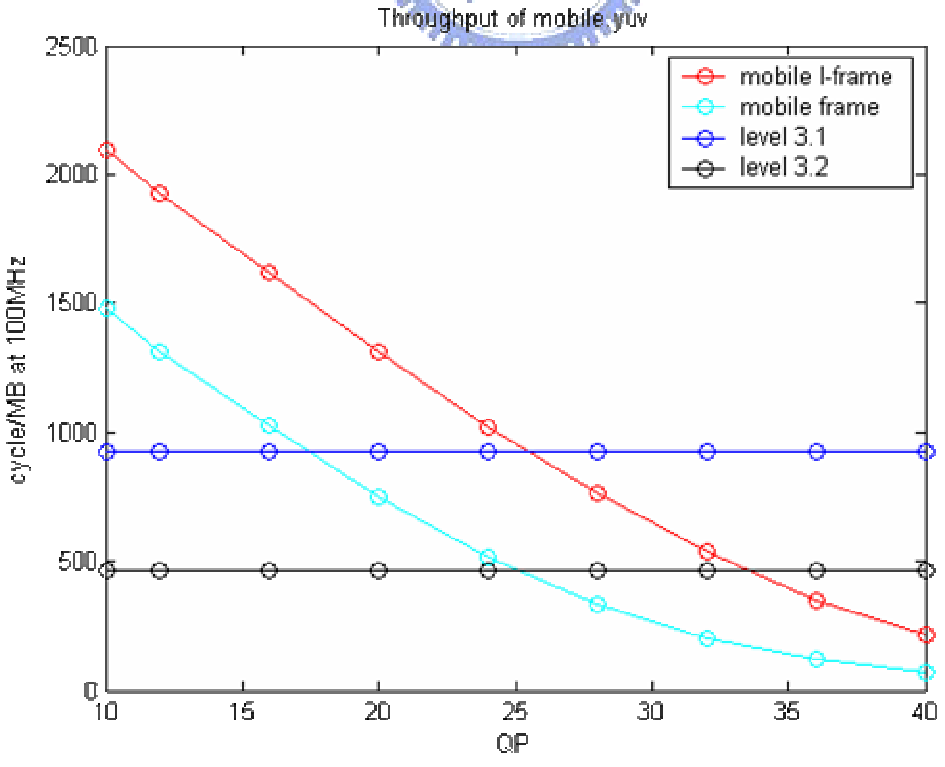


Figure 2-13 : The throughput of mobile.yuv with the proposed VLD

Figure 2-12 and Figure 2-13 show the throughput of two pictures with the proposed VLD. The simulation environment is JM 9.2 which C code of H.264/AVC system. We set nine different values of QP to get the simulation results. In the two figures, the blue line is the throughput requirement of baseline@3.1 specified in H.264/AVC standard when the clock frequency is 100MHz and the black one is for baseline@3.2. In Figure 2-12, the throughput of foreman meets the requirement of baseline@3.2 when QP is 20 and that of I-frame in the same picture also meets that standard when QP is 28. In Figure 2-13, the throughput of mobile meets the demand when QP is 28. Therefore, the proposed design can support H.264/AVC baseline.

	[3]	[4]	Proposed Design
Tech.	0.25 um	0.18 um	0.18 um
Gate-count	6100	4720	CAVLC : 3267 MPEG2 : 945
Target Spec.	Baseline Profile	Main Profile @4.1	Main Profile @4.2 & MPEG-2
Buffer	N.A.	696 bits RAM	3471 gate-count
Clock Constraint	125 MHz	125 MHz	125 MHz

Table 2-3 : Hardware cost evaluation of proposed low power design

Table 2-3 shows the comparison of the hardware cost. Although we show the throughput of two pictures in Figure 2-12 and Figure 2-13 when the clock frequency is 100MHz, the maximum speed of the proposed design is 180MHz under a 0.18um CMOS technology. The performance is fast enough for meeting the real-time processing requirement of CAVLC decoding on main profile @4.2. Compared to the design proposed by [4], The CAVLC part of the proposed design reduce 30% hardware cost, and the total design still has less hardware cost. The proposed design doesn't use RAM as storage due to the power saving.

Spec.	MPEG-2 I-frame	H.264 I-frame	H.264 P-frame
power (mW)	1.719	1.302	1.376

Table 2-4 : The post layout power consumption under 0.18um CMOS Tech.

Table 2-4 shows the post layout power consumption under 0.18um CMOS technology. The proposed design can provide extremely low power, and it is used in our dual-standard system [8], [9].

Chapter 3. A VLC Codec System for dual standards

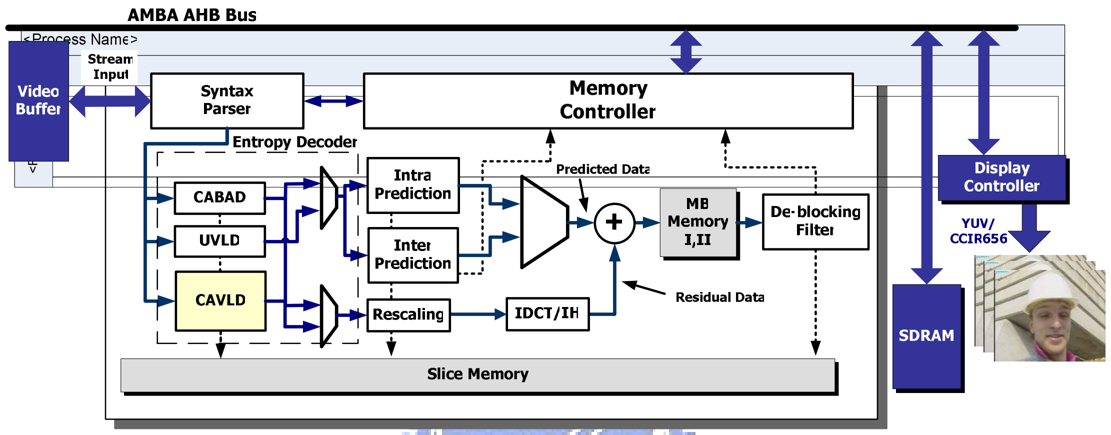


Figure 3-1 : The architecture of our proposed system

Figure 3-1 shows the architecture of our proposed system for H.264/AVC main profile. The entropy decoder contains CABAD, UVLD, and CAVLD. UVLD and CAVLD are the same choice for entropy decoder, and UVLD is used to decode the syntax parser, and CAVLD is for residual data. Therefore, the output of UVLD is to control the decoding mode of H.264/AVC decoder, and the results of CAVLD are the DCT coefficients of residual data. After IDCT, the data will be added with the predicted data to complete a unit block.

In Figure 3-1, CABAD has to use slice memory to store the context model and row-storage. Figure 3-2 shows the usage of memory of CABAD in our proposed H.264/AVC decoder system. The context model of CABAD uses 349.1 bytes memory of the slice memory.

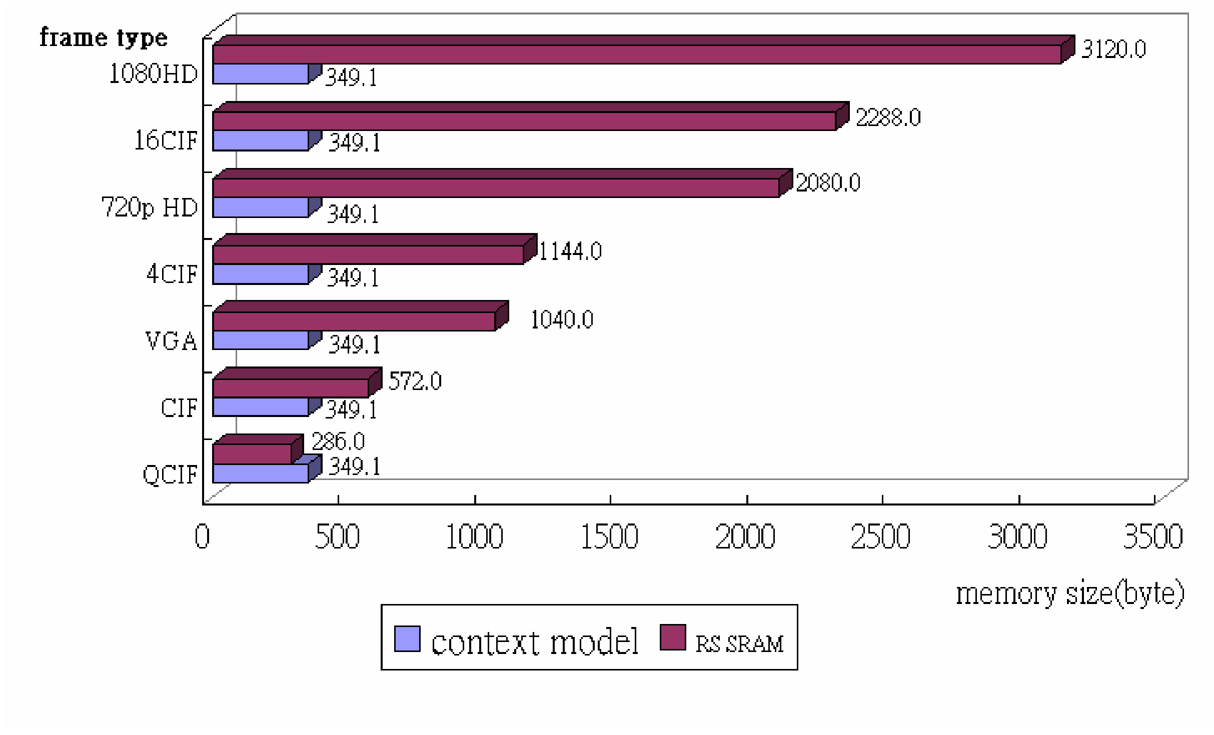


Figure 3-2 : The usage of memory of CABAD in our proposed H.264/AVC decoder

The context model of CABAD uses much memory, so that is an idea to integrate CABAD and CAVLD. The used memory can provide a space to store the VLC tables of CAVLD, and our proposed H.264/AVC decoder receive parallel input of bitstream, so we have to try another approach to implement CAVLD. Besides, as mentioned in my motivation, if we add the CAVLC encoder into the entropy decoder, that can be integrated with H.264/AVC encoder to a H.264/AVC codec system. Therefore, we try to find a method to implement a VLC codec system based on memory. and finally we proposed a new group-based VLC codec system reference to [6] and [7].

3.1. The Architecture of the Proposed VLC Codec System

Here, we will describe the architecture of the proposed VLC codec system. We will focus on the design of CAVLC encoder/decoder, and not to express the MPEG-2 VLC codec in detail. That is because the major difference of the proposed MPEG-2

VLC codec is the group-based algorithm and hardware implementation, and other parts basically are similar to the conventional VLC codec design. Therefore, about the MPEG-2 VLC codec system, we only discuss the proposed group-based alteration, and we will pay attention to the CALVC encoder/decoder.

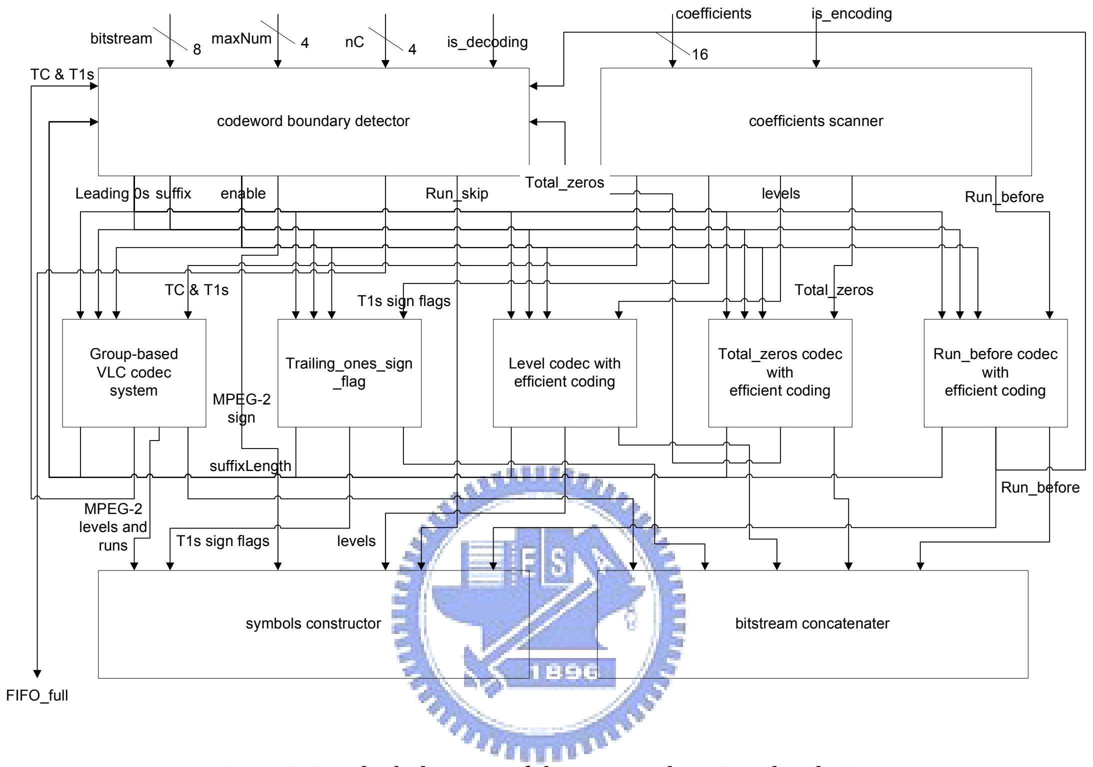


Figure 3-3 : Block diagram of the proposed VLC codec design

The block diagram of the proposed VLC codec design is shown in Figure 3-3. To fit specification of our proposed H.264/AVC decoder system, the input bitstream is parallel input and its length is 8 bits. The decoder is controlled by the enable signal, is_decoding, so is the encoder. The maxNum is to decide the block type which is being decoded or encoded, and nC is introduced in 2.1 to choose the correct VLC table for coeff_token. The serial input data, coefficients, is the DCT coefficient for the encoder in reverse order. The codeword boundary detector has a FIFO to store the input bitstream, and the output signal, FIFO_full, represents whether the bitstream FIFO is full or not. The symbols constructor will send out the results of DCT coefficients arranged and the bitstream concatenater handles the link of the encoded codeword. The illumination of the components is as follows.

- ➢ The major functions of the codeword boundary detector are counting the leading ones and zeros, and fetching the demanded suffix for the each decoding function unit by the recorded bitstream boundary. Besides, it is also a controller to decide the activity of each decoding component, and it has to calculate the number of skipped run_before and then send the information to symbols constructor. For MPEG-2 VLC, it has to detect the special case such as escape mode and end of block.

- ➢ After coefficients scanner receive the serial input data, DCT coefficients, it calculates and sends the necessary data for each encoding component. When doing MPEG-2 VLC encoding, it only counts the levels and runs. After sending the MPEG-2 level and run, it can receive the following coefficients. The more information is needed for CAVLC encoding, and this unit should calculate TotalCoeff, TrailingOnes, T1s flags, levels, and run_befores. Different from MPEG-2 process, coefficients scanner has to receive all coefficients of one block, and then it can begin requesting the coefficients of the next encoding block.

- ➢ Group-based VLC codec system uses the proposed group-based VLC codec algorithm to implement MPEG-2 and CAVLC coeff_token encoder/decoder. Besides, it contains the NUM_FLC of CALVC coeff_token and MPEG-2 escape case. The detailed design contribution will be described in the following section.

- ➢ Trailing_ones_sign_flag encodes and decodes the signs of all trailing ones.

- ➢ Level codec with efficient coding handles the information about levels. The detail of efficient coding will be expressed in the next chapter.

- ➢ Total_zeros codec with efficient coding deals with the coding process of total_zeros.

- ➢ Run_before codec with efficient coding encodes and decodes the run_befores to get the wanted results.

- ➢ The symbols constructor is used for decoding process. It arranges the decoded levels by the decoded runs. In CAVLC decoding process, it works at the same time when decoding run_before to increase the decoding throughput.
- ➢ The bitstream concatenater collects the encoded bit streams and links them. The first step it receives the codeword value and length to assemble the bitstream belonging to each encoding process. Then, it concatenates the separate bit streams to transmitted bitstream.

The decoding procedure of CAVLC decoder has to decode the bitstream step by step, because the bit streams have data dependency. If we don't get some decoded information, we can't do the next step. Therefore, the important thing to increase the decoding throughput is to reduce the decoding cycles for each component. The CAVLC decoding steps are as follows:

- ➢ Counting the leading zeros until detecting the first one of the input bitstream, and then sends the leading zeros and suffix to group-based VLC codec system. If nC is the value of NUM_FLC, we only send suffix.
- ➢ Decoding the coeff_token according to group-based VLC algorithm. The component outputs the suffix length to calculate the used bitstream boundary.
- ➢ After decoding the coeff_token, we will get TrailingOnes that can help us decide suffix length transmitted to Trailing_ones_sign_flag. When decoding Trailing_ones_sign_flag, we also count the leading zeros belong to level decoding process.
- ➢ At the same time to decode levels, we count the leading zeros of level decoding or total_zeros. When the number of decoded level is equal to TotalCoeff, we have to quit decoding level.
- ➢ When decoding total_zeros, we count the leading zeros used for some run_before symbol and the leading ones for zero skipping.

➢ When decoding run_before, we still count the leading zeros used for the next run_before symbol and the leading ones for zero skipping. Then, according the previous decoded run_before, we can begin arranging the DCT coefficients into the correct position in the decoded block. When the zerosLeft is equal to 0 or the last run_before is decoded, the run_before process has to end.

The encoding process of CAVLC encoder doesn't have so many steps, although we can design the encoding process like the way of decoding procedure. However, we consider the throughput of the CAVLC encoder is worse, if we execute the encoding process with the serial steps. We observe that there is no data dependency between the encoded symbols for different encoding component, so we can do the encoding steps parallel. For example, even if coeff_token step doesn't finish, we can still execute level encoding step, because the data for level encoding step doesn't depend on the results of coeff_token encoding step. Therefore, when executing encoding process, all components of our proposed design will work together. The design idea is to increase the encoding throughput, because the throughput of the proposed CAVLC encoder design depends on the most cycles of encoding step instead of the sum of cycles cost by all encoding components.

In order to support the proposed encoder design, how to design a bitstream concatenater is important. The bitstream concatenater has to link the encoded codewords as fast as possible. We don't hope we save the cycles of encoding process, but we take more efforts to concatenate the encoded codewords. Therefore, this design will be described in Chapter 4, and here we first introduce the proposed VLC group-based codec system.

3.2. Conventional Group-based VLC Codec System

This work is previously developed and verified by Bai-Jue Hsieh in [6], [7]. The intention of this section is to quickly give us a sense of what a conventional group-based VLC Codec system is and how it works.

3.2.1. Definition of Codeword Groups

An example of Huffman code and codeword grouping is illustrated in Figure 3-4. Based on this result, the conventional codeword group is a set of codewords whose source symbols are combined to perform the Huffman procedure and receive the same codeword length. According to this definition, the codeword groups have the following properties:

- In a group, the codeword can be treated as a binary number which is codeword length-bit long, called VLC_codenum, since the codeword length is the same.
- The codeword that has the smallest VLC_codenum in a group is denoted VLC_mincode.
- A VLC_codeoffset is the offset value between the VLC_mincode and the VLC_codenum.

VLC table	
C1	01
C2	100
C3	101
C4	1101
C5	1110
C6	1111
C7	0000
C8	0001
C9	0010

Group	symbol	prefix	suffix	VLC _codenum	VLC _codeoffset	Is VLC _mincode?
G0	C7	00	00	0	0	yes
	C8	00	01	1	1	
	C9	00	10	2	2	
G1	C1	01	N A	1	0	yes
G2	C2	10	0	4	0	yes
	C3	10	1	5	1	
G3	C4	11	01	13	0	yes
	C5	11	10	14	1	
	C6	11	11	15	2	

Figure 3-4 : Example of VLC table and codeword groups

In Figure 3-4, the symbols C4, C5, and C6 belong to the codeword group G3. In this group, the codewords have the same codeword length, 4-bit, and the prefix 11_2. The word length of the suffixes is 2-bit. Therefore, the 4-bit VLC_codenums are13, 14, and 15; the VLC_mincode is 4’b1101; and the 2-bit VLC_codeoffsets are 0, 1, and 2. Source symbols that are not combined will belong to different groups, such as C7, C8, and C9 in G0, and C4, C5, and C6 in G3, although codeword lengths are identical. Moreover, there is only one symbol in group G1 since C1 is the only VLC having length of 2 bits.

3.2.2. Intra-Group Decoding Procedure

Besides grouping codewords, mapping symbols onto memories and extraction codeword group information are necessary for VLC decoding. The memory address of a symbol in a group is calculated by the VLC_codeoffset of the symbol and the base address of the VLC_mincode in that group; i.e. the symbol address is the sum of the VLC_codeoffset and the base address of the group. After applying this arithmetic relationship, decoded symbol address can be found by numerical calculation rather than by pattern matching. Thus, the group information to be stored is composed of codeword length, VLC_mincode, and base address. Based on the group information in Figure 3-5, intra-group decoding/encoding procedure is performed as follows.

Assuming we are decoding codeword 10011_2.

- VLC_codeoffset = VLC_codenum(10011_2) – VLC_mincode(10000_2) = 00011_2 =3;
- symbol_address = VLC_codeoffset(3) + base_address(50) = 53;
- the decoded symbol C4 is retrieved from memory address 53;

Assuming the encoded symbol address is 103.

- VLC_codeoffset = symbol_address (103) – base_addresss (100) = 3;
- VLC_codenum = VLC_codeoffset (3) + VLC_mincode (32) = 35;
- The encoded 8-bit codeword is 00100011_2 = 35.

symbol	prefix	suffix	VLC _codenum	VLC _codeoffset	Symbol address
C1	10	000	16	0	50
C2	10	001	17	1	51
C3	10	010	18	2	52
C4	10	011	19	3	53
C5	10	100	20	4	54
C6	10	101	21	5	55
C7	10	110	22	6	56

Group Information : codeword length = 5
VLC_mincode = 10000_2
base address = 50

Figure 3-5 : Example of intra-group symbol memory mapping and group information

3.2.3. Group-searching Scheme

An economical group-searching scheme with high operation rate and low complexity determines the performance of a group-based VLC decoder because the decoding procedure is performed after the group information is obtained. We use inter-group symbol memory mapping and Pseudo-Constant-Length-Code (PCLC) in order to achieve such a group-searching scheme. If all codeword lengths are the same , the numerical properties of codewords in a group can be applied to the whole coding table. We apply a procedure, namely PCLC procedure, to equalize codeword lengths by adding redundant binary digits, 00…0, behind VLC codewords. Therefore, PCLC codewords, which have the same length as the longest VLC codeword, can be treated as binary numbers, PCLC_codenums.

group	symbol	PCLC _codeword	PCLC _codenum	symbol address	PCLC _codeoffset	is PCLC _mincode
G0	S00	**00100100**	36	0	0	o
G0	S01	**00100101**	37	1	1	
G0	S02	**00100110**	38	2	2	
G0	S03	**00100111**	39	3	3	
G1	S10	**001100**00	48	4	0	o
G2	S20	**0110**0000		5	0	o
G2	S21	**0111**0000		6	1	
G1	S11	**011111**00	56	7	3	
..	..	……..	….	..	….	..
..	..	……..	….	..	….	..

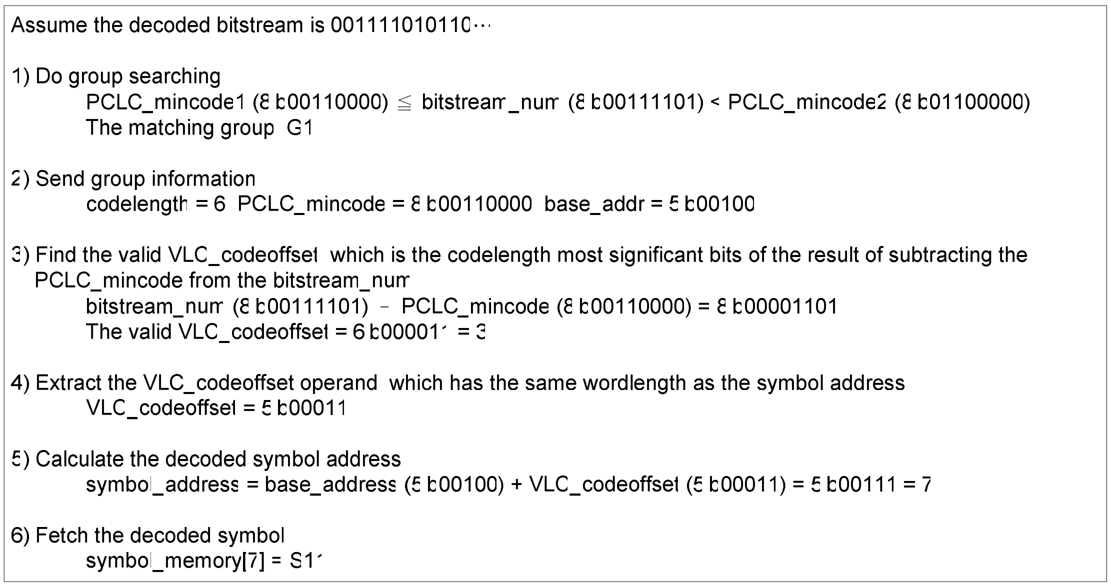
Table 3-1 : Example of inter-group symbol memory mapping

Group	Valid	codelength	PCLC_mincode	base address
G0	1	8	**00100100**	0
G1	1	6	**001100**00	4
G2	1	4	**0110**0000	5

Table 3-2 : Group information for Table 3-1

It is easily to distinguish PCLC codewords and PCLC_codenums from each other because the VLC code is a prefix code. As a result, a PCLC table is established with PCLC_codenums placed in ascending order, i.e. $\text{codenum}_0 < \text{codenum}_1 < \ldots < \text{codenum}_n$. This results in ascending PCLC_mincodes as well, i.e. $\text{mincode}_0 < \text{mincode}_1 < \ldots < \text{mincode}_n$. Based on the PCLC table, the base addresses have to be assigned in PCLC_mincode order, i.e. $\text{base\_addr}_0 < \text{base\_addr}_1 < \ldots < \text{base\_addr}_n$, for inter-group symbol memory mapping. An example of the PCLC table and its intra/inter-group symbol memory mapping is shown in Table 3-1, and the group information of this PCLC table is given in Table 3-2, where the valid bit indicates whether the group information is used. We can see in Table 3-1 that G2 is inserted in the middle of G1. This placement is specialized for decoding to save memory space of symbol memory.

According to PCLC tables and symbol memory maps, the conventional decoding group searching scheme is realized by applying numerical properties to bitstream and symbol addresses. Similar to PCLC codewords, a decoded bitstream that has the same length as the PCLC codewords is treated as a binary number, bitstream_num. Because the bitstream is a sequence of concatenated codewords, such as codeword_i – codeword_j – etc, a relation between the bitstream and the PCLC table can be expressed by $\text{PCLC\_codenum}_i \leqq \text{bitstream\_num} <$ numerical comparisons. The decoded codeword belongs to group G_x when the hit condition, $\text{PCLC\_mincode}_x \leqq \text{bitstream\_num} < \text{PCLC\_mincode}_{x+1}$, is encountered. Let’s see the process of decoding one symbol from bitstream “001111010110…”



Assume the decoded bitstream is 001111010110…

1) Do group searching

    PCLC_mincode1 (8'b00110000) ≦ bitstream_num (8'b00111101) < PCLC_mincode2 (8'b01100000)

    The matching group: G1

2) Send group information

    codelength = 6 PCLC_mincode = 8'b00110000 base_addr = 5'b00100

3) Find the valid VLC_codeoffset, which is the codelength most significant bits of the result of subtracting the PCLC_mincode from the bitstream_num

    bitstream_num (8'b00111101) – PCLC_mincode (8'b00110000) = 8'b00001101

    The valid VLC_codeoffset = 6'b000011 = 3

4) Extract the VLC_codeoffset operand, which has the same wordlength as the symbol address

    VLC_codeoffset = 5'b00011

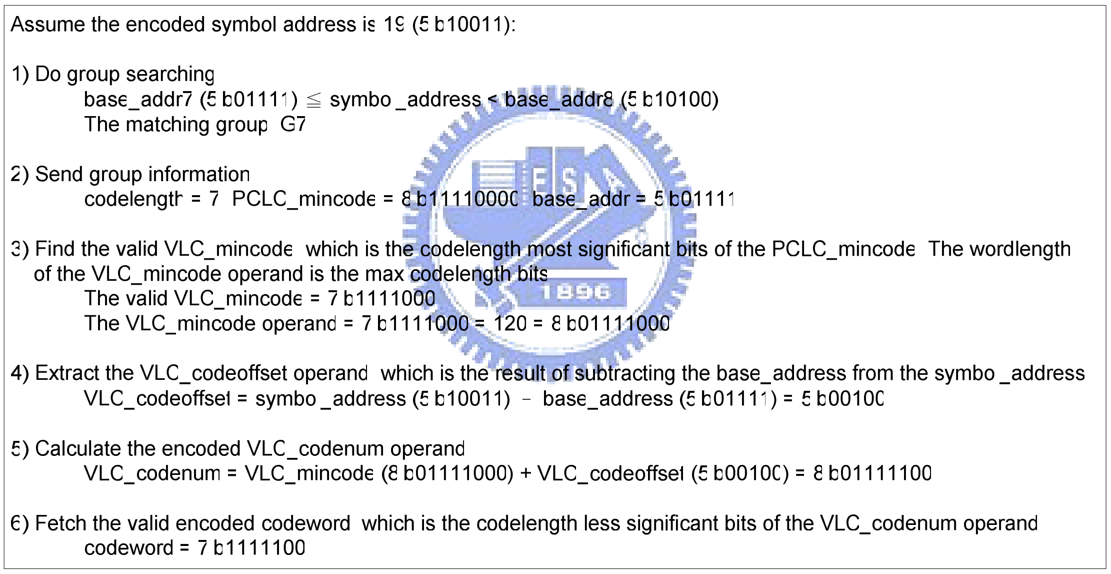
5) Calculate the decoded symbol address

    symbol_address = base_address (5'b00100) + VLC_codeoffset (5'b00011) = 5'b00111 = 7

6) Fetch the decoded symbol

    symbol_memory[7] = S11

Figure 3-6 : Process of decoding a symbol



Assume the encoded symbol address is 19 (5'b10011):

1) Do group searching

    base_addr7 (5'b01111) ≦ symbol_address < base_addr8 (5'b10100)

    The matching group: G7

2) Send group information

    codelength = 7 PCLC_mincode = 8'b11110000 base_addr = 5'b01111

3) Find the valid VLC_mincode, which is the codelength most significant bits of the PCLC_mincode. The wordlength of the VLC_mincode operand is the max codelength bits

    The valid VLC_mincode = 7'b1111000

    The VLC_mincode operand = 7'b1111000 = 120 = 8'b01111000

4) Extract the VLC_codeoffset operand, which is the result of subtracting the base_address from the symbol_address

    VLC_codeoffset = symbol_address (5'b10011) – base_address (5'b01111) = 5'b00100

5) Calculate the encoded VLC_codenum operand

    VLC_codenum = VLC_mincode (8'b01111000) + VLC_codeoffset (5'b00100) = 8'b01111100

6) Fetch the valid encoded codeword, which is the codelength less significant bits of the VLC_codenum operand

    codeword = 7'b1111100

Figure 3-7 : Process of encoding a symbol address

According to the relation between PCLC tables and the symbol address, the conventional encoding group searching scheme is realized by applying numerical properties to codewords and symbol addresses. Based on the encoded symbol, the relative symbol address can be fetched. A relation between the symbol address and the PCLC table can be expressed by base_addr$_i$ ≦ symbol address < numerical comparisons. The decoded codeword belongs to group G$_y$ when the hit condition, base_addr$_y$ ≦ bitstream_num < base_addr$_{y+1}$, is encountered. Let's see the process of encoding one symbol from the symbol address “19 (5’b10011)”

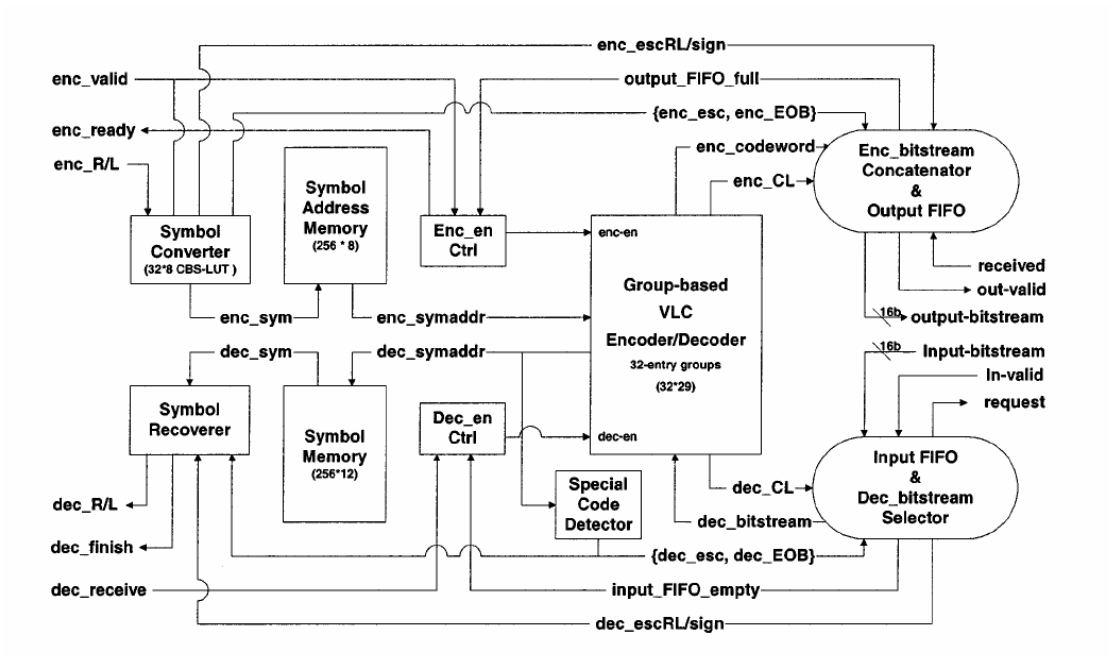


Figure 3-8 : Block diagram of conventional group-based VLC decoder architecture

The conventional VLC codec system is designed for MPEG applications with coding tables up to 256-entry 12-bit symbols and 16-bit codewords. This system performs concurrent encoding and decoding procedures by accessing the same group information and achieves table programmability by loading data into on-chip memories. To complete the VLC codec processes of MPEG videos, this design includes the operations of sign bits and escaped run-levels (escRL) following VLC codewords. By the efficient symbol conversion, the memory requirement is reduced to (25x8 + 28x8 + 28x12 + 32x29) bits for a CBS-LUT, a symbol address memory, a symbol memory, and 32-entry group-information. Block diagram of the conventional VLC codec system is shown in Figure 3-8. It mainly consists of the following components.

- ➢ The group-based VLC encoder/decoder is composed of group detectors and combinational logic circuits to realize the VLC codec processes.

- ➢ The input FIFO stores the input bitstream. According to previous decoded results, the Dec_bitstream selector transmits codewords bitstream to the VLC decoder. Besides, this selector detects sign bits and escRLs when VLC codewords are decoded.

- The Enc_bitstream concatenater adds sign bits or escRL's behind VLC codewords and concatenates encoded results into a single bitstream. Then, every 32 bits of the encoded bitstream in the concatenater is shifted into the Output FIFO.
- The special code detector recognizes special codes, such as escape and EOB, by checking decoded symbol addresses instead of decoded symbols. Without waiting for symbol fetching, this detector can determine the length of the additional bits following a VLC codeword. Hence, the next codeword bitstream can be found by the Dec_bitstream selector immediately and the decoding throughput can be increased.
- The Enc_en and Dec_en Ctrls determine the operations of the VLC_encoder and decoder according to the condition of input data and FIFOs.
- Both symbol address and symbol memories are the on-chip memory modules for storing symbol information.
- The symbol converter performs symbol conversion and detects escaped RLP's and EOB symbols. On the other hand, the symbol recoverer finds correct runs and signed levels based on decoded results.

3.3. The Proposed Group-Based VLC Encoding and Decoding

3.3.1. The Definition of Decoding Codeword Groups

group	symbol	prefix	suffix	suffix_num	suffix _offset	attribute
G2	S3	001	N.A.	N.A.	0	suffix_min
G3	S4	0001	00	0	0	suffix_min
	S5	0001	01	1	1	
	S6	0001	1	2	2	
G4	S7	00001	00	0	0	suffix_min
	S8	00001	01	1	1	
	S9	00001	1	2	2	
G5	S10	000001	00	0	0	suffix_min
	S11	000001	01	1	1	
	S12	000001	10	2	2	
	S13	000001	11	3	3	

Table 3-3 : An example of CAVLC code and codeword grouping

An example of CAVLC code and codeword grouping is illustrated in Table 3-3. CAVLC code is also constructed based on Huffman code, and as we introduce it in Chapter 1, Huffman code is a prefix code, that is any codeword is not the prefix code of other codewords. For example, the symbol, S3, listed in Table 3-3 is relative to the codeword, 001, and in the entire VLC codeword table there is no codeword which starts as 001 except the codeword of S3. Based on the result, the proposed codeword group is a set of codewords whose source symbols are combined to receive the same number of leading zeros. Besides, the number of the group is equal to the relative the number of leading zeros. For example, when the number of leading zeros is 5, the relative group number is also 5. This is very useful to simplify the process of group searching. According to this definition, the codeword groups have the following properties.

- In a group, the suffix of the codeword can be treated as a codeword length-bit binary number, called suffix_num, since the prefix length is the same regardless to the suffix length.
- The codeword that has the smallest suffix_num in a group is denoted suffix_min.
- A suffix_offset is the offset value between the suffix_min and the suffix_num.

Difference from the conventional codeword groups, we set the codeword groups only based on prefix, because the prefix of codewords is unique. We don't have to use the entire codeword to set the groups, so we can save some cost in the process of calculating the group information and building the tables in memories. In Table 3-3, the symbols S10, S11, S12, and S13 belong to the codeword group G5. In this group, the codewords have the same prefix length, 6-bit. The suffix length is 2-bit. Therefore, the 2-bit suffix_nums are 0, 1, 2, and 3, the suffix_min is 2'b00 which is the same as 1'b0, and the 2-bit suffix_offsets are 0, 1, 2, and 3. Symbols which are not combined belong to different groups, such as S7, S8, and S9 in G4, and S4, S5, and S6 in G3. Besides, there is only one symbol in group G3 since symbol S3 completes the Huffman procedure alone.

In CAVLC coeff_token decoding/encoding process, there are four tables used. The number of total groups with the proposed codeword groups is 46, and we have to use 64-entry memories to build the whole CAVLC coeff_token table. Therefore, under the reasonable usage of memory, we can set the first group of NUM_VLC0 is G0, that of NUM_VLC1 is G16, that of NUM_VLC2 is G32, and G48 is for NUM_VLC3. From the distribution of the codeword groups, we can easily get the address of group memory based on the NUM_VLC$_x$ and the codeword group.

In MPEG-2 table B15, some codewords also have the leading ones as the prefix. Based on the rule of the proposed grouping method, we only set the codeword groups according to the leading ones, and then set the leading zeros codeword groups along the leading ones codeword groups. The number of the groups constructed by leading ones is 8, so the beginning group by leading zeros is 8. An example of the Huffman code and codeword grouping is shown in Table 3-4. The number of total groups in MPEG-2 VLC tables with the proposed grouping method is 30. However, we locate the groups of MPEG-2 table B14 from G0 to G31, and we put the groups of MPEG-2 table B15 from G32 to G63, when we combine CAVLC coeff_token tables and MPEG-2 VLC tables. The reason is to get the group number easily under reasonable memory usage, because CAVLC coeff_token has to use 64-entry memories. In order to complete the proposed grouping method, we have to get the information of the number of leading ones and the leading-one prefix or leading-zero one.

group	symbol	prefix	suffix	suffix_num	suffix _offset	attribute
G3	S3	110	0	0	0	suffix_min
	S4	110	1	1	1	
G4	S5	1110	00	0	0	suffix_min
	S6	1110	01	1	1	
	S7	1110	10	2	2	
	S8	1110	11	3	3	
G9	S21	001	01	1	1	suffix_min
	S22	001	10	2	2	
	S23	001	11	3	3	
G10	S24	0001	00	0	0	suffix_min
	S25	0001	01	1	1	
	S26	0001	10	2	2	
	S27	0001	11	3	3	

Table 3-4 : An example of Huffman code and codeword grouping in MPEG-2 table B15

3.3.2. The Definition of the Encoding Symbol Groups

In the conventional group-based VLC codec system, there was no special definition of the encoding symbol groups. It used CBS-LUT for the encoding process to look up the base address of the symbol address memory. However, if we can define the encoding symbol groups, we will get some benefits about the usage of memories.

group	run	level	encoding_num	encoding_offset	attribute
G0	0	1	1	0	encoding_min
	0	2	2	1	
	0	…	…	…	
	0	31	31	30	
G1	0	32	0	0	encoding_min
	0	33	1	1	
	0	…	…	…	
	0	40	8	8	
G5	4	1	4	0	encoding_min
	5	1	5	1	
	…	1	…	…	
	31	1	31	27	

Table 3-5 : An example of MPEG-2 encoding symbol groups

An example of MPEG-2 encoded symbols and symbol grouping is shown in Table 3-5. According to conventional CBS-LUT method, when run is equal to 0, the number of mapping levels is 40, but when run is greater than 7, the number of mapping levels is less than 3. Finally, one run will map to one memory address. Therefore, we consider that we can also define the encoding symbol groups like decoding codeword groups to save the usage of memories. In the symbol grouping procedure, we count the number of the mapping symbols for one symbol. For example, when the value of run is equal to 0, the number of mapping levels is 40. Perhaps, when the value of level is equal to 1, the number of mapping runs is 28. Based on the result of this procedure, the proposed symbol groups are a set of the most symbols mapping to one symbol.

According to this definition, the symbol groups have the following properties.

- In a group, the value of the changed symbol is called encoding_num, since the other symbol is fixed.
- The value of the changed symbol has the smallest encoding_num in a group is denoted encoding_min.
- An encoding_offset is the offset value between the encoding_min and the encoding_num.

In Table 3-5, the levels from 1 to 31 belong to the symbol groups G0. In this group, the runs have the same value, 0. Therefore, the encoding_nums are from 1 to 31, the encoding_min is 1 and the encoding_offsets are from 0 to 30. Source symbols which are not combined will belong to different groups. When the value of the level is equal to 1, the runs from 4 to 31 belong to G5. In symbol groups for MPEG-2 VLC tables, the group, G1, is particular to other groups. The value of the fixed symbol, run, is the same as G0, but we still separate it to another group. The reason is if we combine G0 and G1 as a symbol group, we will use more memories.

group	run	level
G0	0	1~31
G1	0	32~40
G2	1	1~18
G3	2	1~5
G4	3	1~4
G5	4~31	1
G6	4~16	2
G7	4~6	3

Table 3-6 : The symbol groups for MPEG-2 VLC tables

The total symbol groups for MPEG-2 VLC tables are shown in Table 3-6, and both table B14 and B15 use the symbol groups, because the symbols of them are the same.

The similar approach is also used in CAVLC coeff_token encoding process. However, the procedure of symbol grouping in CAVLC is much easier than that in MPEG-2. TrailingOnes is the only choice to be the reference of group number, because each TrailingOnes can map most number of TotalCoeffs. The symbol groups of CAVLC coeff_token are shown in

group	TrailingOnes	TotalCoeff
G0	0	0 ~ 16 or 0 ~ 4
G1	1	1 ~ 16 or 1 ~ 4
G2	2	2 ~ 16 or 2 ~ 4
G3	3	3 ~ 16 or 3 ~ 4

Table 3-7 : The symbol groups of CAVLC coeff_token

3.3.3. Intra-Group Decoding Procedure

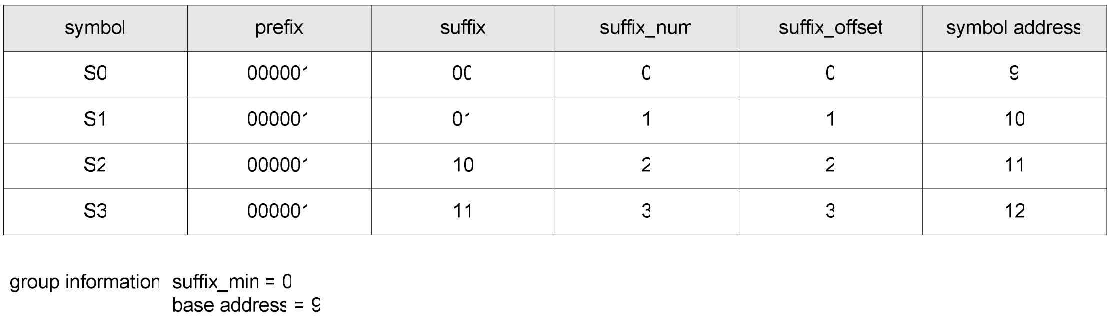
In addition to grouping codewords, it is necessary for decoding procedures to map symbols onto memories and extract codeword group information. During intra-group symbol memory mapping, the memory address of a symbol in a group is calculated by the suffix_offset of this symbol and the base address which denotes the symbol address of the suffix_min of the group. In other words, the symbol address is the sum of the suffix_offset and the base address. After applying this arithmetic relation, suffix_offsets, decoded symbol addresses, and encoded codewords can be found by numerical calculations rather than pattern matching. Therefore, the group information to be stored is suffix_min, and base addresses.

Based on the memory map and the group information in Figure 3-9, intra-group decoding procedures can be described as follows.

Decoding procedure – assume the decoded codeword is $(0000\_0110)_2$:

- ➢ suffix_offset = suffix_num $(10)_2$ – suffix_min (0) = 10_2 = 2;
- ➢ symbol_address = suffix_offset (2) + base_address (9) = 11;

- The decoded symbol, S2, is accessed by the symbol_address, 9;

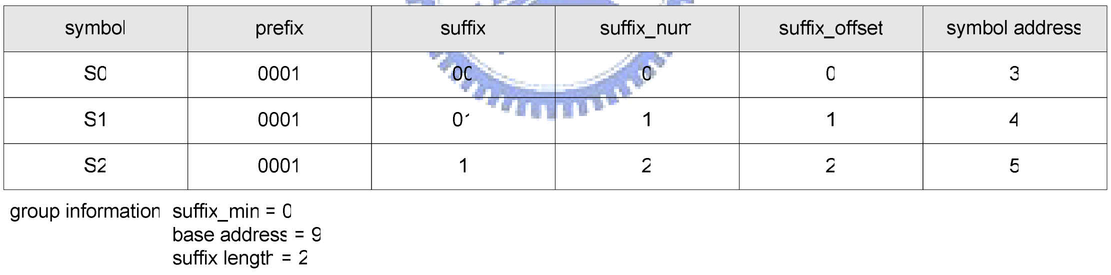


symbol	prefix	suffix	suffix_num	suffix_offset	symbol address
S0	000001	00	0	0	9
S1	000001	01	1	1	10
S2	000001	10	2	2	11
S3	000001	11	3	3	12

group information suffix_min = 0
base address = 9

Figure 3-9 : An example of intra-group memory map and group information

However, when calculating the suffix_offset, we have to get the value of suffix_num. That means we have to know the real suffix length of the relative suffix. Therefore, the group information also has suffix length to fetch the correct suffix. Besides, the suffix length has to be sent back to codeword boundary detector, and the codeword boundary detector can truncate the codewords which are decoded without any error.

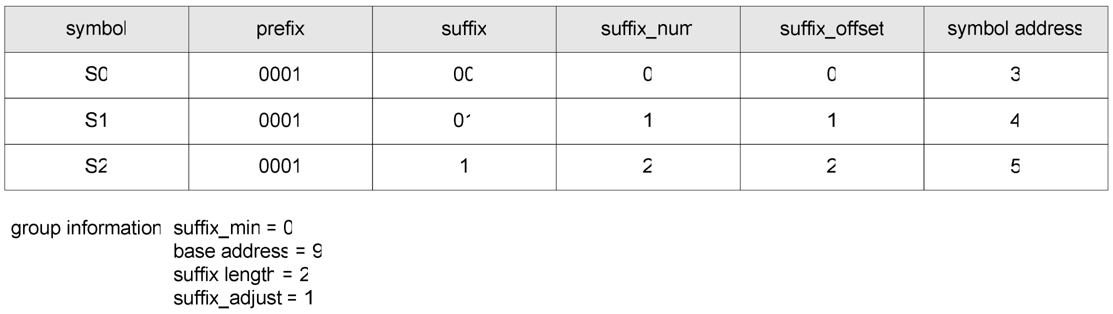


symbol	prefix	suffix	suffix_num	suffix_offset	symbol address
S0	0001	00	0	0	3
S1	0001	01	1	1	4
S2	0001	1	2	2	5

group information suffix_min = 0
base address = 9
suffix length = 2

Figure 3-10 : An example of the special case of suffix length

Some special cases will happen in VLC tables and Figure 3-10 shows an example of the special case of suffix length. The relative codeword of the symbol, S2, is “00011” and the other codewords in this group are “00010x”. We can find the suffix length is different. Therefore, we have to handle the condition to avoid fetching the incorrect suffix; otherwise we will get the wrong suffix_offset to access the wrong location of the symbol memory. Even sending the incorrect suffix length to codeword boundary detector will result the current decoded block fails. To solve this problem, we add an item called suffix_adjust to group information. When we get the group information and the suffix_adjust is set to 1, we have to examine the received suffix to see if we have to adjust the suffix length. The judgment of the suffix length adjusting is to examine if the first bit of the received suffix is 1. If the first bit of the received suffix is 1, the true suffix length is the suffix length of the group information minus 1. Other the other hand, the true suffix length is equal to suffix length.



symbol	prefix	suffix	suffix_num	suffix_offset	symbol address
S0	0001	00	0	0	3
S1	0001	01	1	1	4
S2	0001	1	2	2	5

group information suffix_min = 0
base address = 9
suffix length = 2
suffix_adjust = 1

Figure 3-11 : An example of the complete group information

Based on the memory map and the group information in Figure 3-11, the complete intra-group decoding procedures can be described as follows.

Decoding procedure – assume the decoded codeword is $(00011)_2$:

- ➢ The suffix_adjust is equal to 1, so we have to examine the first bit of the received suffix $(1)_2$. The first bit of the received suffix is also equal to 1, so suffix_num is set to $(10)_2$.
- ➢ suffix_offset = suffix_num $(10)_2$ – suffix_min (0) = 10_2 = 2;
- ➢ symbol_address = suffix_offset (2) + base_address (3) = 5;
- ➢ The decoded symbol, S2, is accessed by the symbol_address, 5;

In order to save the usage of memories, we don't save the whole base addresses to the group information, and we only save the least significant 7 bits. Therefore, when executing the CAVLC coeff_token decoding, the true base addresses have to be added 64, 128, and 192 for NUM_VLC1, NUM_VLC2, and NUM_VLC3. On the other hand, the true address has to be added 128 for MPEG-2 table B15.

group	suffix_adjust	leading_0s	base_address	suffix_length	suffix_min
0	0	1	0	0	1
1	1	1	1	2	0
2	0	1	4	2	1
3	0	1	7	2	0
4	0	1	11	2	0
5	0	1	15	2	0
6	0	1	23	3	0
7	0	1	31	4	0
8	0	1	47	4	0
9	0	1	63	4	0
10	0	1	79	4	0
11	0	1	95	4	0
16	0	0	0	0	0
17	0	0	1	0	0
18	0	0	2	1	0
19	0	0	4	2	0
20	1	0	8	2	0
21	0	0	12	1	0
22	0	0	14	0	0
23	0	0	15	N.A.	N.A.
24	1	1	16	2	0
25	0	1	20	2	1
26	0	1	23	2	0
27	0	1	27	2	0
28	0	1	31	2	0
29	1	1	39	3	0
30	0	1	47	4	1
31	0	1	63	4	0
32	0	1	79	4	0
33	0	1	95	4	0
34	0	1	111	4	0

Table 3-8 : The codeword groups of MPEG-2

group	suffix_adjust	leading_0s	base_address	suffix_length	suffix_min
0	0	1	0	0	1
1	0	1	1	0	1
2	0	1	2	0	1
3	1	1	3	2	0
4	1	1	6	2	0
5	0	1	9	2	0
6	0	1	13	2	0
7	0	1	17	2	0
8	0	1	21	2	0
9	0	1	25	3	0
10	0	1	33	3	0
11	0	1	41	3	0
12	0	1	49	3	0
13	0	1	57	2	0
14	0	1	61	0	1
16	0	1	0	1	0
17	1	1	2	2	0
18	1	1	5	3	0
19	0	1	12	2	0
20	0	1	16	2	0
21	0	1	20	2	0
22	0	1	24	2	0
23	0	1	28	3	0
24	0	1	36	3	0
25	0	1	44	3	0
26	1	1	52	3	0
27	0	1	59	2	0
28	0	1	63	0	1
32	0	1	0	3	0
33	0	1	8	3	0
34	0	1	16	3	0
35	0	1	24	3	0
36	0	1	32	3	0
37	0	1	40	3	0
38	1	1	48	3	0
39	0	1	55	2	0

40	0	1	59	1	0
41	0	1	61	0	1
48	0	1	0	0	1
49	0	1	1	0	1
50	0	1	2	0	1
51	0	1	3	2	0
52	0	1	7	1	0
53	0	1	9	1	0
54	0	1	11	1	0
55	0	1	13	0	0

Table 3-9 : The codeword groups of CAVLC coeff_token

3.3.4. Intra-Group Encoding Procedure

In addition to grouping symbols, it is necessary for encoding procedures to map codewords onto memories and extract symbol group information. During intra-group codeword memory mapping, the memory address of a codeword in a group is calculated by the encoding_offset of this codeword and the base address which denotes the codeword address of the encoding_min of the group. In other words, the codeword address is the sum of the encoding_offset and the base address. After applying this arithmetic relation, encoding_offsets, decoded codeword addresses, and decoded symbols can be found by numerical calculations rather than pattern matching. Therefore, the group information to be stored is encoding_min, and base addresses.

Based on the memory map and the group information in Figure 3-12, intra-group encoding procedures can be described as follows.

Encoding procedure – assume the decoded symbols are TotalCoeff is 8 and TrailingOnes is 0:

- encoding_offset = encoding _num (8) –encoding _min (0) = 8;
- codeword_address = encoding _offset (8) + base_address (0) = 8;
- The encoded codeword, 0000_0000_0100_0, is accessed by the

codeword_address, 8;

codeword	TrailingOnes	TotalCoeff	encoding_num	encoding_offset	symbol address
1	0	0	0	0	0
000101	0	1	1	1	1
00000111	0	2	2	2	2
000000111	0	3	3	3	3
0000000111	0	4	4	4	4
00000000111	0	5	5	5	5
0000000001111	0	6	6	6	6
0000000001011	0	7	7	7	7
0000000001000	0	8	8	8	8
00000000001111	0	9	9	9	9
00000000001011	0	10	10	10	10
000000000001111	0	11	11	11	11
000000000001011	0	12	12	12	12
0000000000001111	0	13	13	13	13
0000000000001011	0	14	14	14	14
0000000000000111	0	15	15	15	15
0000000000000100	0	16	16	16	16

group information: suffix_min = 0
base_address = 0

Figure 3-12 : An example of intra-group codeword memory map and group information

In order to save the usage of memories, we don't save the whole base addresses to the group information, and we only save the least significant 7 bits. Therefore, when executing the CAVLC coeff_token decoding, the true base addresses have to be added 64, 128, and 192 for NUM_VLC1, NUM_VLC2, and NUM_VLC3. On the other hand, the true address has to be added 128 for MPEG-2 table B15.

3.3.5. Decoding Group-Searching Scheme and overall group-based decoding processes

Because the decoding procedures are performed after the group information is acquired, and efficient group-searching scheme with low complexity and high operation rate determines the performance of a group-based VLC decoding system. To realize such a group searching scheme, we abandon the conventional group-searching scheme which is to calculate the range of PCLC_codenum to fetch the decoding group. If the number of decoding group is large, we have to iterate the group-searching scheme until the correct range is found. Besides, if we use PCLC_codenum group-searching scheme, we have to save the PCLC_mincode with the longest codeword length. Therefore, the conventional group-searching scheme is not efficient enough about group-searching time and memory usage. We use the proposed group-searching scheme called prefix-zero-group-searching (PZGS) and inter-group symbol memory mapping to realize the decoding group searching.

A PZGS scheme is to count the leading zeros of the received codeword and the value of the leading zeros is the base of the group number. Then, we have to fetch the additional group number according to the value of NUM_VLC in CAVLC or the table which is used in MPEG-2. The relative additional group numbers are 0, 16, 32, and 48 for NUM_VLC0, NUM_VLC1, NUM_VLC2, and NUM_VLC3 in CAVLC, and those are 0 and 32 for MPEG-2 table B14 and B15. The sum of the base group number and additional group number is the group number we have to access. Based on the codeword group table, the base addresses have to be assigned in group number order, i.e. $base\_addr_0 < base\_addr_1 < \dots < base\_addr_n$ for inter-group symbol memory mapping. An example of the PZGS table and the intra-/inter-group symbol memory map is shown in Table 3-10. The group information of the PZGS table is given in

Table 3-11.

group	symbol	prefix	NUM_VLC	symbol address	suffix_offset
G0	S000	0	0	0	N.A.
G1	S010	1	0	1	N.A.
G2	S020	2	0	2	N.A.
G3	S030	3	0	3	0
	S031	3	0	4	1
	S032	3	0	5	2
G16	S160	0	1	64	0
	S161	0	1	65	1
G17	S170	1	1	66	0
	S171	1	1	67	1
	S172	1	1	68	2
G18	S180	2	1	69	0
	S181	2	1	70	1
	S182	2	1	71	2
	S183	2	1	72	3
	S184	2	1	73	4
	S185	2	1	74	5
G19	S190	3	1	75	0
	S191	3	1	76	1
	S192	3	1	77	2
	S193	3	1	78	3

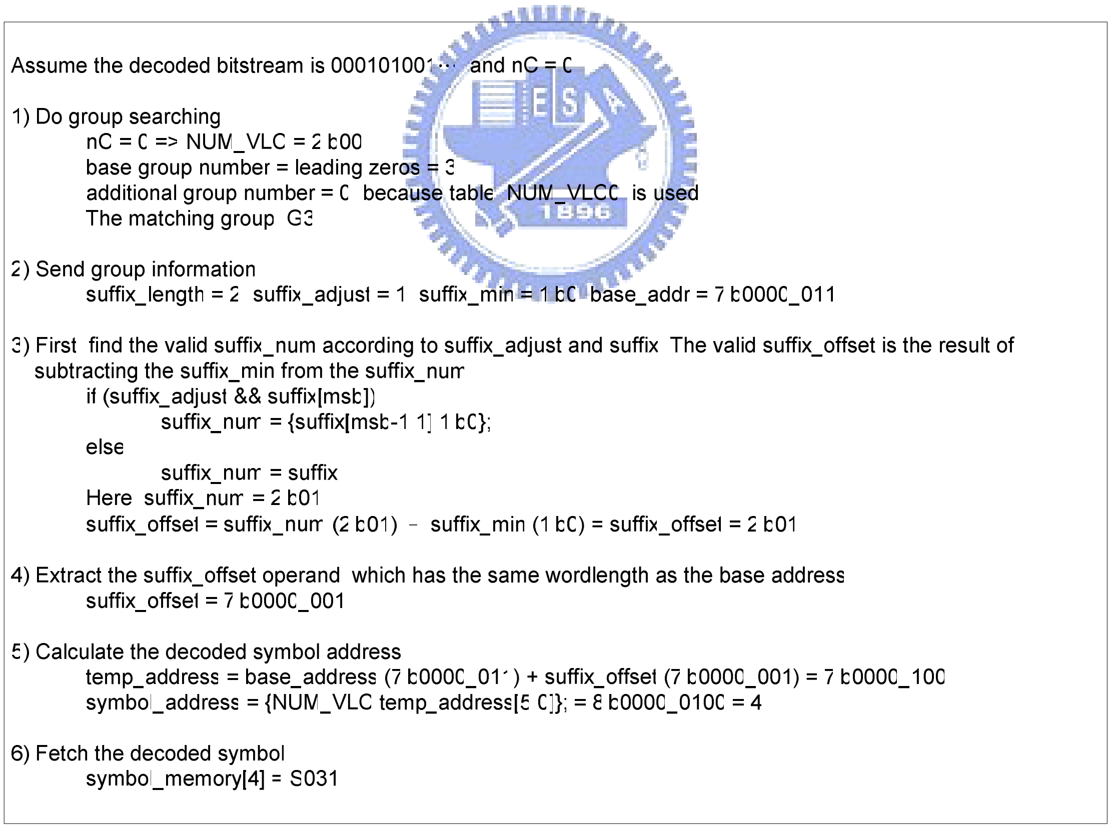
Table 3-10 : CAVLC PZGS table and intra-/inter-group symbol memory map

group	suffix_adjust	leading_0s	base_address	suffix_length	suffix_min
0	0	1	0	0	1
1	0	1	1	0	1
2	0	1	2	0	1
3	1	1	3	2	0
16	0	1	0	1	0
17	1	1	2	2	0
18	1	1	5	3	0
19	0	1	12	2	0

Table 3-11 : CAVLC group information of the coding table shown in Table 3-10

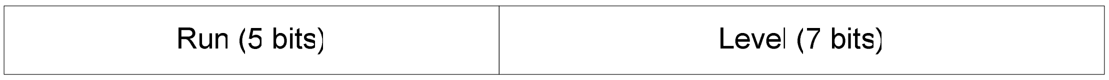
Before realizing the decoding processes, the word lengths of both suffix_offset and suffix_num operands have to be determined, since it is difficult to implement arithmetic units with variable length inputs. To perform memory mapping, the supported symbol memory must satisfy the requirement of coding tables. Consequently, the value of suffix_offsets and suffix_num will not exceed the address space of the symbol memory. For this reason, it is reasonable that the word length of the suffix_offset and suffix_num operands equal that of the base address.

Based on the word lengths of the operands discussed above, the VLC decoding algorithm is completed by the group searching scheme and the intra-group decoding procedures. Detailed descriptions of the VLC decoding processes and corresponding example based the coding table in Table 3-11, Table 3-12, and Figure 3-13.



```
Assume the decoded bitstream is 000101001…, and nC = 0

1) Do group searching
        nC = 0 => NUM_VLC = 2'b00
        base group number = leading zeros = 3
        additional group number = 0, because table NUM_VLC0 is used
        The matching group: G3

2) Send group information
        suffix_length = 2, suffix_adjust = 1, suffix_min = 1'b0, base_addr = 7'b0000_011

3) First, find the valid suffix_num according to suffix_adjust and suffix. The valid suffix_offset is the result of
   subtracting the suffix_min from the suffix_num
        if (suffix_adjust && suffix[msb])
              suffix_num = {suffix[msb-1:1], 1'b0};
        else
              suffix_num = suffix
        Here, suffix_num = 2'b01
        suffix_offset = suffix_num (2'b01) – suffix_min (1'b0) = suffix_offset = 2'b01

4) Extract the suffix_offset operand, which has the same wordlength as the base address
        suffix_offset = 7'b0000_001

5) Calculate the decoded symbol address
        temp_address = base_address (7'b0000_011) + suffix_offset (7'b0000_001) = 7'b0000_100
        symbol_address = {NUM_VLC, temp_address[5:0]}; = 8'b0000_0100 = 4

6) Fetch the decoded symbol
        symbol_memory[4] = S031
```

Figure 3-13 : CAVLC decoding processes and corresponding examples



Run (5 bits)	Level (7 bits)

Figure 3-14 : The memory usage for conventional symbol memory

In the conventional symbol memory, two symbols, run and level, are directly stored into the symbol memory, and the length of binary number for run is 5-bit, and the length of binary number of level is 7-bit including the sign bit. Besides, the memory is 256-entry. If we decrease 4 bits of the symbol memory, we can reduce 1024-bit memories. In order to memory reduction, we add one step for MPEG-2 decoding processes, but under will pipelined scheme it doesn't make great influence on the decoding throughput. We take advantage of the feature of symbol groups, that is, when we fetch the group number, the group number can be translated into one symbol. We only save the value of the other symbol, and finally we can get all decoded symbols. Therefore, the symbols groups have to provide the group information about which decoded symbol derived from symbol groups and its value. The group information of MPEG-2 symbol groups is shown in Table 3-12. Symbol_adjust also helps reduce the memory usage. When we put the entire value of levels into symbol memories, we have to use 6-bit memory width. Therefore, we separate the levels, 1 ~ 31, and the level, 32 ~ 40, into two groups in Table 3-6. In symbol memories, we only save the least 5-bit binary number of levels, and when symbol_adjust is equal to 1, we get the result of decoded level by adding the level derived from symbol memories and 32. The variable, run_or_level, means the value stored in the mapping symbol group is run or level, and 1 is to store run. Of course, symbol means the value of decoded symbol stored in symbol groups.

group	symbol_adjust	run_or_level	base_address	symbol
G0	0	1	0	00
G1	1	1	31	00
G2	0	1	40	01
G3	0	1	58	10
G4	0	1	63	11
G5	0	0	67	01

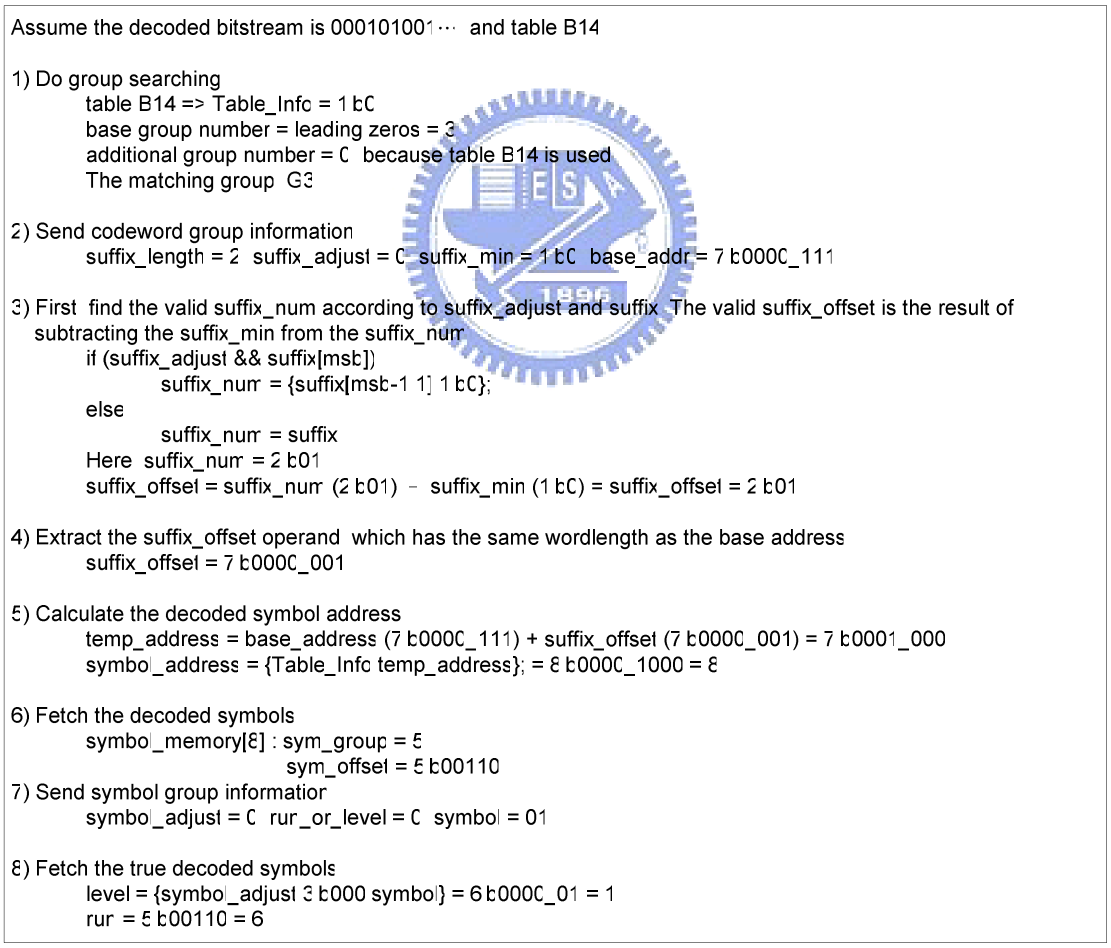
Table 3-12 : An example of MPEG-2 symbol group information

group	symbol	prefix	Table	symbol address	suffix_offset
G3	S030	2	B14	7	0
	S031	3	B14	8	1
	S032	3	B14	9	2
	S033	3	B14	10	3

Table 3-13 : MPEG-2 PZGS table and intra-/inter-group symbol memory map

group	suffix_adjust	leading_0s	base_address	suffix_length	suffix_min
0	0	1	0	0	1
1	1	1	1	2	0
2	0	1	4	2	1
3	0	1	7	2	0

Table 3-14 : An example of MPEG-2 group information of the coding table



```
Assume the decoded bitstream is 000101001…  and table B14

1) Do group searching
        table B14 => Table_Info = 1 b0
        base group number = leading zeros = 3
        additional group number = 0  because table B14 is used
        The matching group  G3

2) Send codeword group information
        suffix_length = 2  suffix_adjust = 0  suffix_min = 1 b0  base_addr = 7 b0000_111

3) First  find the valid suffix_num according to suffix_adjust and suffix  The valid suffix_offset is the result of
   subtracting the suffix_min from the suffix_num
        if (suffix_adjust && suffix[msb])
                suffix_num  = {suffix[msb-1 1] 1 b0};
        else
                suffix_num  = suffix
        Here  suffix_num  = 2 b01
        suffix_offset = suffix_num  (2 b01)  -  suffix_min (1 b0) = suffix_offset = 2 b01

4) Extract the suffix_offset operand  which has the same wordlength as the base address
        suffix_offset = 7 b0000_001

5) Calculate the decoded symbol address
        temp_address = base_address (7 b0000_111) + suffix_offset (7 b0000_001) = 7 b0001_000
        symbol_address = {Table_Info temp_address}; = 8 b0000_1000 = 8

6) Fetch the decoded symbols
        symbol_memory[8] : sym_group = 5
                           sym_offset = 5 b00110
7) Send symbol group information
        symbol_adjust = 0  rur_or_level = 0  symbol = 01

8) Fetch the true decoded symbols
        level = {symbol_adjust 3 b000 symbol} = 6 b0000_01 = 1
        rur  = 5 b00110 = 6
```

Figure 3-15 : MPEG-2 decoding processes and corresponding examples

3.3.6. Encoding Group-Searching Scheme and overall group-based decoding processes

Due to the encoding efficiency, we don't consider the conventional encoding group-searching scheme which has the same disadvantage as the conventional decoding group-searching scheme. When we build the symbol groups, we already take the efficiency of encoding group-searching scheme into consideration. The symbol groups and their group information are shown in Table 3-15 and Table 3-16. The proposed encoding group-searching scheme is similar to the proposed decoding one. When executing the CAVLC coeff_token encoding processes, we search the symbol groups according to the value of TrailingOnes, and based on the value of run or level, we can do the same thing in MPEG-2. In CAVLC, the value of TrailingOnes directly maps to the symbol groups. In MPEG-2, the symbol groups only map to run or level whose value is less than 4. When receiving the value of run equals 0, if the value of level is less than 32, the matching group is G0. Besides, when the value of run is less than 4, the matching group is the result of adding 1 and the value of level. On the other hand, when the value of run is greater than 3, the matching group is the result of adding 4 and the value of level. We can see this relationship in Table 3-15.

group	run	level	symbol_adjust	run_or_level	base_address	symbol
G0	0	1~31	0	1	0	2'b00
G1	0	32~40	1	1	31	2'b00
G2	1	1~18	0	1	40	2'b01
G3	2	1~5	0	1	58	2'b10
G4	3	1~4	0	1	63	2'b11
G5	4~31	1	0	0	67	2'b01
G6	4~16	2	0	0	95	2'b10
G7	4~6	3	0	0	110	2'b11

Table 3-15 : MPEG-2 symbol groups and group information

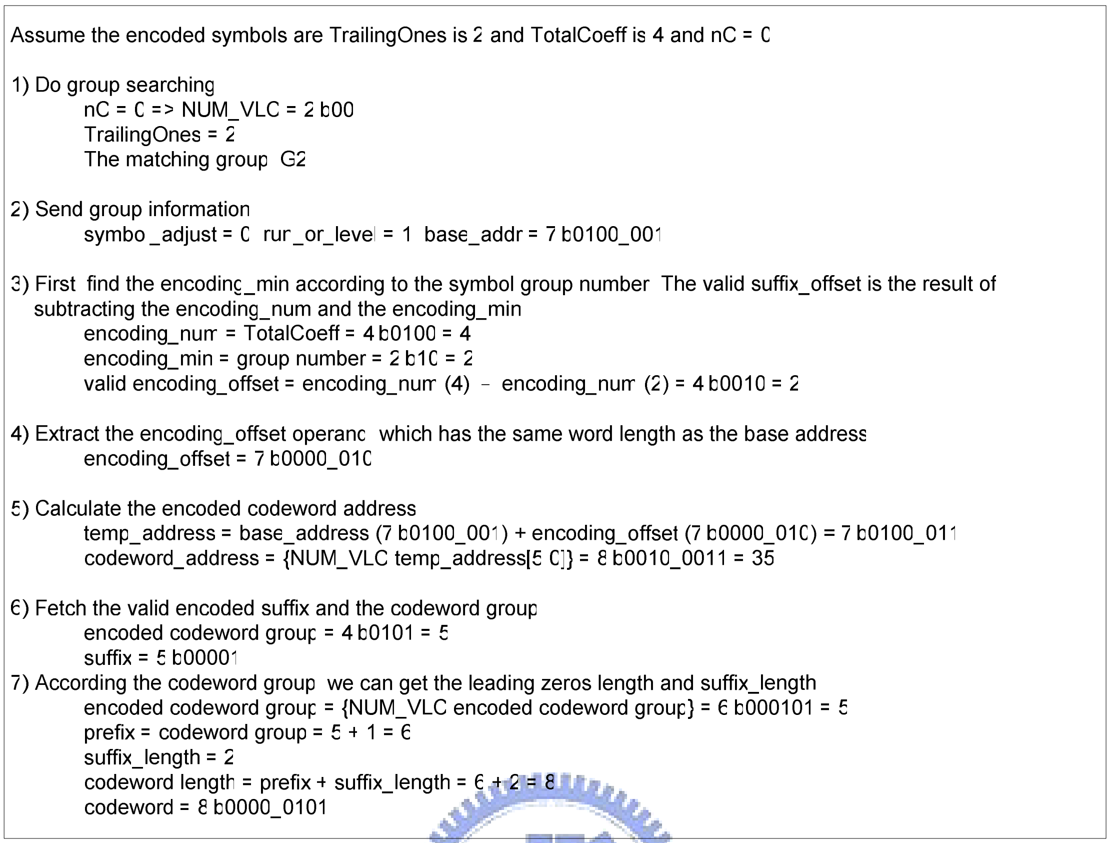
group	T1s	TC	symbol_adjust	run_or_level	base_address	symbol
G0	0	0~16	0	1	0	2'b00
G1	1	1~16	0	1	17	2'b00
G2	2	2~16	0	1	33	2'b01
G3	3	3~16	0	1	48	2'b10

Table 3-16 : CAVLC coeff_token symbol groups and group information

Before realizing the encoding processes, we have to check encoding_min and encoding_num, and the word lengths of both encoding_offset and encoding_num operands have to be determined, since it is difficult to implement arithmetic units with variable length inputs. For the purpose of memory reduction, we don't save encoding_min into group information, but from symbol_adjust and run_or_level we can get the information. When run_or_level is 1, we can know the fixed symbol for this group is run, and the minimum value of level is 0 or 1 which can determined by symbol_adjust. On the other hand, the fixed symbol of this group is level, so encoding_min is equal to 4. Besides, according to run_or_level, we can decide encoding_num is run or level. In CAVLC coeff_token encoding symbol groups, we also don't store encoding_min in the group information, but we can get it from the group number.

To perform memory mapping, the supported codeword memory must satisfy the requirement of coding tables. Consequently, the value of encoding_offsets and encoding_num will not exceed the address space of the codeword memory. For this reason, it is reasonable that the word length of the encoding_offset and encoding_num operands equal that of the base address.

Based on the word lengths of the operands discussed above, the VLC encoding algorithm is completed by the group searching scheme and the intra-group encoding procedures. Detailed descriptions of the VLC decoding processes and corresponding example based the coding table in Figure 3-16 and Figure 3-17.


Assume the encoded symbols are TrailingOnes is 2 and TotalCoeff is 4 and nC = 0

1) Do group searching
    nC = 0 => NUM_VLC = 2 b00
    TrailingOnes = 2
    The matching group G2

2) Send group information
    symbo _adjust = 0 rur _or_level = 1 base_addr = 7 b0100_001

3) First find the encoding_min according to the symbol group number The valid suffix_offset is the result of
   subtracting the encoding_num and the encoding_min
    encoding_num = TotalCoeff = 4 b0100 = 4
    encoding_min = group number = 2 b10 = 2
    valid encoding_offset = encoding_num (4) – encoding_num (2) = 4 b0010 = 2

4) Extract the encoding_offset operand which has the same word length as the base address
    encoding_offset = 7 b0000_010

5) Calculate the encoded codeword address
    temp_address = base_address (7 b0100_001) + encoding_offset (7 b0000_010) = 7 b0100_011
    codeword_address = {NUM_VLC temp_address[5 0]} = 8 b0010_0011 = 35

6) Fetch the valid encoded suffix and the codeword group
    encoded codeword group = 4 b0101 = 5
    suffix = 5 b00001
7) According the codeword group we can get the leading zeros length and suffix_length
    encoded codeword group = {NUM_VLC encoded codeword group} = 6 b000101 = 5
    prefix = codeword group = 5 + 1 = 6
    suffix_length = 2
    codeword length = prefix + suffix_length = 6 + 2 = 8
    codeword = 8 b0000_0101

Figure 3-16 : CAVLC encoding processes and corresponding examples

Assume the encoded symbols are run is 0 and level is 34 and table B14.

1) Do group searching.

```
    table B14 => Table_Info = 1'b0;
    run = 0 and level > 32;
    The matching group: G1;
```

2) Send group information.

```
    symbol_adjust = 1, run_or_level = 1, base_addr = 7'b0011_111;
```

3) First, find the encoding_min according to symbol_adjust and run_or_level or the encoding_num according. The valid encoding_offset is the result of subtracting the encoding_num and the encoding_min.

```
    if (run_or_level == 1)
        encoding_num = level[4:0];
    else
        encoding_num = run;
    if (run_or_level == 1)
    {
        if (symbol_adjust == 1)
            encoding_min = 5'b00000;
        else
            encoding_min = 5'b00001;
    }
    encoding_num = 5'b00010;
    encoding_min = 5'b00000;
    valid encoding_offset = encoding_num (5'b00010) – encoding_min (5'b00000) = 5'b00010 = 2;
```

4) Extract the encoding_offset operand, which has the same word length as the base address.

```
    encoding_offset = 7'b0000_010;
```

5) Calculate the encoded codeword address.

```
    temp_address = base_address (7'b0011_111) + encoding_offset (7'b0000_010) = 7'b0100_001;
    codeword_address = {Table_Info,temp_address} = 8'b0010_0001 = 33;
```

6) Fetch the valid encoded suffix and the codeword group.

```
    encoded codeword group = 4'b1010 = 10;
    suffix = 5'b00010;
```

7) According the codeword group, we can get the leading zeros length and suffix_length;

```
    encoded codeword group = {Table_Info,1'b0,encoded codeword group} = 6'b001010 = 10;
    prefix = codeword group = 10 + 1 = 11;
    suffix_length = 4;
    codeword length = prefix + suffix_length = 11 + 4 = 15;
    codeword = 15'b0000_0000_0011_0;
```

Figure 3-17 : MPEG-2 encoding processes and corresponding examples

3.3.7. Group-Based VLC Coded System Architecture

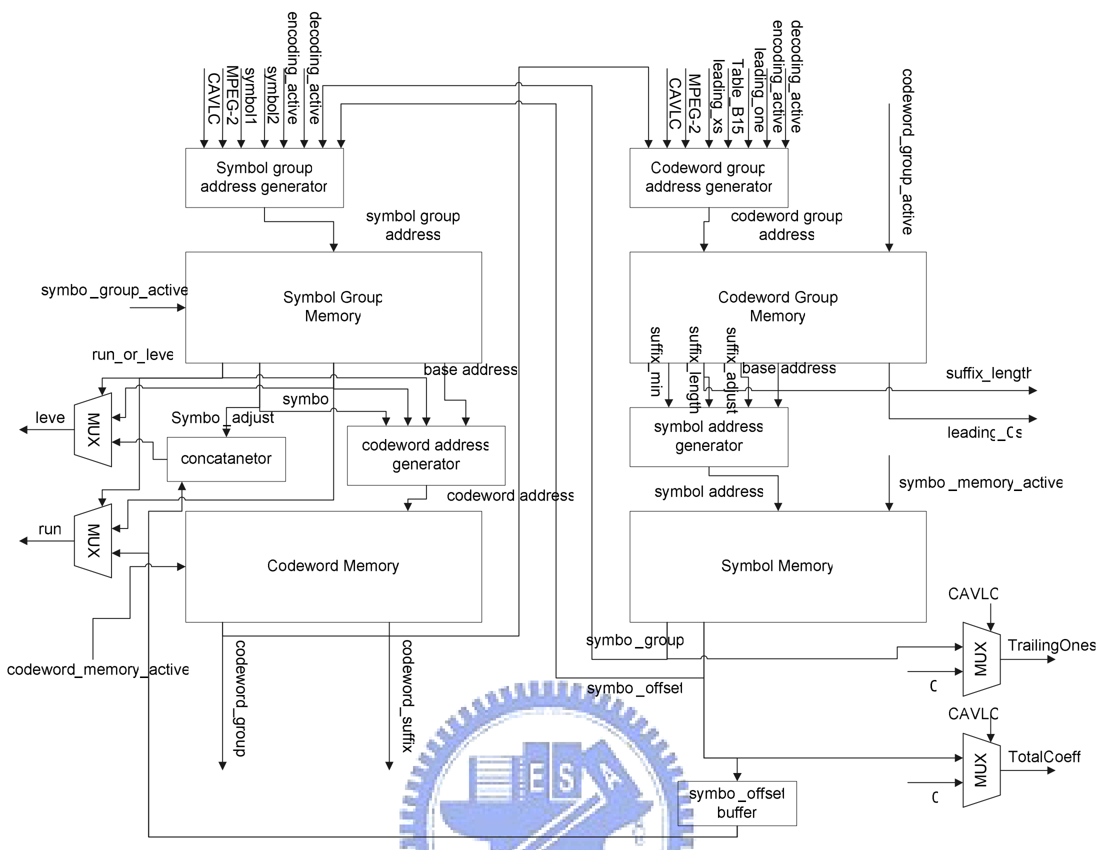


Figure 3-18 : Block diagram of the proposed VLC codec system for MPEG applications

The proposed VLC codec system is designed for MPEG applications with coding tables up to 256-entry 12-bit symbols and 16-bit codewords, and H.264/AVC CAVLC coeff_token with coding tables up to 256-entry 6-bit symbols and 16-bit codewords. This system performs concurrent encoding and decoding procedures by accessing the group information and achieves table programmability by loading data into on-chip memories. Block diagram of the proposed VLC codec system is shown in Figure 3-18. It mainly consists of the following components.

➢ The codeword group address generator calculates the codeword group address according to the information of leading_xs, Table_B15, leading_one, and codeword_group. The whole function unit is controlled by two signals, encoding_active and decoding_active. The variable, leading_xs, is the number of leading zeros or ones, and leading_one determines leading_xs is leading zeros or leading ones. Table_B15 means the codewords table is table B15 in MPEG-2. The detailed architecture of codeword group address generator is shown in Figure 3-19.

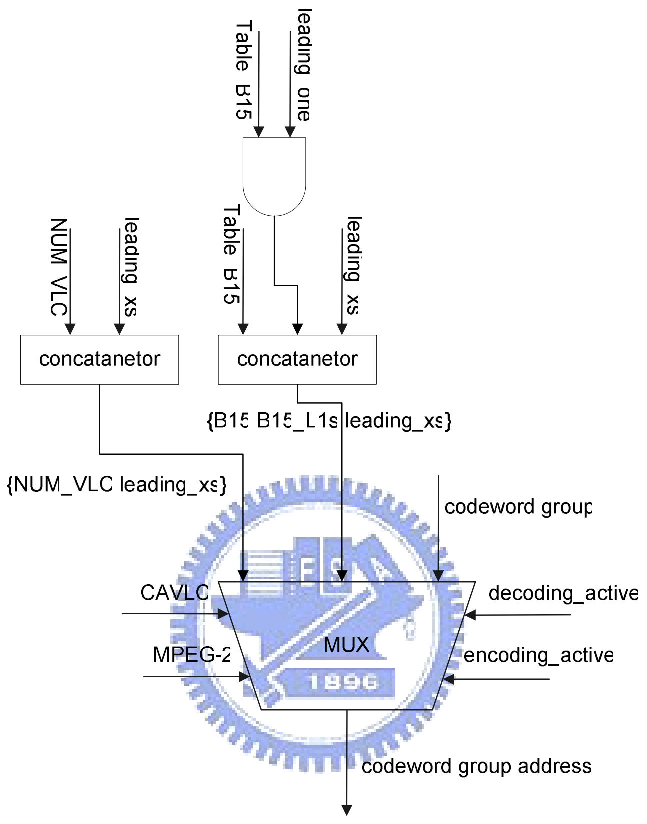


Figure 3-19 : Architecture of codeword group address generator

➢ The symbol address generator calculates the symbol address according to the information of suffix_min, suffix_adjust, and base address which is derived from the codeword group information. The whole function unit is controlled by two signals, encoding_active and decoding_active. When we get the correct suffix_length for the decoded symbol, we can fetch the correct suffix according to the barrel shifter to get the suffix_num. The detailed architecture of codeword group address generator is shown in Figure 3-20.

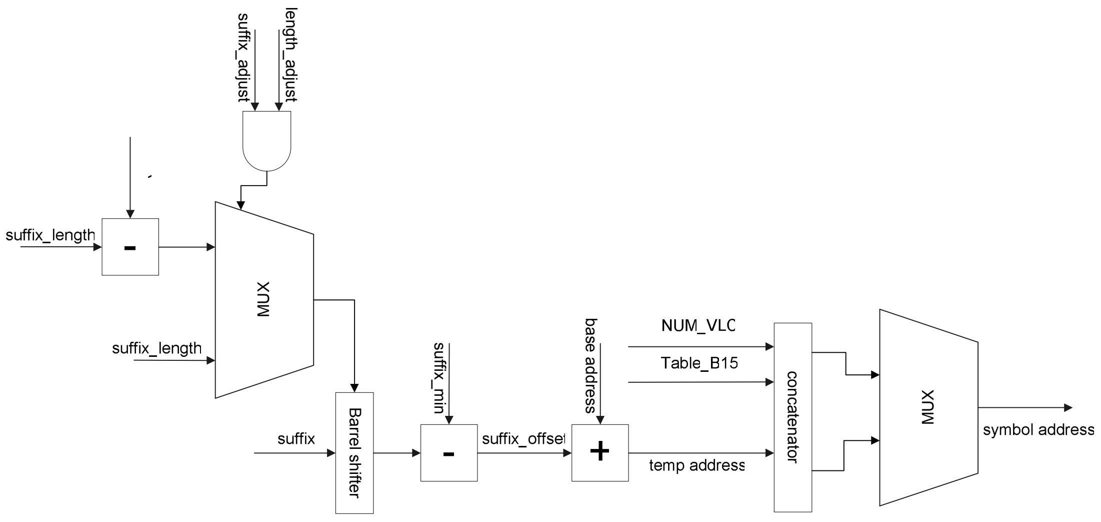


Figure 3-20 : Architecture of symbol address generator

➢ The symbol group address generator calculates the symbol group address according to the information of symbol1 and symbol2. In we execute MPEG-2 decoding/encoding, symbol1 is run and symbol2 is level. On the other hand, symbol1 is TrailingOnes and symbol2 is TotalCoeff. The whole function unit is controlled by two signals, encoding_active and decoding_active. The detailed architecture of codeword group address generator is shown in Figure 3-21.

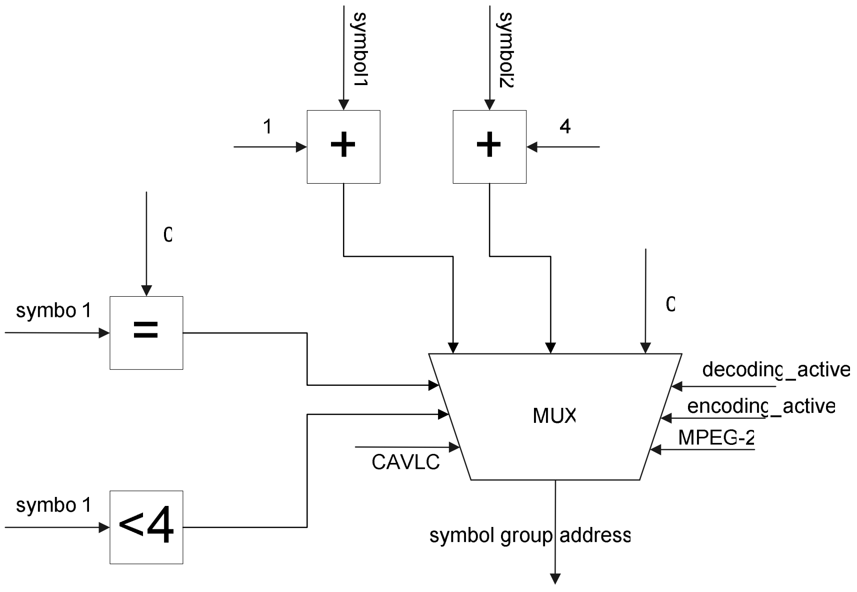


Figure 3-21 : Architecture of symbol group address generator

➢ The codeword address generator calculates the codeword address according to the information of symbol_adjust, run_or_level, symbol, and base address which are derived from the symbol group information. The whole function unit is controlled by two signals, encoding_active and decoding_active. The detailed architecture of codeword group address generator is shown in Figure 3-22.

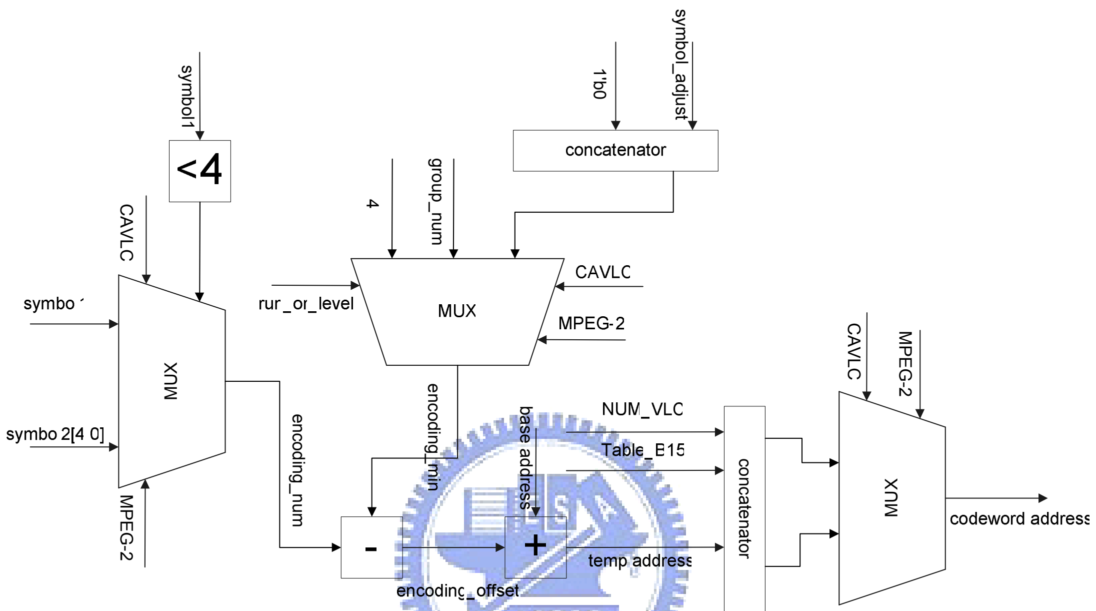


Figure 3-22 : Architecture of codeword address generator

3.4. Summary

suffix_adjust 1 bit	leading_0s 1 bit	base address 7 bits	suffix_length 3 bits	suffix_min 1 bit

codeword group

symbol_group 3 bits	symbol_offset 5 bits

symbol memory

symbol_adjust 1 bit	run_or_level 1 bit	base address 7 bits	symbol 2 bits

symbol group

suffix 4 bits	codeword_group 5 bits

codeword memory

Figure 3-23 : Formats of all kinds of memories

Figure 3-23 shows the formats of codeword group memory, symbol memory, symbol group memory, and codeword memory. After the result of memory reduction of the proposed VLC codec system, the total memories are 5272 bits, and the memory usage for each memory is shown in Figure 3-24. Compared to the memory usage of CABAD shown in Figure 3-2, the memory usage of the proposed VLC codec system is much smaller than that of CABAD. Therefore, the proposed VLC codec system can easily share the memory with CABAD.

Besides, the conventional VLC codec system use 6304-bit memory, and our proposed VLC codec system save 16% memory usage. However, the conventional VLC codec system can't support CAVLC, so under the same environment, that is, we compare the memory usage without considering the CAVLC encoding/decoding, we use 4856-bit memory and save 23% memory usage. The proposed VLC codec system has almost the same throughput as the conventional one.

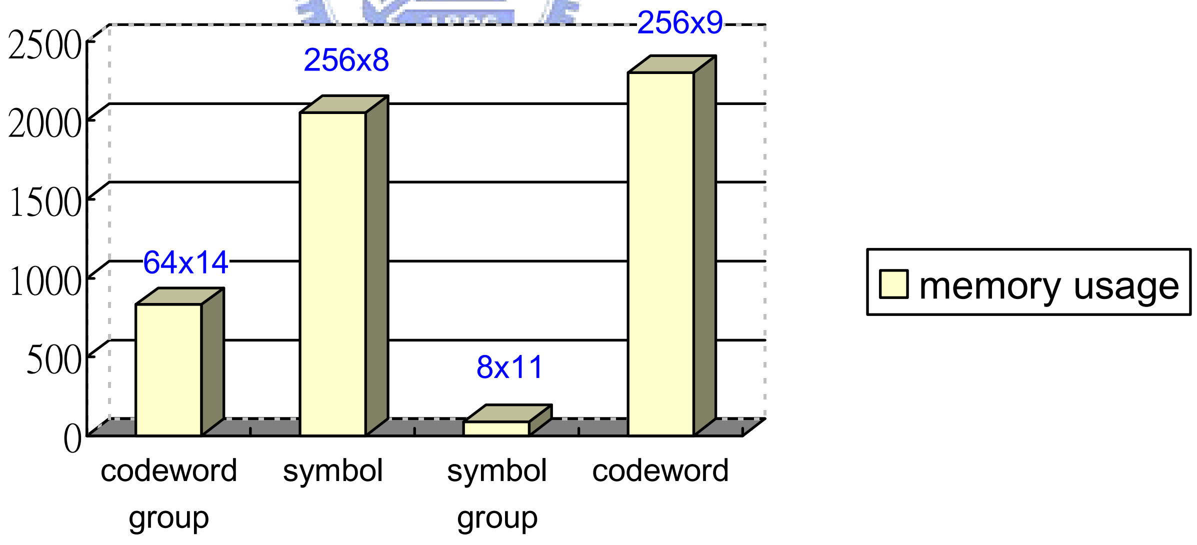


Figure 3-24 : The memory usage of each memory

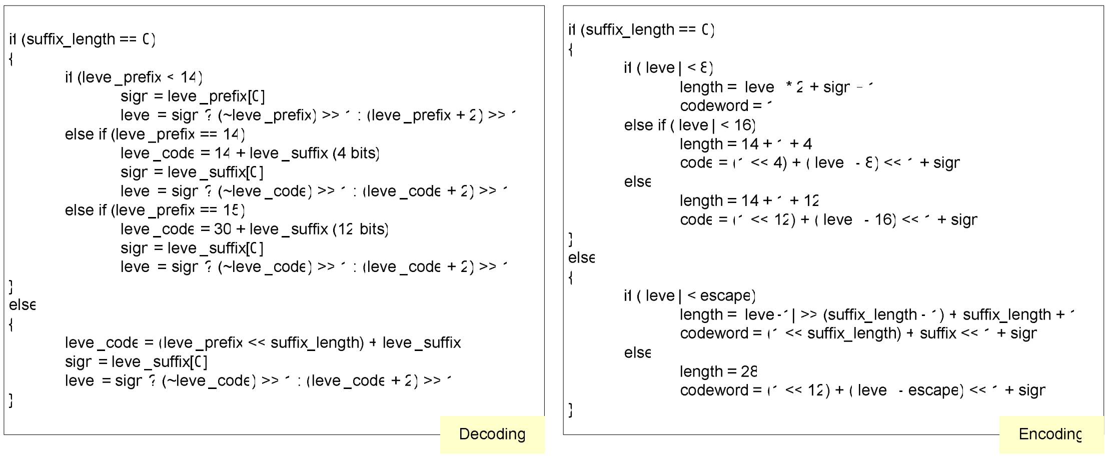
Chapter 4. Optimization of the Proposed VLC Codec System

From Figure 3-3, we can see other decoding/encoding procedures in addition to coeff_token in CAVLC. Even if they are VLC encoding/decoding procedures based on Huffman coding except trailing_ones_sign_flag, for the purpose of the performance, we don’t use the proposed group-based VLC encoding/decoding approaches to realize other parts. Besides, the components, symbols constructor and bitstream concatenater, also have great influence on the encoding/decoding procedures. In this chapter, we will introduce the other optimization of the proposed VLC codec system.

4.1. Efficient Coding

Efficient coding means we can use simple arithmetic approaches to realize the VLC tables, because we would find the same numerical rules among different tables. From these numerical rules, we can cluster the original tables into a few groups. In the same group, the symbols or the codewords have the same numerical calculations to get the relative codewords or symbols. By this way, we can implement these decoding/encoding procedures without memory usage, and we can get the encoded codewords or decoded symbols quickly to provide the better throughput for the entire codec system.

4.1.1. Level Efficient Coding



```
if (suffix_length == 0)
{
        if (level_prefix < 14)
                sign = level_prefix[0]
                level = sign ? (~level_prefix) >> 1 : (level_prefix + 2) >> 1
        else if (level_prefix == 14)
                level_code = 14 + level_suffix (4 bits)
                sign = level_suffix[0]
                level = sign ? (~level_code) >> 1 : (level_code + 2) >> 1
        else if (level_prefix == 15)
                level_code = 30 + level_suffix (12 bits)
                sign = level_suffix[0]
                level = sign ? (~level_code) >> 1 : (level_code + 2) >> 1
}
else
{
        level_code = (level_prefix << suffix_length) + level_suffix
        sign = level_suffix[0]
        level = sign ? (~level_code) >> 1 : (level_code + 2) >> 1
}
```

```
if (suffix_length == 0)
{
        if ( |level| < 8)
                length = |level| * 2 + sign - 1
                codeword = 1
        else if ( |level| < 16)
                length = 14 + 1 + 4
                code = (1 << 4) + ( |level| - 8) << 1 + sign
        else
                length = 14 + 1 + 12
                code = (1 << 12) + ( |level| - 16) << 1 + sign
}
else
{
        if ( |level| < escape)
                length = |level-1| >> (suffix_length - 1) + suffix_length + 1
                codeword = (1 << suffix_length) + suffix << 1 + sign
        else
                length = 28
                codeword = (1 << 12) + ( |level| - escape) << 1 + sign
}
```

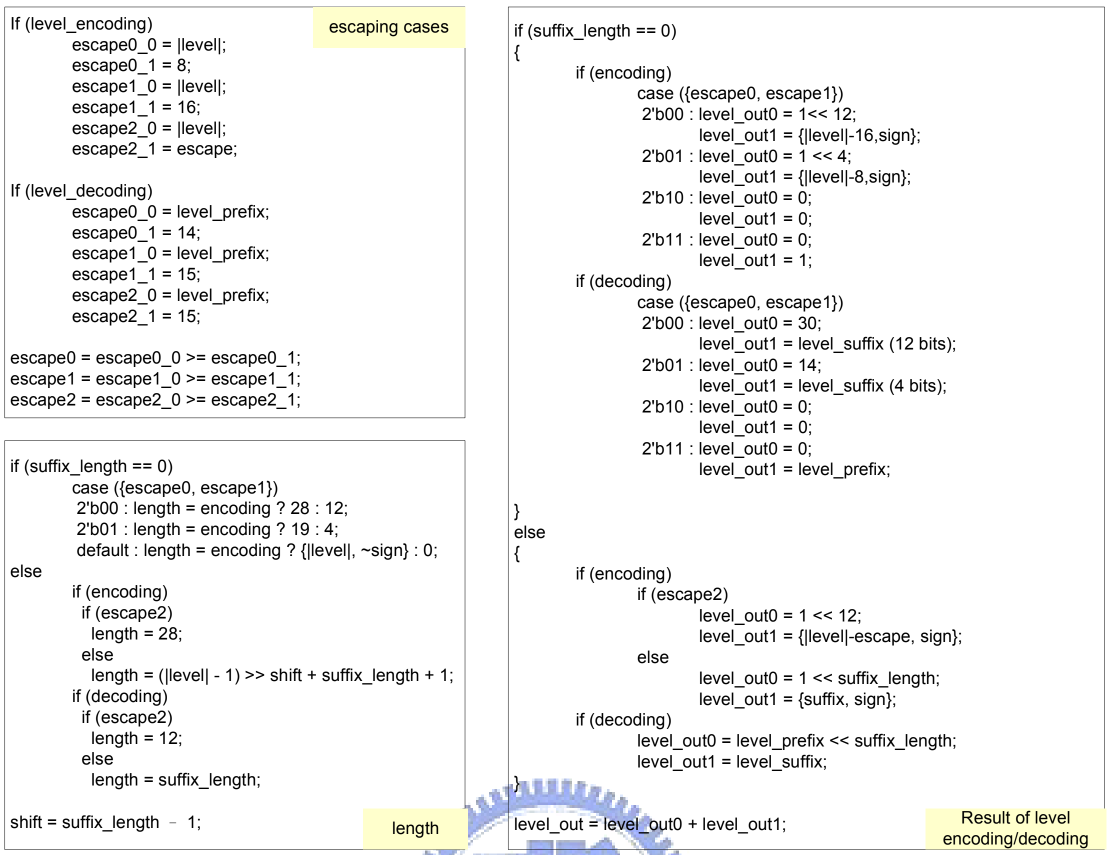
Figure 4-1 : Algorithm of level encoding and decoding

Figure 4-1 shows the algorithm of level encoding and decoding. According to the value of suffix_length, we can choose the decoded table from NUM_VLC0 ~ NUM_VLC6. When suffix_length is 0, there are two escape cases (level_prefix = 14 or 15) which have to fetch level_suffix to decode. On the other hand, the suffix length is equal to the variable, suffix_length. The variable, sign, means the level is positive or negative.

In the encoding procedures, length is the codeword length and code represents the codeword value. The variable, escape, is defined as the following equation.

$$escape = 15 << suffix\_length$$

The variable, escape, determines the threshold of escape case, and if the value of level is greater than or equal to escape, the encoding procedure enters the escape case. In escape case, level_prefix is given 15 and the level_suffix length is 12. This is for the large value of encoding levels. The two cases, |level| < 16 and |level| ≧ 16, are the mapping to the two escape cases in the decoding process. According to the encoding and decoding algorithm shown in Figure 4-1, we can formulates the calculations of level encoding/decoding shown in Figure 4-2.


escaping cases
length
Result of level encoding/decoding

```
If (level_encoding)
        escape0_0 = |level|;
        escape0_1 = 8;
        escape1_0 = |level|;
        escape1_1 = 16;
        escape2_0 = |level|;
        escape2_1 = escape;

If (level_decoding)
        escape0_0 = level_prefix;
        escape0_1 = 14;
        escape1_0 = level_prefix;
        escape1_1 = 15;
        escape2_0 = level_prefix;
        escape2_1 = 15;

escape0 = escape0_0 >= escape0_1;
escape1 = escape1_0 >= escape1_1;
escape2 = escape2_0 >= escape2_1;
```

```
if (suffix_length == 0)
        case ({escape0, escape1})
         2'b00 : length = encoding ? 28 : 12;
         2'b01 : length = encoding ? 19 : 4;
         default : length = encoding ? {|level|, ~sign} : 0;
else
        if (encoding)
          if (escape2)
            length = 28;
          else
            length = (|level| - 1) >> shift + suffix_length + 1;
        if (decoding)
          if (escape2)
            length = 12;
          else
            length = suffix_length;

shift = suffix_length – 1;
```

```
if (suffix_length == 0)
{
        if (encoding)
                case ({escape0, escape1})
                 2'b00 : level_out0 = 1<< 12;
                         level_out1 = {|level|-16,sign};
                 2'b01 : level_out0 = 1 << 4;
                         level_out1 = {|level|-8,sign};
                 2'b10 : level_out0 = 0;
                         level_out1 = 0;
                 2'b11 : level_out0 = 0;
                         level_out1 = 1;
        if (decoding)
                case ({escape0, escape1})
                 2'b00 : level_out0 = 30;
                         level_out1 = level_suffix (12 bits);
                 2'b01 : level_out0 = 14;
                         level_out1 = level_suffix (4 bits);
                 2'b10 : level_out0 = 0;
                         level_out1 = 0;
                 2'b11 : level_out0 = 0;
                         level_out1 = level_prefix;

}
else
{
        if (encoding)
                if (escape2)
                        level_out0 = 1 << 12;
                        level_out1 = {|level|-escape, sign};
                else
                        level_out0 = 1 << suffix_length;
                        level_out1 = {suffix, sign};
        if (decoding)
                level_out0 = level_prefix << suffix_length;
                level_out1 = level_suffix;
}

level_out = level_out0 + level_out1;
```

Figure 4-2 : Calculations of level encoding and decoding

The escape cases for level encoding procedures are |level| < 8, |level| < 16, and |level| < escape, and those for level decoding procedures are level_prefix < 14, 14 ≤ level_prefix < 15, and level_prefix = 15. The lengths for level encoding procedures are the length of the encoded codeword, and those for level decoding procedures is the suffix length of decoding codeword which is transmitted to codeword boundary detector to calculate the codeword boundary. The level_out is the codeword value for level encoding and that is the value of level_code for level decoding. According to level_code, we can get the value of the decoded level shown in Figure 2-10. Based on these calculations, we can simplify the complexity of level encoding and decoding, and this architecture can help us handle the parallel input bitstream for level decoding and integrate level encoding/decoding to an area-efficient level codec system. The level decoding/encoding procedures and the corresponding examples are shown in Figure 4-3 and Figure 4-4.

```
Decoding procedures – assume the decoding codewords "00001101000…" and suffix_length is 3.

1) Count leading zeros and fetch level_prefix.
        leading zeros = 4 => level_prefix = 4;

2) Evaluate the escape case and fetch level_suffix according to suffix_length.
        level_prefix < 14 => escape0 = 0;
        level_prefix < 15 => escape1 = 0 & escape2 = 0;
        suffix_length = 3 => suffix = 3'b101;

3) According to suffix_length, escape cases, level_prefix, and level_suffix, we can get the decoded suffix_length, level_out0, level_out1, and sign.
        suffix_length != 0 && escape2 = 0 => length = suffix_length = 3;
        suffix_length != 0 => level_out0 = level_prefix << suffix_length = 4'b0100 << 3 = 7'b0100_000 = 32;
                              leve_out1 = level_suffix = 3'b101 = 5;
        sign = level_suffix[0] = 1'b1;

4) Extract the lengths of level_out0 and level_out1 the same as the word length of levels which is 16-bit.
        level_out0' = 16'b0000_0000_0010_0000 = 32;
        level_out1' = 16'b0000_0000_0000_0101 = 5;

5) Calculate level_code by adding level_out0' and level_out1' and derive level according to sign and level_code.
        level_code = level_out0' + level_out1' = 16'b0000_0000_0010_0101 = 37;
        sign = 1 => level = ~level_code >> 1= 16'b1111_1111_1110_1101 = -19;
```

Figure 4-3 : Level decoding procedures and the corresponding examples

```
Encoding procedure – assume the encoding level is 14 and suffix_length is 1.

1) Calculate the absolute value of level, the escape value according to suffix_length, and sign.
        level = 14 => |level| = 14;
        escape = 15 << suffix_length = 15 << 1 = 30;
        level >= 0 => sign = 0;

2) Evaluate the escape cases according to the absolute value of level.
        |level| = 14 => escape0 = 1, escape1 = 0, escape2 = 0;

3) According to suffix_length, escape cases, and the absolute value of level, we can get the encoded codeword length and level_out0 and level_out1.
        suffix_length == 0 && escape2 == 0 => codeword length = (|level| - 1) >> shift + suffix_length + 1 = (14 - 1) >> 0 + 1 + 1 = 16.
        suffix = (|level| - 1) & (~((0xffffffff)<<shift)) = (4'b1101) & (0x0000_0000) = 0;
        escape2 == 0 => level_out0 = 1 << suffix_length = 1 << 1 = 2;
                        level_out1 = {suffix,sign} = 0;

4) Extract the lengths of level_out0 and level_out1 the same as the codeword length.
        level_out0' = 16'b0000_0000_0000_0010 = 2;
        level_out1' = 16'b0000_0000_0000_0000 = 0;

5) Calculate codeword by adding level_out0' and level_out1'.
        codeword = level_out0' + level_out1' = 16'b0000_0000_0000_0010;
```

Figure 4-4 : Level encoding procedures and the corresponding examples

The architecture of the proposed level codec system is shown in Figure 4-5. The level decoding and encoding procedures can work on this codec system. When executing level encoding, the valid outputs are codeword and length; on the other hand, the valid outputs are level and length. The results of |level| - 8, |level| - 16, and |level| - escape can be derived from the calculations of escape cases. The codeword boundary detector always sends 12-bit bitstream for level decoding, and according to level_prefix and suffix_length the system will fetch the wanted level_suffix with correct length. The information of suffix_length is given from the suffix_length generator, and the architecture is the same as that in Figure 2-11. Therefore, we don't describe it here. The three components, decoder for length, decoder for level_out0, and decoder for level_out1 implement the calculations shown in Figure 4-2 with PLA architectures. The three adders in the left part in Figure 4-5 calculate the escape cases of level encoding and decoding. The two input signals, level_encoding and level_decoding, are not only the selecting signal, but also enable signals to open or close the level codec system. The component to get the absolute value of the input is to do 2's complement or pass the original input according to the most significant bit (msb) of the input which can judge the input value is positive or negative. That is the approximate introduction of the proposed level encoding and decoding architecture.

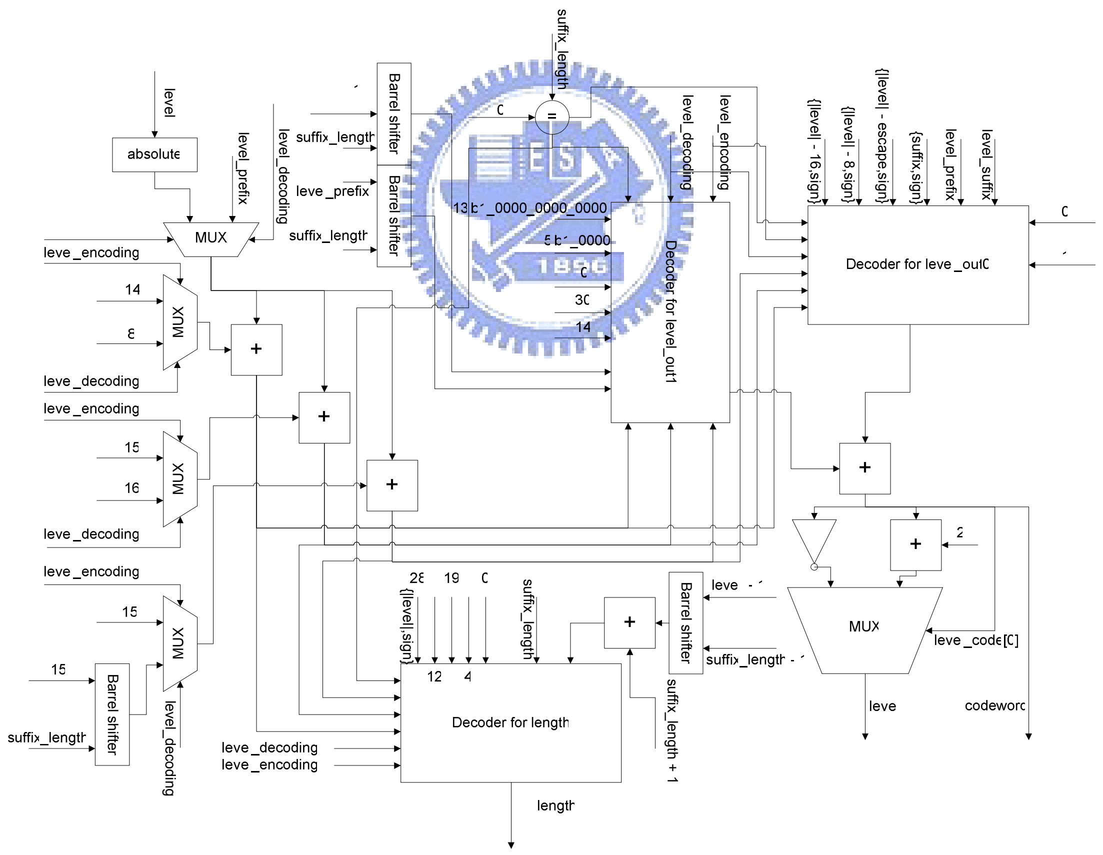


Figure 4-5 : Architecture of level decoding/encoding

4.1.2. Run_before Efficient Coding

run_before	zerosLeft						
	1	2	3	4	5	6	>6
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

Table 4-1 : Table for run_before

The run_before table is shown in Table 4-1. Even if we can get good performance with PZTP to realize run_before table, we hope to find the easier and more efficient method to implement run_before codec system with parallel input bitstream and combine the encoding part. After observing the run_before table, we can find the numerical relation for run_before decoding and encoding shown in Figure 4-6. No matter the decoding or encoding procedures, we can divide the run_before table into three groups, which are zerosLeft < 6, zerosLeft = 6, and zerosLeft > 6. Besides, the calculations in each group are similar. For example, when zerosLeft is equal to 6, run_before is the result of adding codeword and one in decoding processes, and codeword is the difference of run_before and one. Such relation helps us to complete the efficient coding for run_before table.

```
decoding

zerosLeft > 6
        if (leading zeros > 2)
                run_before = leading zeros + 4
        else
                run_before = ~codeword (3 bits)
zerosLeft = 6
        codeword >= 6  run_before = 0
        codeword <= 1  run_before = codeword + 1
        codeword == 3  5  run_before = codeword
        codeword == 2  4  run_before = codeword + 2
zerosLeft < 6
        if (leading zeros == 0)
                run_before = ~codeword
        else
                run_before = zerosLeft - codeword
```

```
encoding

zerosLeft > 6
        if (run_before < 7)
                codeword = ~run_before
        else
                codeword = 1
zerosLeft = 6
        run_before == 0  codeword = 3
        run_before <= 2  codeword = run_before - 1
        run_before == 3  5  codeword = run_before
        run_before == 4  6  codeword = run_before - 2
zerosLeft < 6
        if (run_before < 2)
                codeword = ~run_before
        else
                codeword = zerosLeft - run_before
```

Figure 4-6 : The numerical calculations of run_before encoding and decoding

The architecture of the proposed run_before codec system is shown in Figure 4-7. We can use the architecture of run_before efficient coding instead of look-up table method. The advantage of the proposed architecture is the major function units can be shared for the encoding and decoding procedures. However, if we implement the run_before codec system with look-up table, we have to build two tables for both procedures.

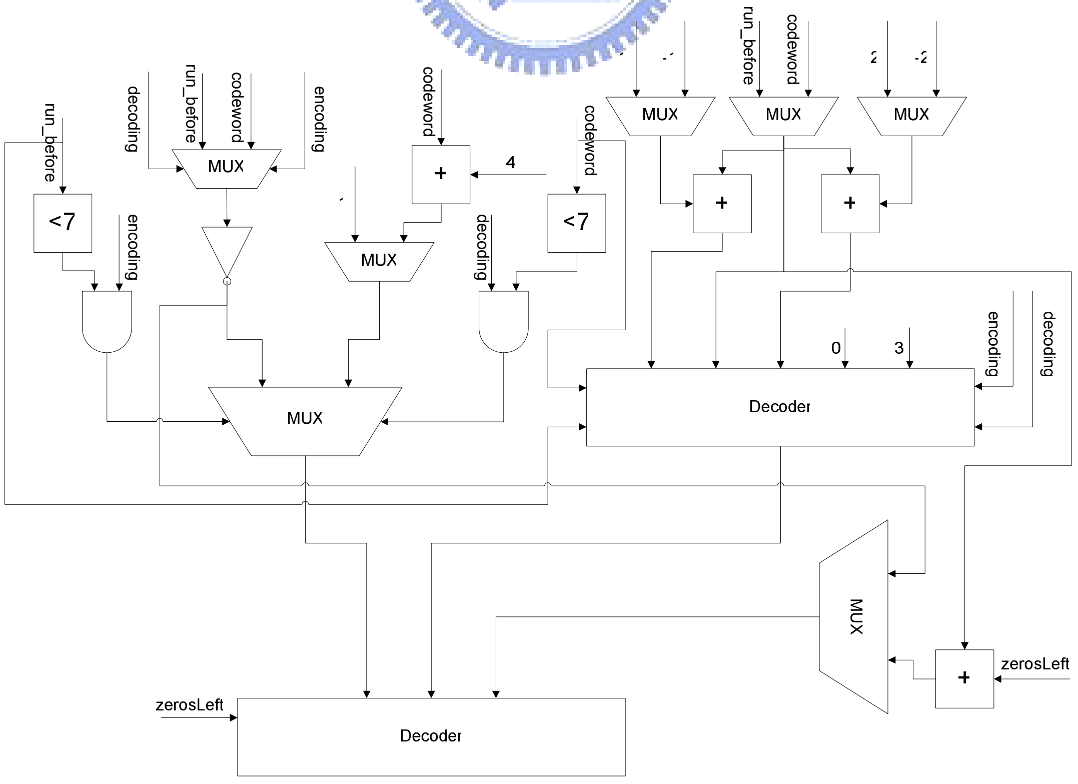


Figure 4-7 : Architecture of run_before codec system

4.2. Zero skipping and proposed symbols constructor

Code	Element	Value	Output array
0000100	coeff_token	TotalCoeff = 5, TrailingOnes = 3	Empty
0	T1 sign	+	1
1	T1 sign	-	-1,1
1	T1 sign	-	-1,-1,1
1	level	+1	1,-1,-1,1
0010	level	+3	3,1,-1,-1,1
111	total_zeros	3	3,1,-1,-1,1
10	run_before	1	3,1,-1,-1,0,1
1	run_before	0	3,1,-1,-1,0,1
1	run_before	0	3,1,-1,-1,0,1
01	run_before	1	3,0,1,-1,-1,0,1

Figure 4-8 : An example of decoding procedures of CAVLC

Figure 4-8 shows an example of decoding procedures of CAVLC. We can see the processes of constructing the DCT coefficients in zigzag order. Generally, we will arrange the DCT coefficients after decoding all run_befores. Such method will take additional cycles whose value is the same as the value of TotalCoeff to arrange the DCT coefficients. If the decoded run_before is derived, we arrange the coefficients in the next cycle, and we can save a few cycles to arrange the DCT coefficients. Before executing the proposed symbols construction, we have to know the location of last non-zero coefficient in the coefficients storage. According to TotalCoeff and total_zeros, we can calculate the location of the last DCT coefficient. The procedures of proposed symbols construction and the corresponding example are shown in Figure

4-9. In Figure 4-9, cycle means the cycle of symbols construction and run_before is being decoded in cycle 1 ~ 4, run_before is the value of decoded run_before in the present cycle, level_count represents the pointer to the levels buffer, coeff_count means the pointer to the coefficients buffer, and coeff_buffer records the values of coefficients buffer in the next cycle. The default value of coeff_count is the sum of TotalCoeff and total_zeros minus one. The sum of TotalCoeff and total_zeros means the total number of decoded symbols including non-zero and zero coefficients, so according the sum of TotalCoeff and total_zeros we can know the location of last non-zero coefficient in coeff_buffer. In the first cycle, level_count equal to 4 maps the level is 1. Therefore, we put 1 to coeff_buffer at the location coeff_buffer 7, and the next coeff_count is the result of subtracting current coeff_count and 1. At the same time, the decoded run_before is 1, and the next coeff_count also has to subtract the value of run_before, so the next coeff_count is 5. Repeating the above steps, finally we can get the DCT coefficients in zigzag order.

cycle	run_before	level_count	coeff_count	coeff_buffer 0 ~ 15
1	1	4	7	0000_0001_0000_0000
2	0	3	6 - 1	0000_0-101_0000_0000
3	0	2	4	0000_-1-101_0000_0000
4	1	1	3	0001_-1-101_0000_0000
5	N.A.	0	2 - 1	0301_-1-101_0000_0000

Figure 4-9 : The proposed symbols construction for example in Figure 4-8.

However, the proposed symbols construction is not the optimal solution. When the decoded run_before is equal to 0, the next coefficient location can be predicted, even if we don't decode the run_before. That is, if we skip the zero run_before and decode the next run_before, we can still store the levels into correct locations in coefficients buffer. That is not difficult, and when calculating the results of level_count and coeff_count, we take the number of zero-skipping run_befores into consideration. The example of the proposed symbols construction with zero-skipping is shown in Figure 4-10.

cycle	run_before	level_count	coeff_count	coeff_buffer 0 ~ 15
1	1	4	7	0000_0001_0000_0000
4	(0), (0), 1	3 - 2	6 – 1 - 2	0001_-1-101_0000_0000
5	N.A.	0	2 - 1	0301_-1-101_0000_0000

Figure 4-10 : An example of the proposed symbols construction with zero-skipping

The final problem is how to realize the function unit to detect the condition of zero-skipping. Figure 4-11 shows the run_before table mapping to zero run_before under different zerosLefts. We can find that the codewords of zero run_before are “1”, “11”, and “111”. Therefore, the realization of zero-skipping detector is quite easy, because we already design a leading-one counter in the codeword boundary detector for MPEG-2 codewords. Here, we only use that leading-one counter and add another decoder whose inputs are leading ones and zerosLeft, and we can get the information about the number of zero-skipping run_befores.

run_before	zerosLeft						
	1	2	3	4	5	6	>6
0	1	1	11	11	11	11	111

Figure 4-11 : The run_before table mapping to zero run_before

4.3. Summary

proposed method	I-frame	P-frame	frame
level efficient coding	40%	17%	29%
run_before efficient coding	4%	12%	8%
symbols construction	14%	12%	13%
zero skipping	4%	5%	4%

Table 4-2 : Throughput improvement of each proposed method, foreman QP = 10

Table 4-2 shows the improvement of throughput for each proposed approach, when we decode the picture, foreman, and the QP is equal to 10. We can see the effect is the best when applying level efficient coding, and the method can save about 40% throughput when decoding an I-frame. Besides, run_before efficient coding has more performance for P-frame than I-frame, because the blocks of P-frame have more zero coefficients than those in I-frame. Symbols construction also has good improvement both for I-frame and P-frame. However, the effect of zero skipping is not so significant. We consider that the number of zero run_befores is not so much in this picture. Therefore, we decode another picture, mobile, and set QP is 28. The improvement of throughput is shown in Table 4-3.

proposed method	I-frame	P-frame	frame
level efficient coding	27%	5%	22%
run_before efficient coding	6%	12%	7.5%
symbols construction	14%	5%	12%
zero skipping	4%	3%	4%

Table 4-3 : Throughput improvement of each proposed method, mobile QP = 28

Table 4-3 shows the improvement of throughput for each proposed approach, when we decode the picture, foreman, and the QP is equal to 10. The proposed approach, level efficient coding, still has excellent performance for I-frame, but the performance for P-frame is not so good. The proposed approach, run_before efficient coding, also has good performance in P-frame, and symbols construction provides much improvement in I-frame. However, zero-skipping approach still has not good performance. Blessedly, the hardware cost of zero-skipping is acceptable, although the improvement of throughput is not good enough.

Chapter 5. Implementation Results and Conclusion

5.1. Implementation Results

Figure 5-1 and Figure 5-2 show the encoding throughput of the proposed VLC group-based codec system with the H.264/AVC standard C code, JM 9.2, and in Figure 5-1 we encode the picture, mobile.yuv; on the other hand, the picture is foreman.yuv. The proposed VLC group-based codec system can support H.264/AVC main profile @5.1, when QP is equal to 28 in Figure 5-1, and Figure 5-2. Table 5-1 shows the average encoding cycles per MB in the proposed design.

QP	mobile	foreman
10	368	329
12	353	292
16	320	226
20	278	156
24	227	102
28	165	69
32	114	50
36	86	35
40	68	23
average	220	142

Table 5-1 : The average encoding cycles per MB in the proposed design

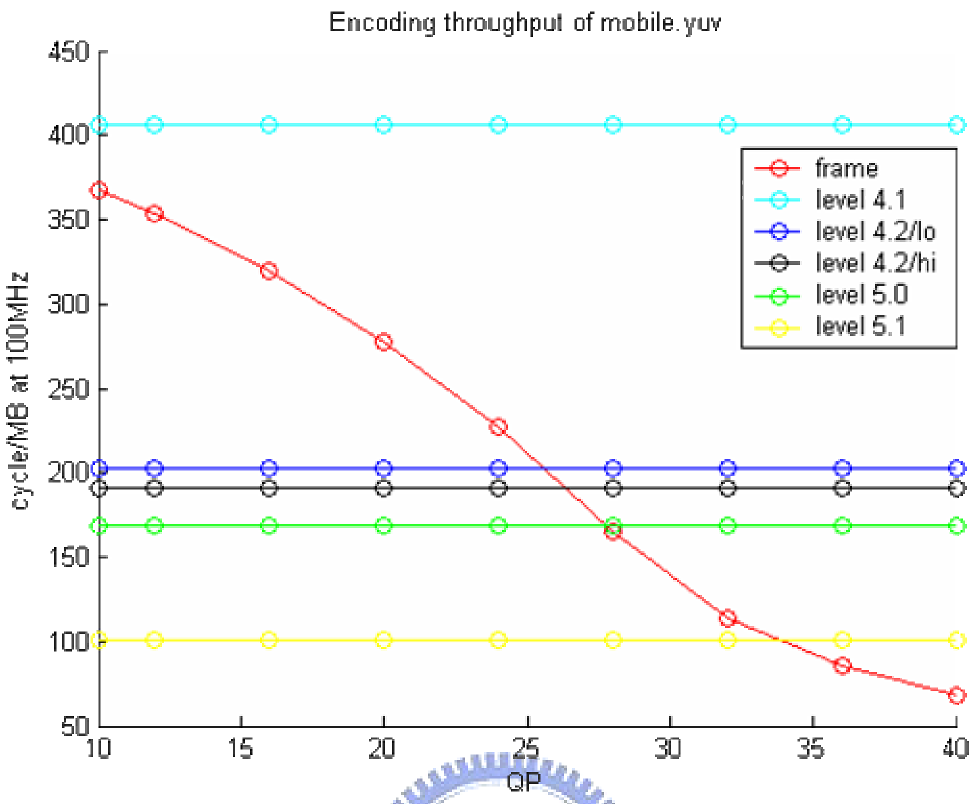


Figure 5-1 : The encoding throughput of proposed design running mobile.yuv

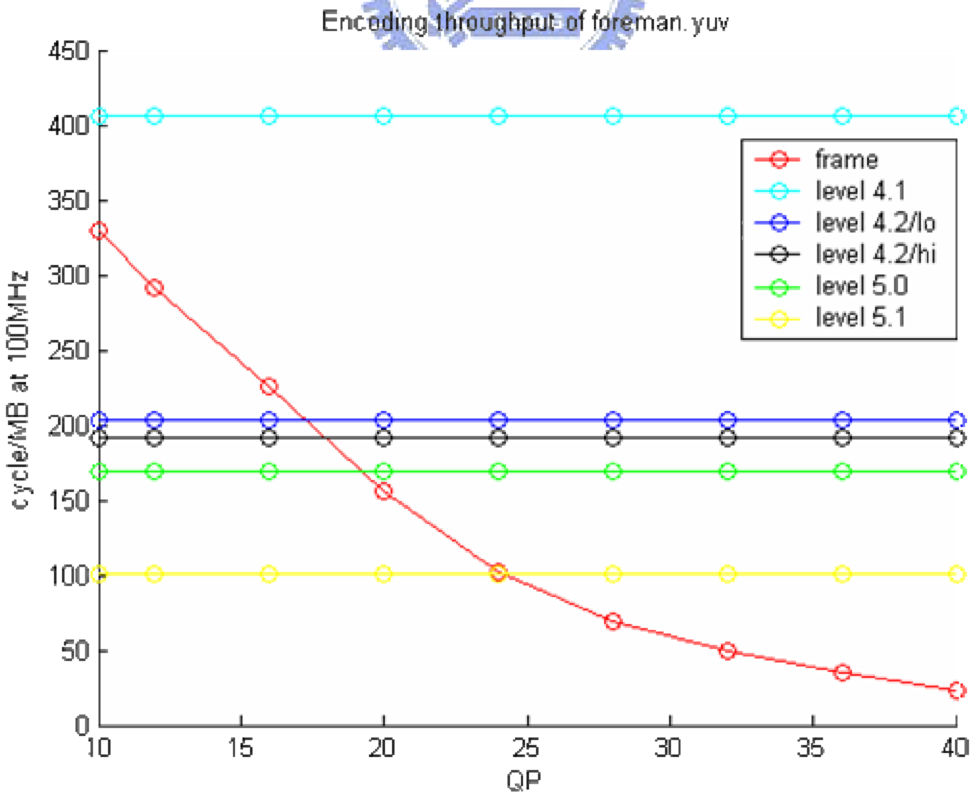


Figure 5-2 : The encoding throughput of proposed design running foreman.yuv

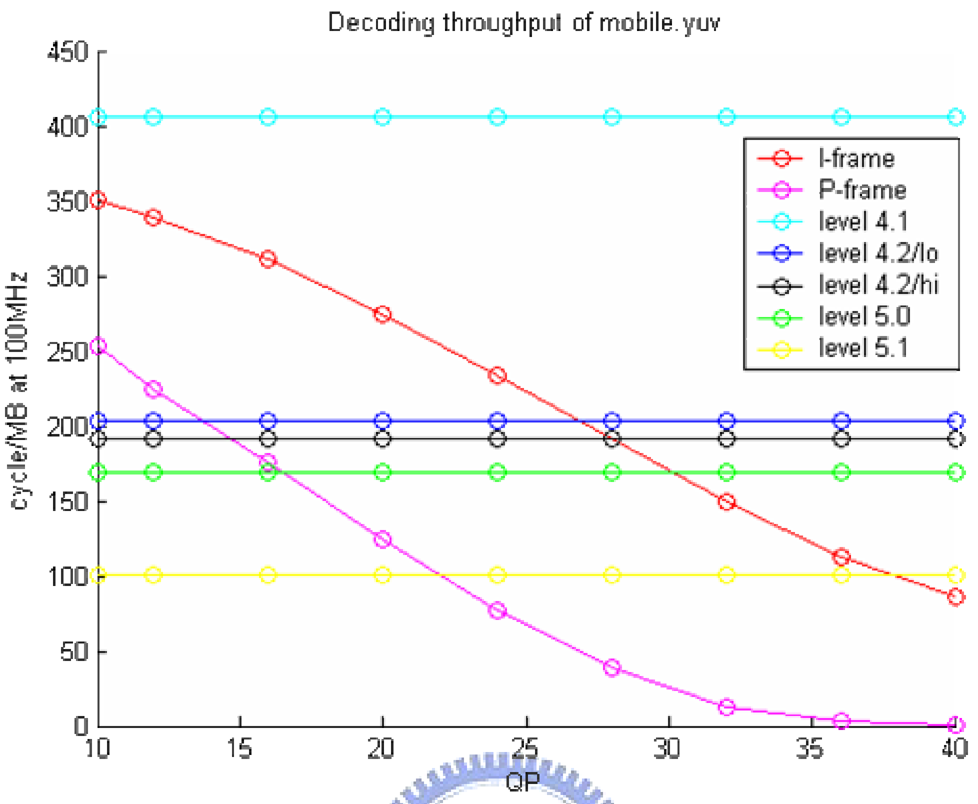


Figure 5-3 : The decoding throughput of proposed design running mobile.yuv

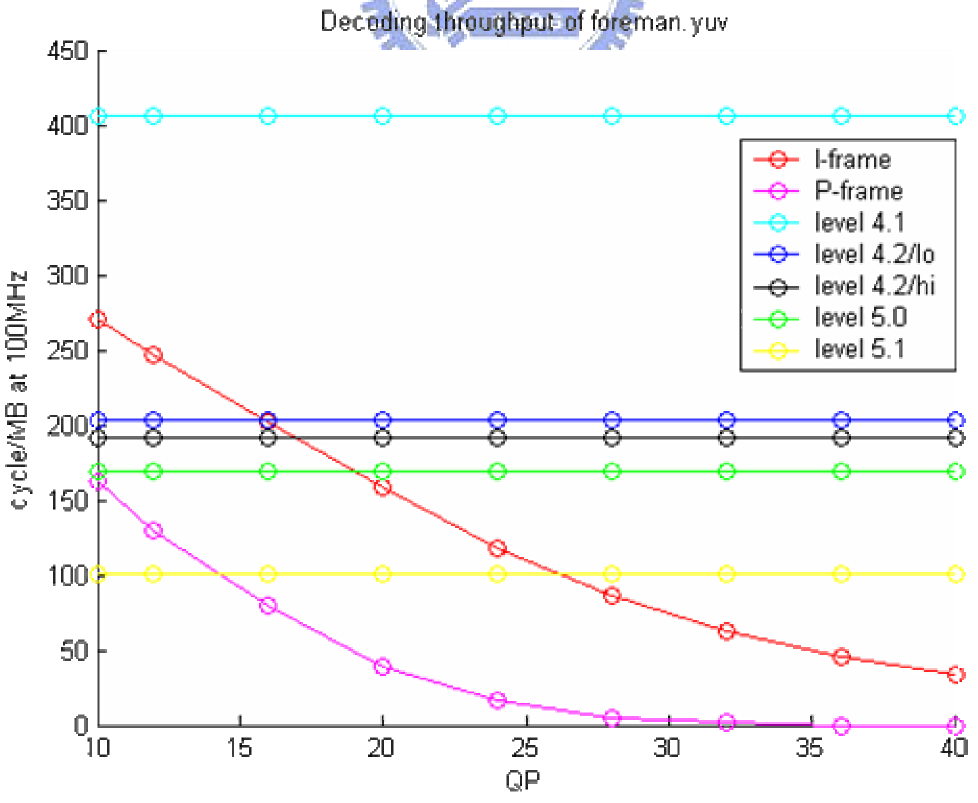


Figure 5-4 : The decoding throughput of proposed design running foreman.yuv

Figure 5-3 and Figure 5-4 show the encoding throughput of the proposed VLC group-based codec system, and in Figure 5-3 we decode mobile.yuv, and foreman.yuv in Figure 5-4. Usually, compared to the decoding throughput, we often consider I-frame of a decoded picture. In Figure 5-3, the decoding throughput of I-frame can reach the standard of H.264/AVC main profile @5.0 when QP is 32 and in Figure 5-4 the proposed VLC codec system can meet that when QP is 28. Therefore, the proposed VLC codec system can support H.264/AVC main profile @5.0.

	Chien[2]	Chen [1]	Yu[5]	Proposed
Technology	0.18um	0.18um	0.18um	0.13um
Gate Count	9724	17635	13192	20357
Clock Frequency	125 MHz	100 MHz	125 MHz	125 MHz
Encoding/Decoding	Encoding	Encoding	Decoding	Decoding : 8554 Encoding : 5519 Shared 6284
Target Format	HD1080 30fps	HD 1080 30fps	Main Profile @5.0	Main Profile @5.0

Table 5-2 : Comparison of the proposed design with others

In implementing the proposed CAVLC codec system, we performed logic synthesis on the proposed design according to a 0.13um CMOS technology. The comparison of the proposed design with other is shown in Table 5-1. Design [1] contains a bitstream packer which packs the codewords produced by symbol encoders, the packing of bitstream headers and Exp-Golomb.

In MPEG-2, the only difference of the throughput from the conventional group-based VLC codec design is the decoding procedure, because we have to access the symbol group memory when decoding a MPEG-2 symbol in our proposed design. However, under well pipelined architecture, such difference is not obvious. Besides, the encoding procedures in the proposed design have the same steps as the conventional group-based VLC codec design, so of course the throughput is the same as the conventional one. The simulation results are shown in Table 5-3. We can see the average symbol rate of encoding process is 99.98 Msps at 100 MHz-clock rate and the average symbol rate of decoding process is 99.8 Msps at the same clock rate. Some overheads are introduced due to stalls of the bitstream FIFOs.

image: (4:2:2) @ 1920 X 1080 / simulation results		
# of bitstream (bit)	3439392	1912640
# of symbols	590302	252817
Encoding cycle	590348	252864
Decoding cycle	591484	253323

Table 5-3 : Simulation results based on HDTV systems (I-frame) in MPEG-2

5.2. Conclusion

In this thesis, we propose one low power and hardware cost VLC decoder for dual standards, MPEG-2 and H.264/AVC. Compared to [4], we reduce 30% hardware cost in H.264/AVC CAVLD. The hardware cost of the proposed dual-standard VLD is 7683 gate-count and the power is 1.719 mW for MPEG-2, 1.302 mW for H.264/AVC baseline@3.0 I-frame, and 1.376 mW for H.264/AVC baseline@3.0 P-frame at 100 MHz.

Besides, we proposed another group-based VLC codec system for dual standards, MPEG-2 and H.264/AVC. According the group-based, level efficient coding, run_before efficient coding, the proposed symbols construction, and zero-skipping, we design a VLC codec system which can support H.264/AVC main profile @5.0 with 20357 gate counts at 100 MHz. Each proposed method can improve the percentage of throughput shown in Table 5-4 and Table 5-5. Compared to the conventional VLC group-based VLC codec system, the proposed design reduce 16% memory usage.

proposed method	I-frame	P-frame	frame
level efficient coding	40%	17%	29%
run_before efficient coding	4%	12%	8%
symbols construction	14%	12%	13%
zero skipping	4%	5%	4%

Table 5-4 : Throughput improvement of each proposed method, foreman QP = 10

proposed method	I-frame	P-frame	frame
level efficient coding	27%	5%	22%
run_before efficient coding	6%	12%	7.5%
symbols construction	14%	5%	12%
zero skipping	4%	3%	4%

Table 5-5 : Throughput improvement of each proposed method, mobile QP = 28

5.3. Future Work

The hardware cost is a problem for the proposed group-based VLC codec design, because under such performance in throughput the hardware cost is not efficient enough. Therefore, hardware cost reduction can be a target to make effort. Besides, the power issue is always the problem of the group-based design. How to reduce the power consumption of the proposed group-based VLC codec design is another point. Perhaps, we can solve this problem with memory hierarchy, because the codewords of VLC tables are the representation of the occurring probabilities.

On the other hand, the mobile devices are used generally. In the process of the wireless communication, the problem of receiving error bitstream due to the noise is serious. It will result in the error blocks decoded, and the picture decoded maybe has mosaics. Therefore, to develop the error resilience approaches very important.

Reference

[1] T. C. Chen, Y. W. Huang, C. Y. Tsai, B. Y. Hsieh, and L. G. Chen, "Dual-block-pipelined VLSI architecture of entropy coding for H.264/AVC baseline profile", Proc. International Symposium on VLSI Design, Automation and Test (VLSI-DAT), pp. 271-274, 2005.

[2] C. D. Chien, K. P. Lu, Y. H. Shih, and J. I. Guo "A High Performance CAVLC Encoder Design for MPEG-4 AVC/H.264 Video Coding Applications", in Proc. ISCAS, 2006.

[3] Wu Di, Gao Wen, Hu Mingzeng, and Ji Zhenzhou, "A VLSI Architecture Design of CAVLC Decoder" Proc. 5th International Conference on ASIC, Vol. 2 pp. 962-965, 21-24 Oct. 2003..

[4] H. C. Chang, C. C. Lin, J. I. Guo, "A Novel Low-Cost High-Performance VLSI Architecture for MPEG AVC/H.264 CAVLC Decoding", in Proc, ISCAS, pp. 6110 – 6113, 2005.

[5] K. S. Yu and T. S. Chang "A Zero-Skipping Multi-symbol CAVLC Decoder for MPEG-4 AVC/H.264" in Proc. , ISCAS, 2006

[6] B. J. Shieh, Y. S. Lee, C. Y. Lee, "A New Approach of Group-Based VLC Codec System", in Proc. , ISCAS, Vol. 4, pp. 609 - 612, 28-31 May 2000.

[7] B. J. Shieh, Y. S. Lee, C. Y. Lee, "A New Approach of Group-Based VLC Codec System with Full Table Programmability", in Proc. , ISCAS, Vol. 2, pp. 210 – 221, Feb 2001.

[8] T. M. Liu, T. A. Lin, S. Z. Wang, W. P. Lee, K. C. Hou, J. Y. Yang and C. Y. Lee, "An 865-uW H.264/AVC Video Decoder for Mobile Applications", in Proc. ASSCC, 2005.

[9] T. M. Liu, T. A. Lin, S. Z. Wang, W. P. Lee, K. C. Hou, J. Y. Yang and C. Y. Lee, "A 125-uW, Fully Scalable MPEG-2 and H.264/AVC Video Decoder for Mobile Applications", in Proc. ISSCC, 2006.

[10] Joint Video Team, 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.

簡　歷

姓　　　　名：楊俊彥

出　　生　　地：台灣省高雄市

出　生　日　期：民國七十一年五月二十九日

學歷：

- 2004 年 9 月~2006 年 7 月　國立交通大學電子研究所系統組碩士班
- 2000 年 9 月~2004 年 7 月　國立交通大學電子工程學系
- 1997 年 9 月~2000 年 6 月　高雄市立高雄高級中學