國立交通大學

電子工程學系　電子研究所碩士班

碩　士　論　文

低成本高效率內容適應性可變長度編碼器之設計

A Low Cost and High Throughput CAVLC Encoder Design

學生　:吳 秈 璟

指導教授:張添烜 博士

中 華 民 國 九十六 年 七 月

低成本高效率內容適應性可變長度編碼器之設計

A Low Cost and High Throughput CAVLC Encoder Design

研 究 生：吳秈璟　　Student ：Sian-Jing Wu

指導教授：張添烜 博士　　Advisor ：Dr. Tian-Sheuan Chang

國 立 交 通 大 學

電子工程學系 電子研究所 碩士班

碩 士 論 文

A Thesis

Submitted to Institute of Electronics

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for Degree of Master of Science

in

Electronic Engineering

July 2007

Hsinchu, Taiwan, Republic of China

低成本高效率內容適應性可變長度編碼器之設計

學生 ： 吳私璟　　　　　　　　　　　　指導教授 ： 張添烜 博士

國立交通大學

電子工程學系　　電子研究所碩士班

摘要

本論文提出一個低成本高效率的內容適應性可變長度編碼器。本論文的動機是為了補償編碼區塊樣式先決的低效能以達到高效率，使得我們所提出的內容適應性可變長度編碼器能支援每秒處理30張1080p畫面。此外，在如此高效率之下，我們必須維持少量的邏輯閘。編碼區塊樣式先決能跳過一些零方塊的編碼流程來提高效率。然而，我們的統計數據指出有大量的零係數無法被編碼區塊樣式先決偵測到，使得許多的週期數被浪費在零係數。在我們的設計裡，除了採用編碼區塊樣式先決，我們還使用可以直接對非零係數作編碼的新奇架構，以避免花費時間在零係數：零方塊碼字表與非零索引表。

當我們所提出的內容適應性可變長度編碼器在一個週期內讀取一個方塊的所有係數時，非零索引表同時記錄那些係數是非零的。然後非零索引表會分辨這個方塊是否為全零。假如這方塊是全零，零方塊碼字表可以在不用跑完整套內容適應性可變長度編碼流程的情況下，直接產生這個方塊的全部碼字。另一方面，如果這個方塊含有至少一個非零係數，非零索引表使用組合電路找出非零係數的位置，使得零係數被忽略。再者，

當非零索引表鎖定一個非零係數時，這個非零係數的碼字會直接被連接到H.264/AVC的位元串流。因此，我們不需要額外的緩衝存儲器來儲存這個非零係數，使得少量的邏輯閘被消耗。

最後，基於聯華電子點一三微米製程，我們所提出的設計在145 MHz的工作時脈之下，消耗了9.03 K個邏輯閘，且可支援每秒處理30張1080p的畫面。和其它的設計相較之下，我們可以節省61%的邏輯閘與29%的週期數。

A Low Cost and High Throughput CAVLC Encoder Design

Student : Sian-Jing Wu   Advisor : Dr. Tian-Sheuan Chang

Institute of Electronics

National Chiao Tung University

ABSTRACT

This thesis proposes a low cost and high throughput CAVLC encoder. The motivation is to compensate the inefficiency of CBP Look-Ahead to achieve higher throughput such that the proposed CAVLC encoder can support 1080p at 30 fps. Moreover, under such high throughput, we must keep logic gate count low. CBP Look-Ahead can skip encoding flow of some zero blocks such that throughput can be improved. However, our statistics show that abundant zero coefficients cannot be detected by CBP Look-Ahead such that many cycle counts are wasted on the zero coefficients. In our design, we use novel direct significance encoding architectures, as well as CBP Look-Ahead, to avoid spending time on zero coefficients: Zero-block Codeword Table and Nonzero Index Table.

Nonzero Index Table concurrently records which coefficients are significant while the proposed CAVLC encoder is reading all coefficients of a block in one cycle. Then Nonzero Index Table determines whether the block is all-zero. If the block is all-zero, Zero-block

Codeword Table will generate the overall codeword of the block without going through the whole CAVLC encoding flow. On the other hand, if the block consists of at least one significant coefficient, Nonzero Index Table uses combinational circuits to locate significant coefficients such that zero coefficients are ignored. In addition, while Nonzero Index Table is aiming for a significant coefficient, the codeword of the significant coefficient can be directly concatenated into the H.264/AVC bit-stream. Hence, we do not need additional buffers to store the significant coefficient such that small logic gate count is consumed.

Eventually, based on 0.13um UMC technology, the proposed design can support 1080p at 30 fps while consuming 9.03 K gate count at 145 MHz. Compared with other designs, we can reduce 61% logic gate count and 29% cycle count.

誌　　　謝

首先，要感謝我的指導教授－張添烜博士，在張教授的指導之下，我學習到作研究正確的方法與態度。此外，張教授所提供的優良實驗室環境和軟硬體資源，使得我的研究能夠順利進行，才有這本論文的誕生。

同時也要謝謝我的口試委員們，交大研發長李鎮宜教授、清大電機陳永昌教授，感謝各位在百忙之中抽空前來指導我，各位教授的寶貴意見讓本論文得以更加完備。

接著，我要感謝實驗室的夥伴。謝謝林佑昆學長，領導我們走向晶片下線成功的坦途，學長在晶片下線期間努力建構與維護測試平台，並時常與我交流驗證的心得，使我的驗證能夠順利進行。謝謝李國龍學長和張彥中學長，給予課程和研究上的指導，讓研究得以順利進行。謝謝李得瑋同學、林嘉俊同學和郭子筠同學，你們所設計的高效能演算法，得以讓我們團隊的晶片達到下線的規格要求。也謝謝張瑋城學弟和戴瑋呈學弟，在晶片下線期間，分擔部分的驗證工作與後段佈局流程。此外，要謝謝廖英澤同學，宇晟，宗憲，景竹等學弟，你們的幫忙讓我的實驗室生活能順利渡過。所有的一切，都是我在交大的寶貴回憶。

最後，我要感謝我的家人們，你們的默默支持，是我能夠完成學業的最大動力。

在此，我謹把這篇論文獻給所有愛我與我愛的人。

Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Motivation

H.264/AVC [1] is the latest standard for video coding. H.264/AVC is the result of the collaboration between the ISO/IEC Moving Picture Experts Group and the ITU-T Video Coding Experts Group [2]. H.264/AVC is designed to address a large range of applications, such as storage, entertainment, multimedia short message, videophone, videoconference, HDTV broadcasting, and Internet streaming. Compared to previous video coding standards, H.264/AVC can achieve 50% bit-rate reduction under the same quality [10]. The better compression efficiency results from innovative coding tools such as multiple reference frames, variable block size motion estimation, and in-loop de-blocking filter [3].

The essence of H.264/AVC is block-based motion estimation transform coding [3]. For H.264/AVC Baseline profile, context-based adaptive variable length coding (CAVLC) is used to encode quantized transform coefficients of the residual images [2]. Compared with entropy coders of previous standards, CAVLC removes more statistical redundancy by switching VLC tables according to previously transmitted symbols [2]. However, coding is not started until syntax elements are extracted by scanning all coefficients of a block such that throughput is very low [3]. To support high-end applications such as 1080p at 30 fps, we must improve throughput of CAVLC encoder.

H.264/AVC specification stipulates that some all-zero blocks can be skipped according to the coded block pattern (CBP) [1]. For example, Chen [3] uses CBP Look-Ahead to improve throughput. However, our statistics show abundant zero coefficients are not covered by the CBP. Therefore, we propose two zero-skipping methods to avoid wasting cycles on zero coefficients such that throughput is improved. The resulting architecture not only has a high throughput but also consumes small area.

1.2 Organization of thesis

This thesis is organized as follows. In Chapter 2, we present the algorithm of CAVLC in H.264/AVC. Chapter 3 presents the proposed design with the two throughput enhancement methods. Moreover, we present related statistics to prove that the methods are necessary. Chapter 4 presents simulation results, implementation results, and comparison with other designs. Chapter 5 concludes this thesis.

Chapter 2. Overview of CAVLC in H.264/AVC

2.1 Overview of an H.264/AVC encoder

Fig. 1 shows the block diagram of an H.264/AVC encoder. An input frame is processed in units of a macro-block [16]. The data flow in Fig. 1 can be divided into the following three steps:

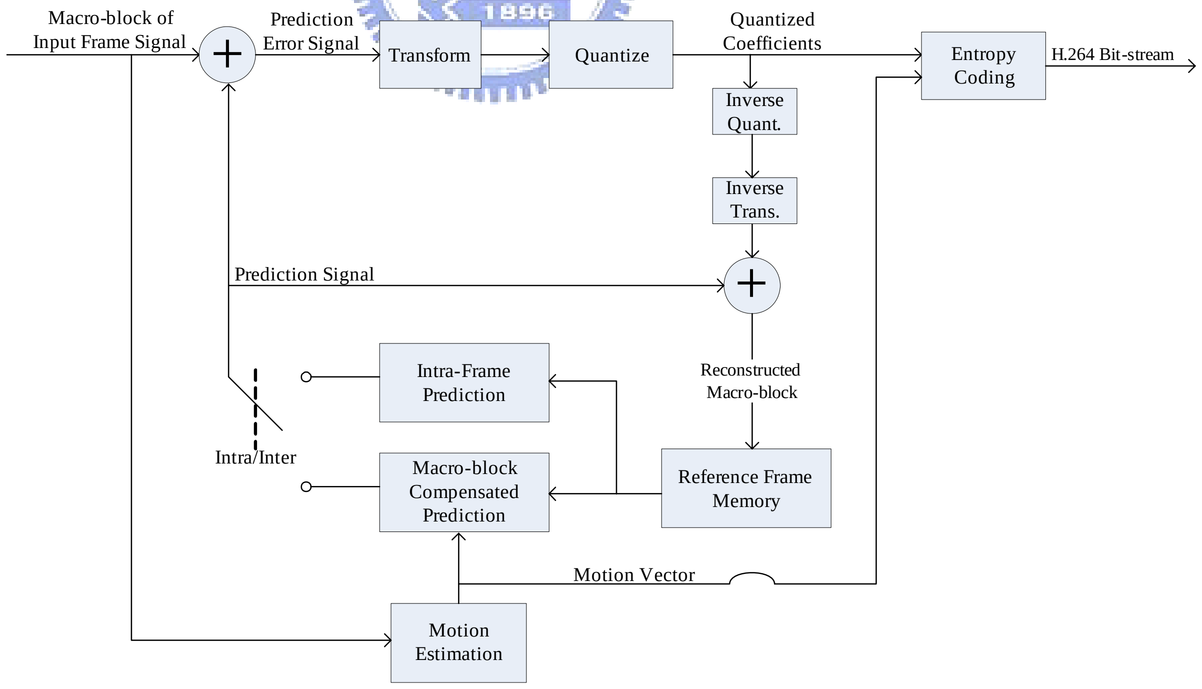


Fig. 1 Block Diagram of an H.264/AVC encoder [2]

1. First, we calculate the prediction signal of the macro-block. In general, there are two prediction modes: Intra and Inter. In Intra mode, the prediction signal is calculated according to pixels in the current frame, which have been encoded, reconstructed and stored into Reference Frame Memory. In Inter mode, we use Motion Estimation to estimate the motion vectors, which refer to the corresponding position of the macro-block in an already transmitted (to decoder) frame stored in Reference Frame Memory [2]. Then the prediction signal is generated by Macro-block Compensated Prediction. Note that the motion vectors must be encoded into H.264/AVC bit-stream by Entropy Coding.
2. Second, we subtract the prediction signal from the macro-block to obtain the prediction error signal. Then the prediction error signal is transformed and quantized to generate quantized coefficients, which is compressed into H.264/AVC bit-stream by Entropy Coding. H.264/AVC has two major entropy coding tools: CAVLC for the Baseline Profile and CABAC for the High Profile [1].
3. Third, the quantized coefficients are inverse quantized, inverse transformed, and added to the prediction signal. The result is the reconstructed macro-block which is stored into Reference Frame Memory in order to calculate the prediction signal of the future macro-block.

2.2 Overview of block types

2.2.1 Partitions of a macro-block

As mentioned in section 2.1, Inter mode or Intra mode finds prediction signal of a macro-block [2]. The residual data is obtained by subtracting the prediction signal from the macro-block. Then we apply transform matrices on the residual data to obtain so-called quantized transform coefficients [2]. The coefficients of a macro-block can be divided into three components as follows: luminance Y, chrominance U, and chrominance V. The component Y comprises 16 by 16 coefficients. The component U comprises 8 by 8 coefficients, and the component V does, too.

The main purpose of CAVLC is to encode the coefficients. Before encoded by CAVLC, the three components of a macro-block are individually divided into several smaller blocks. For the component Y, we divide it into sixteen 4x4 sub-blocks, as Fig. 2 shows. Each of the sixteen sub-blocks has a DC coefficient. If Intra_16x16 [1] is the prediction mode of the macro-block, we separate the DC coefficients to form another 4x4 sub-block, as Fig. 3 shows.

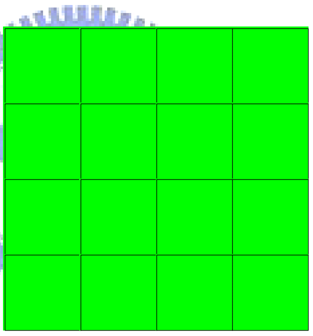


Fig. 2 Division of Y component in a macro-block

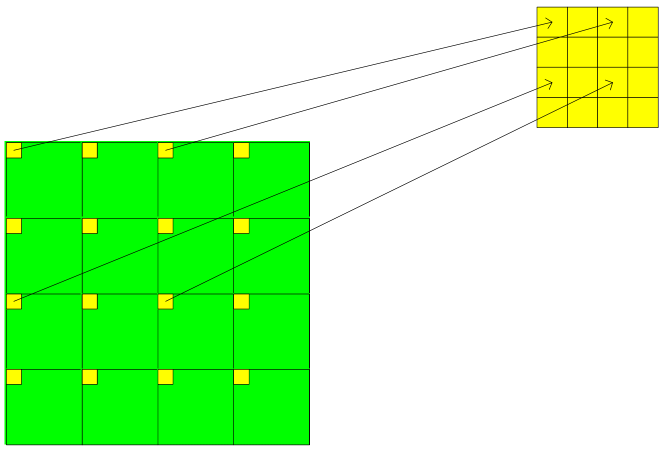


Fig. 3 Separation of DC and AC for Y

The component U is divided into four 4x4 sub-blocks, as Fig. 4 shows. Each of the four 4x4 sub-blocks has a DC coefficient. No matter which prediction mode the macro-block uses, we separate the DC coefficients to form another 2x2 sub-block, as Fig. 5 shows. V and U use the identical scheme of partition, as Fig. 6 shows.

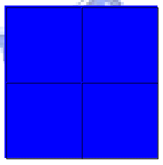


Fig. 4 Division of U component in a macro-block

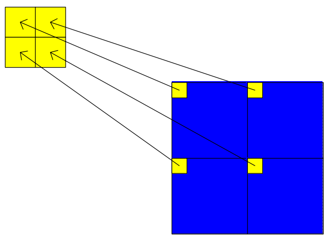


Fig. 5 Separation of DC and AC for U

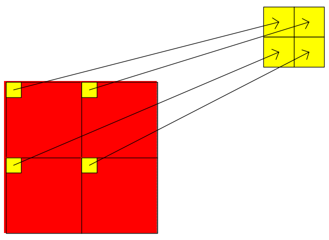


Fig. 6 Separation of DC and AC for V

In conclusion, macro-blocks have only two types in point of CAVLC. First, an Intra_16x16 macro-block is divided as Fig. 7 shows. Sub-blocks “0”-“16” represent Y, “17” and “19”-“22” represent U, and the remaining sub-blocks represent V. The numbers represent the order in which the sub-blocks are encoded by CAVLC [1].

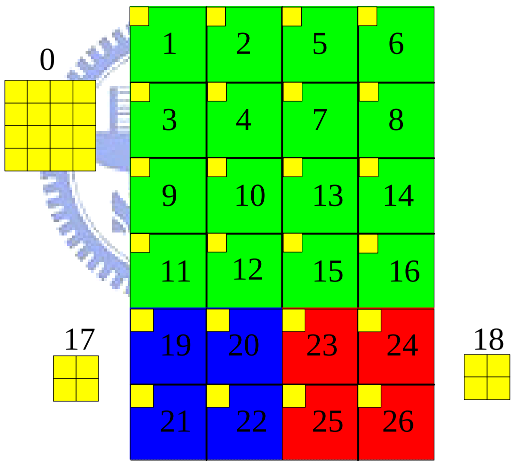


Fig. 7 Division of an Intra_16x16 macro-block

Second, a non-Intra_16x16 macro-block is divided as Fig. 8 shows. Sub-blocks “0”-“15” represent Y, “16” and “18”-“21” represent U, and the remaining sub-blocks represent V. The numbers in Fig. 8 and Fig. 7 have the same meaning.

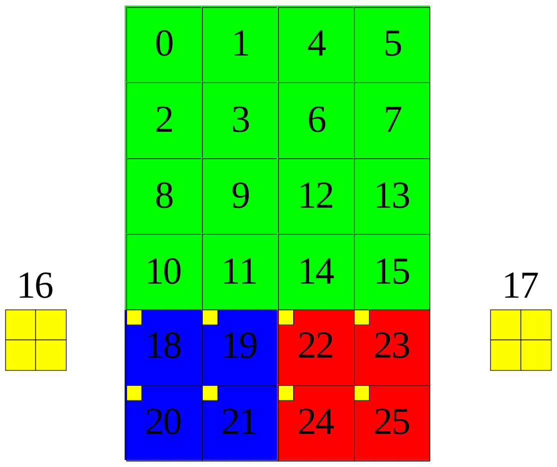


Fig. 8 Division of a non-Intra_16x16 macro-block

2.2.2 The five sub-block types

After a macro-block is divided into several sub-blocks, the sub-blocks are individually encoded by CAVLC in order as mentioned in section 2.2.1. The sub-blocks are divided into five types [9]. For every type the flow of CAVLC is similar but little different. The first type is LUMA_DC referring to sub-block “0” in Fig. 7. LUMA_DC comprises 16 coefficients. The second type is LUMA_AC referring to sub-blocks “1”-“16” in Fig. 7. Each LUMA_AC sub-block comprises 15 coefficients. The third type is CHROMA_DC referring to sub-blocks “17”-“18” in Fig. 7. Each CHROMA_DC sub-block comprises four coefficients. In Fig. 8 sub-blocks “16” and “17” are also CHROMA_DC. The fourth type is CHROMA_AC referring to sub-blocks “19”-“26” in Fig. 7. Each CHROMA_AC comprises 15 coefficients. Sub-blocks “18”-“25” of Fig. 8 are also classified as CHROMA_AC. The final type is LUMA referring to sub-blocks “0”-“15” in Fig. 8. A LUMA comprises 16 coefficients.

In conclusion, an Intra_16x16 macro-block comprises one sub-block of LUMA_DC, sixteen sub-blocks of LUMA_AC, two sub-blocks of CHROMA_DC, and eight sub-blocks of CHROMA_AC. In a non-Intra_16x16 macro-block, there are sixteen sub-blocks of LUMA, two sub-blocks of CHROMA_DC, and eight sub-blocks of CHROMA_AC.

2.2.3 Coded block pattern

The coded block pattern is a syntax element in H.264/AVC [1]. Every macro-block has a coded block pattern which comprises six bits. The lower four bits represent the component Y and the upper two bits represent U and V. We use it to indicate which sub-blocks comprise only zero coefficients.

For an Intra_16x16 macro-block, we present the definition of the coded block pattern below. The 16 LUMA_AC sub-blocks are divided into four groups, as Fig. 9 shows.

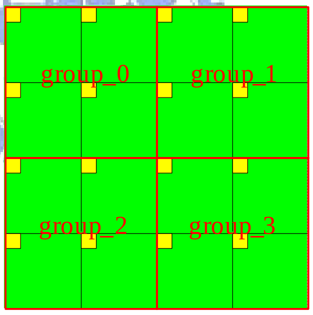


Fig. 9 Four groups of an Intra_16x16 macro-block

Each group comprises four LUMA_AC sub-blocks. The 0th bit (LSB) of the coded block pattern, corresponds to group_0, the 1st bit corresponds to group_1, and so on. If all coefficients of some group are zero, its corresponding bit is zero. Bit “1” means that the corresponding group comprises at least one nonzero coefficient. As for U and V, the upper two bits are defined as TABLE 1.

TABLE 1 Definition of the coded block pattern for chrominance part [1]

CodedBlockPatternChroma	Description
0	All chroma transform coefficient levels are equal to 0.
1	One or more chroma DC transform coefficient levels are non-zero. All chroma AC transform coefficient levels are equal to 0.
2	Zero or more chroma DC transform coefficient levels are non-zero valued. One or more chroma AC transform coefficient levels are non-zero valued.

Only three combinations of the 5th (MSB) and 4th bit are possible, and we describe the combinations in detail as follows:

1. “00” means that every sub-block is all-zero, including two CHROMA_DC sub-blocks and eight CHROMA_AC sub-blocks.
2. “01” means that either CHROMA_DC sub-block comprises at least one nonzero coefficient but each CHROMA_AC sub-block is all-zero.
3. “10” means that at least one nonzero coefficient is among the eight CHROMA_AC sub-blocks.

For a non-Intra_16x16 macro-block, we present the definition of the coded block pattern below. The sixteen LUMA sub-blocks are divided into four groups as Fig. 10 shows.

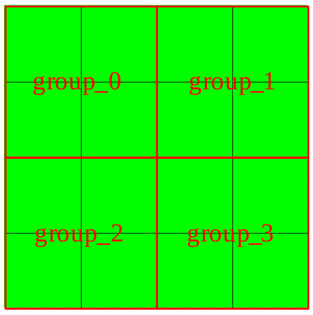


Fig. 10 Four groups of a non_Intra16x16 macro-block

Each group comprises four LUMA sub-blocks. The 0th bit (LSB) corresponds to group_0, the 1st bit corresponds to group_1, and so on. If group_0 consists of only zero coefficients, the 0th bit is zero. The 0th bit is one means that group_0 comprises at least one nonzero coefficient. For the remaining three groups, the meaning of their corresponding bits in the coded block pattern is identical with group_0. The definition for the U and V is identical with an Intra_16x16 macro-block.

We describe how H.264/AVC specification skips sub-blocks according to the coded block pattern below. Because LUMA_DC has no relation to the coded block pattern, even an all-zero LUMA_DC sub-block must be encoded by CAVLC. As for LUMA, CHROMA_DC, and CHROMA_AC, CAVLC can ignore the sub-block as long as the coded block pattern indicates that the sub-block is all-zero. As for LUMA_AC, skip condition is defined as Eq. 1 shows [1] [9].

```
if ($CBP[3:0] == 4'd0$)
  CAVLC ignores all sixteen LUMA_AC sub - blocks
else
  CAVLC encodes all sixteen LUMA_AC sub - blocks
```

Eq. 1

In conclusion, all sub-block types except LUMA_DC have relation to the coded block pattern. We can check the coded block pattern to determine whether CAVLC can skip a sub-block to save cycle counts.

2.2.4 Neighbor sub-blocks

In the flow of CAVLC, every sub-block type except CHROMA_DC needs neighbor information [9]. The information comes from the sub-blocks neighboring to the left and to the top of the current one. We define the neighbor sub-blocks for the four sub-block types, including LUMA, LUMA_AC, LUMA_DC, and CHROMA_AC.

For convenience of explanation, a frame is divided into three parts which respectively represent Y, U, and V. We refer to the three parts as Y-frame, U-frame, and V-frame respectively. A macro-block comprises 16 sub-blocks in Y-frame, as Fig. 11 shows.

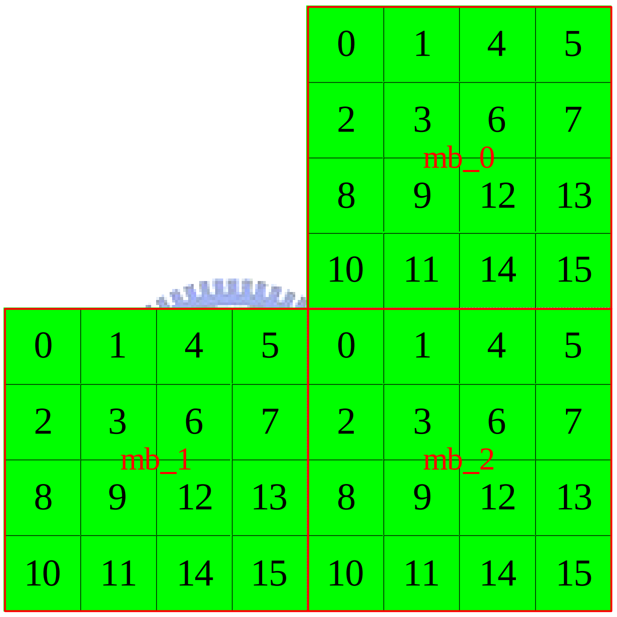


Fig. 11 Three adjacent macro-blocks in Y-frame

For an Intra_16x16 macro-block the sixteen sub-blocks are LUMA_AC. The sixteen sub-blocks of a non-Intra_16x16 macro-block are LUMA. For “12” of mb_2 in Fig. 11, the left neighbor is “9” and the top neighbor is “6”. Some sub-blocks are on the border of a macro-block such as “2” of mb_2, so their neighbor may reside in a different macro-block. For example, the left neighbor of “2” in mb_2 is “7” of mb_1. In a frame, some macro-blocks are Intra_16x16 and some are not, so the neighbor of LUMA may be LUMA_AC. Moreover, some sub-blocks falls on the edge of a frame such that their neighbors may be not available. As for a LUMA_DC sub-block, for example, we assume mb_2 is Intra_16x16; The

LUMA_DC sub-block and “0” have the identical neighbors.

A macro-block comprises four CHROMA_AC sub-blocks in U-frame, as Fig. 12 shows. The neighbors are the sub-blocks neighboring to the left and to the top of the current sub-block. For example, in mb_2 the left neighbor of “3” is “2”, and the top neighbor is “1”. If a sub-block is on the border of a macro-block such as “2” of mb_2, its neighbors may be in another macro-block. As for a macro-block on the border of a frame, some sub-blocks’ neighbors may be not available.

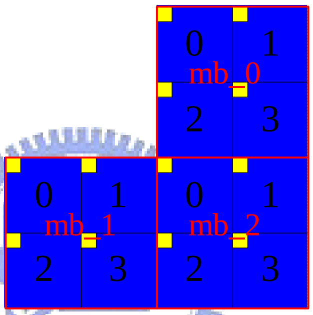


Fig. 12 Three adjacent macro-blocks in U-frame

A macro-block comprises four CHROMA_AC sub-blocks in V-frame, as Fig. 13 shows. V-frame and U-frame have the identical explanation for neighbors.

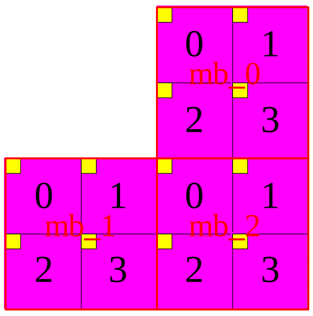


Fig. 13 Three adjacent macro-blocks in V-frame

2.3 Flow of CAVLC in H.264/AVC

A sub-block is a coding unit of CAVLC. We describe the flow of CAVLC for each sub-block type below.

2.3.1 Encoding flow of LUMA_DC

A LUMA_DC sub-block comprises sixteen coefficients as Fig. 14 shows [11]. The coefficients are scanned in zigzag scan order, as Fig. 15 shows [1]. Then the coefficients are mapped to a 1-D array, Array_Coeff, as Fig. 16 shows.

0	3	-1	0
0	-1	1	0
1	0	0	0
0	0	0	0

Fig. 14 A LUMA_DC sub-block

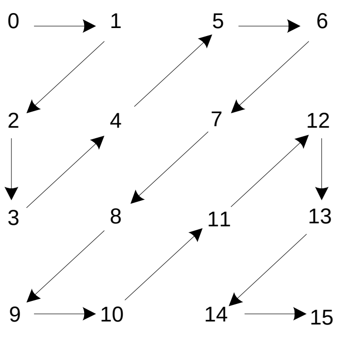


Fig. 15 Zigzag scan order

index	0	1	2	3	4	5	6	7
Array_Coeff	0	3	0	1	-1	-1	0	1
index	8	9	10	11	12	13	14	15
Array_Coeff	0	0	0	0	0	0	0	0

Fig. 16 Structure of Array_Coeff

We analyze Array_Coeff to obtain syntax elements as follows [9]:

1. TotalCoeff
2. TrailingOnes
3. Trailing_one_sign_flag
4. Level
5. Total_zeros
6. Run_before

TotalCoeff is the number of nonzero coefficients in Array_Coeff. TrailingOnes means the number of trailing ones (T1s). To define trailing ones, we scan the coefficients in Array_Coeff from index "15" one by one until the occurrence of a nonzero coefficient whose magnitude is greater than one. If we find a coefficient with magnitude equal to one, the coefficient is so-called trailing one. In H.264/AVC the maximum of TrailingOnes is three such that

Array_Coeff[3] is not classified as a trailing one. H.264/AVC combines TotalCoeff and TrailingOnes into Coeff_token, an H.264/AVC syntax element [1]. Each trailing one has a corresponding trailing_one_sign_flag which comprises only one bit. If the trailing one is negative, its trailing_one_sign_flag is equal to one. A zero trailing_one_sign_flag means that the corresponding trailing one is positive. A level refers to the nonzero coefficient which is not a trailing one. Each nonzero coefficient has a corresponding run_before which is the number of consecutive zeros before the nonzero coefficient in Array_Coeff. For example, the run_before of Array_Coeff[7] is one and that of Array_Coeff[4] is zero. Finally, total_zeros is the sum of all run_before.

After all syntax elements are obtained, we encode them in numerical order as follows [9]:

1. Coeff_token
2. Trailing_one_sign_flags
3. Levels
4. Total_zeros
5. Run_befores

Therefore, the structure of H.264/AVC bit-stream is as Fig. 17 shows.

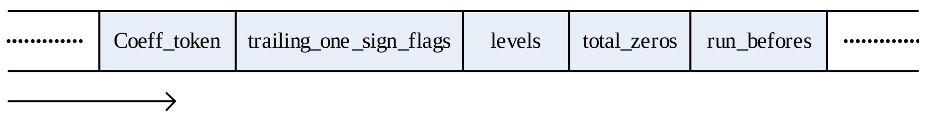


Fig. 17 Structure of H.264/AVC bit-stream

First, we encode Coeff_token by TABLE 2.

TABLE 2 coeff_token mapping to TotalCoeff and TrailingOnes [1]

TrailingOnes (coeff_token)	**TotalCoeff (coeff_token)**	**0 <= nC < 2**	**2 <= nC < 4**	**4 <= nC < 8**	**8 <= nC**	**nC == -1**
0	0	1	11	1111	0000 11	01
0	1	0001 01	0010 11	0011 11	0000 00	0001 11
1	1	01	10	1110	0000 01	1
0	2	0000 0111	0001 11	0010 11	0001 00	0001 00
1	2	0001 00	0011 1	0111 1	0001 01	0001 10
2	2	001	011	1101	0001 10	001
0	3	0000 0011 1	0000 111	0010 00	0010 00	0000 11
1	3	0000 0110	0010 10	0110 0	0010 01	0000 011
2	3	0000 101	0010 01	0111 0	0010 10	0000 010
3	3	0001 1	0101	1100	0010 11	0001 01
0	4	0000 0001 11	0000 0111	0001 111	0011 00	0000 10
1	4	0000 0011 0	0001 10	0101 0	0011 01	0000 0011
2	4	0000 0101	0001 01	0101 1	0011 10	0000 0010
3	4	0000 11	0100	1011	0011 11	0000 000
0	5	0000 0000 111	0000 0100	0001 011	0100 00	-
1	5	0000 0001 10	0000 110	0100 0	0100 01	-
2	5	0000 0010 1	0000 101	0100 1	0100 10	-
3	5	0000 100	0011 0	1010	0100 11	-
0	6	0000 0000 0111 1	0000 0011 1	0001 001	0101 00	-
1	6	0000 0000 110	0000 0110	0011 10	0101 01	-
2	6	0000 0001 01	0000 0101	0011 01	0101 10	-
3	6	0000 0100	0010 00	1001	0101 11	-
0	7	0000 0000 0101 1	0000 0001 111	0001 000	0110 00	-
1	7	0000 0000 0111 0	0000 0011 0	0010 10	0110 01	-
2	7	0000 0000 101	0000 0010 1	0010 01	0110 10	-
3	7	0000 0010 0	0001 00	1000	0110 11	-
0	8	0000 0000 0100 0	0000 0001 011	0000 1111	0111 00	-
1	8	0000 0000 0101 0	0000 0001 110	0001 110	0111 01	-
2	8	0000 0000 0110 1	0000 0001 101	0001 101	0111 10	-
3	8	0000 0001 00	0000 100	0110 1	0111 11	-
0	9	0000 0000 0011 11	0000 0000 1111	0000 1011	1000 00	-

1	9	0000 0000 0011 10	0000 0001 010	0000 1110	1000 01	-
2	9	0000 0000 0100 1	0000 0001 001	0001 010	1000 10	-
3	9	0000 0000 100	0000 0010 0	0011 00	1000 11	-
0	10	0000 0000 0010 11	0000 0000 1011	0000 0111 1	1001 00	-
1	10	0000 0000 0010 10	0000 0000 1110	0000 1010	1001 01	-
2	10	0000 0000 0011 01	0000 0000 1101	0000 1101	1001 10	-
3	10	0000 0000 0110 0	0000 0001 100	0001 100	1001 11	-
0	11	0000 0000 0001 111	0000 0000 1000	0000 0101 1	1010 00	-
1	11	0000 0000 0001 110	0000 0000 1010	0000 0111 0	1010 01	-
2	11	0000 0000 0010 01	0000 0000 1001	0000 1001	1010 10	-
3	11	0000 0000 0011 00	0000 0001 000	0000 1100	1010 11	-
0	12	0000 0000 0001 011	0000 0000 0111 1	0000 0100 0	1011 00	-
1	12	0000 0000 0001 010	0000 0000 0111 0	0000 0101 0	1011 01	-
2	12	0000 0000 0001 101	0000 0000 0110 1	0000 0110 1	1011 10	-
3	12	0000 0000 0010 00	0000 0000 1100	0000 1000	1011 11	-
0	13	0000 0000 0000 1111	0000 0000 0101 1	0000 0011 01	1100 00	-
1	13	0000 0000 0000 001	0000 0000 0101 0	0000 0011 1	1100 01	-
2	13	0000 0000 0001 001	0000 0000 0100 1	0000 0100 1	1100 10	-
3	13	0000 0000 0001 100	0000 0000 0110 0	0000 0110 0	1100 11	-
0	14	0000 0000 0000 1011	0000 0000 0011 1	0000 0010 01	1101 00	-
1	14	0000 0000 0000 1110	0000 0000 0010 11	0000 0011 00	1101 01	-
2	14	0000 0000 0000 1101	0000 0000 0011 0	0000 0010 11	1101 10	-
3	14	0000 0000 0001 000	0000 0000 0100 0	0000 0010 10	1101 11	-
0	15	0000 0000 0000 0111	0000 0000 0010 01	0000 0001 01	1110 00	-

1	15	0000 0000 0000 1010	0000 0000 0010 00	0000 0010 00	1110 01	-
2	15	0000 0000 0000 1001	0000 0000 0010 10	0000 0001 11	1110 10	-
3	15	0000 0000 0000 1100	0000 0000 0000 1	0000 0001 10	1110 11	-
0	16	0000 0000 0000 0100	0000 0000 0001 11	0000 0000 01	1111 00	-
1	16	0000 0000 0000 0110	0000 0000 0001 10	0000 0001 00	1111 01	-
2	16	0000 0000 0000 0101	0000 0000 0001 01	0000 0000 11	1111 10	-
3	16	0000 0000 0000 1000	0000 0000 0001 00	0000 0000 10	1111 11	-

The symbol nC represents the average of TotalCoeff of the neighbor sub-blocks. Eq. 2 shows the algorithm of calculating nC [9]. "left_available" represents the availability of the left neighbor and nL is the TotalCoeff of the left neighbor. "top_available" represents the availability of the top neighbor and nU is the TotalCoeff of the top neighbor.

```
if (left_available && top_available)
  $nC = (nU + nL + 1)/2;$
else if(left_available && !top_avalable)
  $nC = nL;$
else if(!left_available && top_available)
  $nC = nU;$
else
  $nC = 0;$
```

Eq. 2

Second, the number of bits for trailing_one_sing_flags is equal to TrailingOnes. We encode T1s from high frequency to low frequency such that "011" is the codeword of T1s for Array_Coeff, where "0" represents Array_Coeff[7] (see Fig. 16).

Third, we encode each level from high frequency to low frequency. There are seven VLC tables to encode a level and the choice of tables depends on already encoded levels. Eq. 3

shows the algorithm of selecting the VLC table for the first encoded level [9]. Each table has a serial number: vlcnum. Because we have seven VLC tables, vlcnum ranges from zero to six.

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

Eq. 3

The VLC table for the second encoded level depends on the absolute value of the first encoded level. If the absolute value is greater than three, we add one to vlcnum. Otherwise, the second and the first use the identical table. The third encoded level and the following use the identical algorithm to choose the VLC table, as Eq. 4 shows [9].

```
int incVlc[] = {0,3,6,12,24,48,32768};

if (abs(level) > incVlc[vlcnum])
  vlcnum++;
```

Eq. 4

The choice of VLC table for the third encoded level depends on the absolute value of the second encoded level, the fourth depends on the third, and so on. Each table corresponds to a threshold denoted by incVlc in Eq. 4. If the absolute value of the second encoded level is greater than the threshold corresponding to the table used by the second encoded level, we add one to vlcnum such that the third encoded level use an updated table. Otherwise, we apply the identical VLC table on the third encoded level.

Fourth, total_zeros is encoded by TABLE 3.

TABLE 3 total_zeros tables for LUMA, LUMA_DC, LUMA_AC, CHROMA_AC [1]

total_zeros	**TotalCoeff(coeff_token)**						
	1	2	3	4	5	6	7
0	1	111	0101	0001 1	0101	0000 01	0000 01
1	011	110	111	111	0100	0000 1	0000 1
2	010	101	110	0101	0011	111	101
3	0011	100	101	0100	111	110	100
4	0010	011	0100	110	110	101	011
5	0001 1	0101	0011	101	101	100	11
6	0001 0	0100	100	100	100	011	010
7	0000 11	0011	011	0011	011	010	0001
8	0000 10	0010	0010	011	0010	0001	001
9	0000 011	0001 1	0001 1	0010	0000 1	001	0000 00
10	0000 010	0001 0	0001 0	0001 0	0001	0000 00	
11	0000 0011	0000 11	0000 01	0000 1	0000 0		
12	0000 0010	0000 10	0000 1	0000 0			
13	0000 0001 1	0000 01	0000 00				
14	0000 0001 0	0000 00					
15	0000 0000 1						

total_zeros	**TotalCoeff(coeff_token)**							
	8	9	10	11	12	13	14	15
0	0000 01	0000 01	0000 1	0000	0000	000	00	0
1	0001	0000 00	0000 0	0001	0001	001	01	1
2	0000 1	0001	001	001	01	1	1	
3	011	11	11	010	1	01		
4	11	10	10	1	001			
5	10	001	01	011				
6	010	01	0001					
7	001	0000 1						
8	0000 00							

Finally, we encode each run_before except the nonzero coefficient of the lowest frequency such as Array_Coeff[1] (see Fig. 16). Run_befores are encoded from high frequency to low frequency such that the first encoded run_before belongs to Array_Coeff[7]. The codeword is looked up in TABLE 4, where zerosLeft means how many run_befores are not encoded including the current run_before. For example, the zerosLeft of Array_Coeff[7] is three and that of Array_Coeff[5] is two. When zerosLeft is equal to zero, the encoding process for the current and the following run_befores can be terminated in advance [11].

TABLE 4 Tables for run_before [1]

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	[illegible]	-	-	-	-	-	00001
9	-	-	-	-	-	-	000001
10	-	-	-	-	-	-	0000001
11	-	-	-	-	-	-	00000001
12	-	-	-	-	-	-	000000001
13	-	-	-	-	-	-	0000000001
14	-	-	-	-	-	-	00000000001

2.3.2 Encoding flow of LUMA, LUMA_AC, and CHROMA_AC

The CAVLC flow of LUMA is the same as LUMA_DC. LUMA_AC comprises fifteen coefficients such that the operation of scanning the coefficients into Array_Coeff is different from LUMA_DC, as Fig. 18 shows.

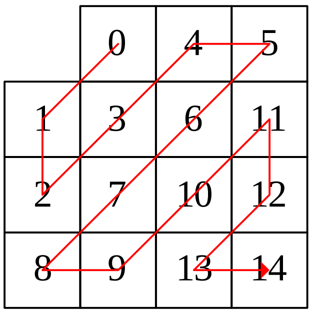


Fig. 18 Scanning for a LUMA_AC sub-block

Then the following processing of Array_Coeff is similar to LUMA_DC and the differences are described below. If TotalCoeff is equal to 15, we can skip encoding of total_zeros and run_befores [9]. Finally, the CAVLC flow of CHROMA_AC is the same as LUMA_AC.

2.3.3 Encoding flow of CHROMA_DC

CHROMA_DC comprises four coefficients and Fig. 19 represents the scanning order for the coefficients into Array_Coeff. The following processing of Array_Coeff is similar to LUMA_DC and the differences are described below. First, nC is always set as -1 when we apply TABLE 2 on Coeff_token [1]. Second, we use TABLE 5 for total_zeros instead of TABLE 3. Moreover, if TotalCoeff is equal to four, we can skip encoding of total_zeros and run_befores [9].

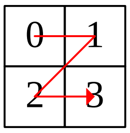


Fig. 19 Scanning for a CHROMA_DC sub-block

TABLE 5 total_zeros tables for CHROMA_DC [1]

total_zeros	TotalCoeff(coeff_token)		
	1	2	3
0	1	1	1
1	01	01	0
2	001	00	
3	000		

Chapter 3. Architecture Design of the Proposed CAVLC Encoder

3.1 Motivation: statistics of zero coefficients

For a sub-block, we can check the coded block pattern to decide whether the CAVLC flow can be skipped. However, there are many zero coefficients that cannot be skipped by the coded block pattern. For example, Fig. 20 represents a group of four LUMA sub-blocks, where only subblock_0 comprises one nonzero coefficient and another three sub-blocks are all-zero. As mentioned in section 2.2.3, all sub-blocks must go through the CAVLC flow such that many cycle counts are wasted on zeros.

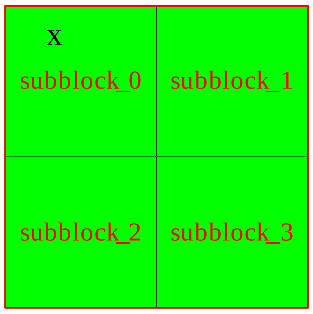


Fig. 20 A group of four LUMA sub-blocks

In the section, we represent statistics to show that abundant zeros cannot be skipped by the coded block pattern. To obtain the statistics, we ran several simulations using the

H.264/AVC reference software [9]. The test sequences include the following (motion from low to high): akiyo, coastguard, foreman, mobile_calendar, and Stefan. TABLE 6 shows the simulation setting. Two types of statistics are obtained and described below.

TABLE 6 Simulation setting

Video Size	CIF
Frame Number	300
Intra Period	10
Number of Reference Frames	1
Use FME	ON
RD Optimization	OFF

3.1.1 Statistics of nonzero coefficients in NS4B

NS4B (Not Skipped 4x4 Block) denotes a sub-block, excluding CHROMA_DC, which cannot be skipped by the coded block pattern. Fig. 21 represents the average number of nonzero coefficients in a NS4B. CHROMA_DC is excluded because maximal TotalCoeff of CHROMA_DC is only four such that the average is reduced unfairly. Note that Mobile_calendar with QP equal to 12 has the highest bit-rate. Even for the highest bit-rate, a NS4B comprises less than seven nonzero coefficients; in other words, up to 60% of coefficients are zero.

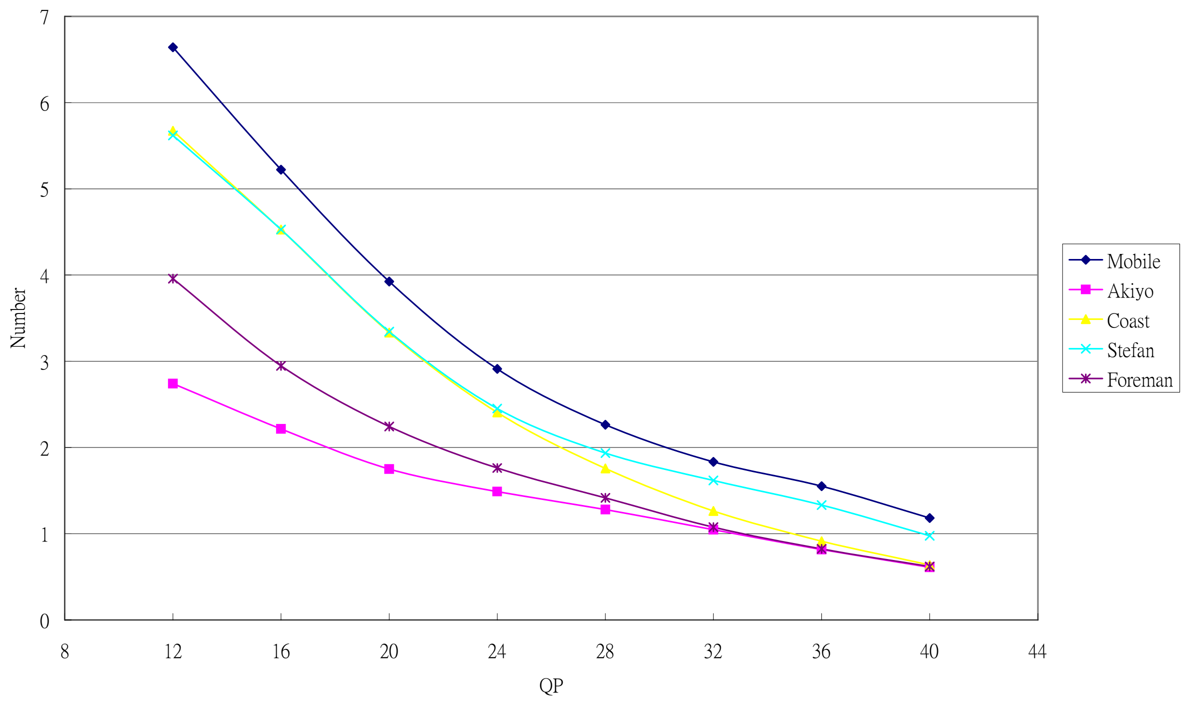


Fig. 21 Average number of nonzero coefficients per NS4B sub-block

3.1.2 Statistics of NSZB

NSZB (Not Skipped Zero Block) denotes an all-zero sub-block, which cannot be skipped by the coded block pattern. For example, subblock_1 of Fig. 20 can be classified as a NSZB. Fig. 22 shows how many percent of all-zero sub-blocks are NSZB. Note that a higher bit-rate results in a higher percentage. This is because a lower bit-rate induces more zero CBP bits such that more all-zero sub-blocks are covered by the CBP. Note that the percentage is significant at middle QP and low QP. Therefore, Fig. 22 proves that many all-zero sub-blocks cannot be skipped by the coded block pattern in most cases.

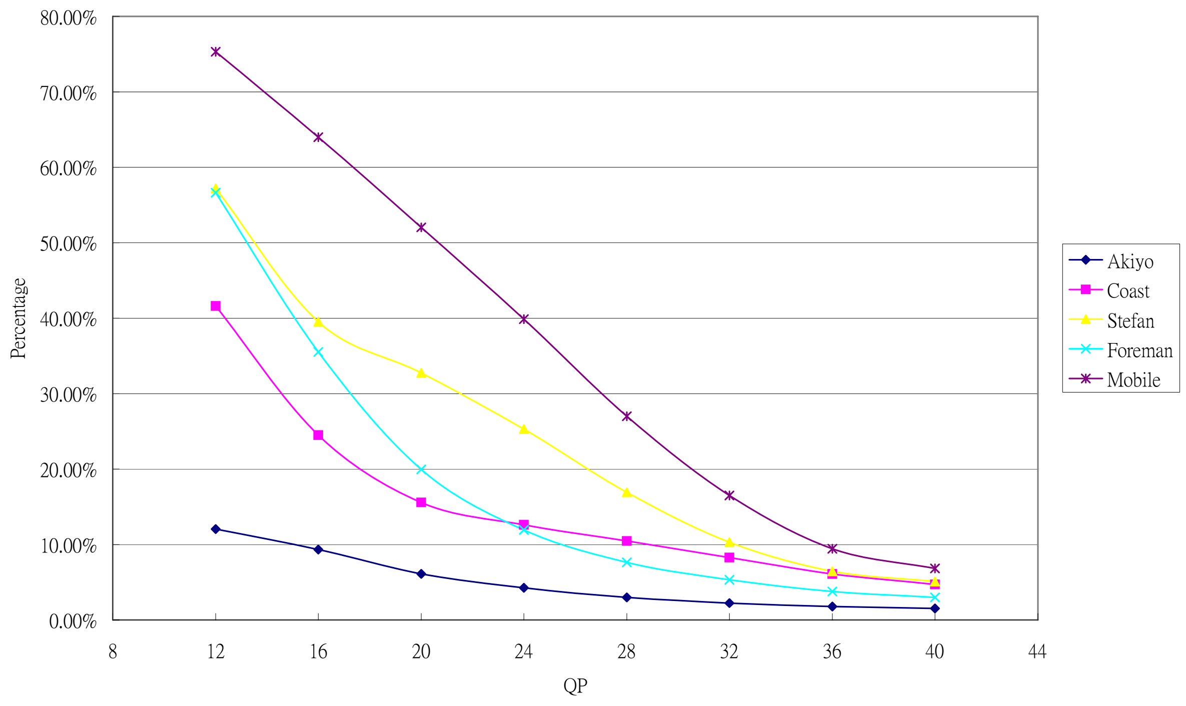


Fig. 22 Percentage of NSZB in all-zero sub-blocks

3.1.3 Summary

In this section, we show the statistics to prove that abundant zero coefficients cannot be skipped by the coded block pattern. Thus CBP Look-Ahead is an inefficient method to skip zeros. To solve the problem, we propose two methods such that more zeros can be ignored. First, we use Nonzero Index Table to skip zeros in a non-all-zero sub-block. Second, we use Zero-block Codeword Table to directly encode a NSZB without going through the whole course of CAVLC flow. The two methods will be examined in section 3.4

3.2 System consideration of CAVLC in H.264/AVC encoder

Fig. 23 shows the partial system architecture of an H.264/AVC encoder. The encoder adopts a macro-block pipeline schedule. Global Control Unit administers the progress of macro-blocks in the pipeline, where each stage comprises and processes one macro-block. Global Control Unit generates the following two things which are propagated through the pipeline registers:

1. Syntax elements which are so-called header information such as mb_type [1].
2. Information about the macro-block at PR0 (Pipeline Register 0) such as coordinates.

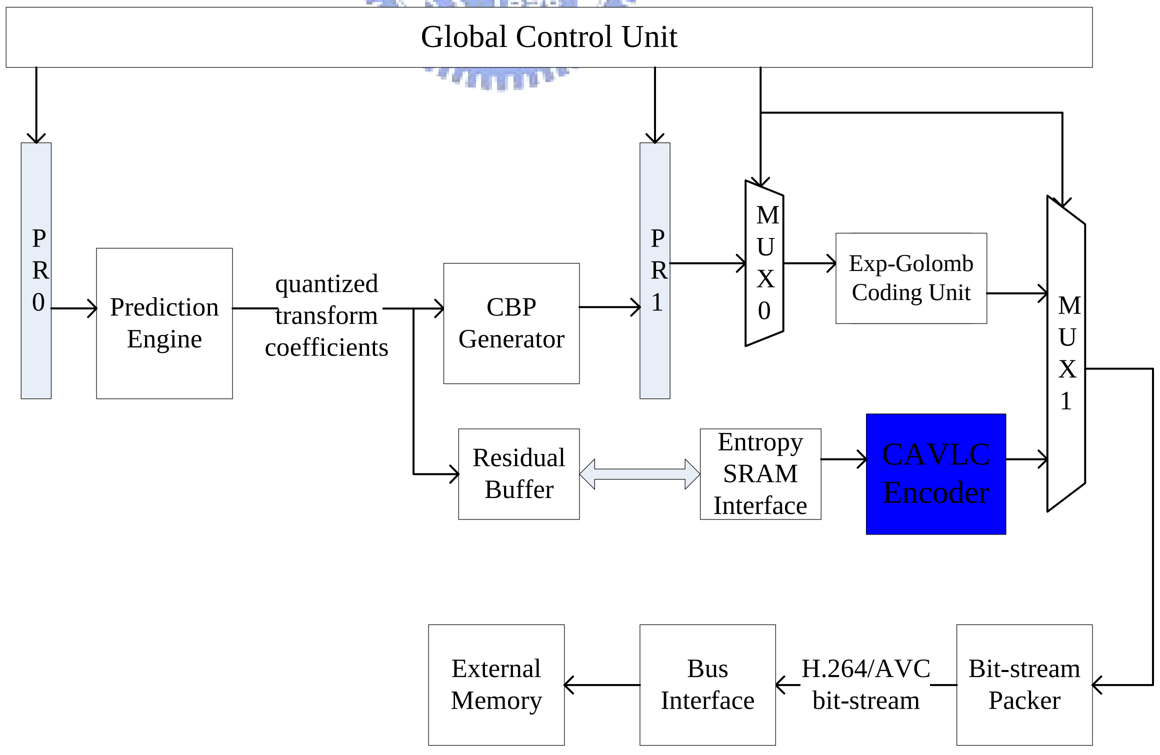


Fig. 23 Partial system architecture of an H.264/AVC encoder

Prediction Engine uses data in PR0 to execute Inter or Intra prediction algorithms such that the best prediction signal can be worked out. Then Prediction Engine subtracts the prediction signal from the macro-block to obtain residual data. Finally, Prediction Engine applies transform and quantization on the residual data to obtain quantized transform coefficients. CBP Generator examines whether some coefficients are zero to determine the coded block pattern which will be stored into PR1. Moreover, the coefficients are stored into Residual Buffer. After all stages are finished, Global Control Unit grants the progress of the pipeline such that the data in PR0 goes into PR1.

Header information in PR1 is encoded in predefined order which Global Control Unit maintain by MUX 0 (Multiplexer 0) and MUX 1 (Multiplexer 1). The codeword of header information is generated by Exp-Golomb Coding Unit and is concatenated by Bit-stream Packer such that H.264/AVC bit-stream is formed. The bit-stream is written into External Memory via Bus Interface.

After encoding of header information is finished, Entropy SRAM Interface fetches coefficients of one sub-block from Residual Buffer to CAVLC Encoder. The codeword of each syntax element such as Coeff_token is sent to Bit-stream Packer via MUX 1. When encoding of the sub-block is finished, CAVLC Encoder will request Entropy SRAM Interface to fetch the next sub-block.

3.2.1 Residual Buffer

Residual Buffer comprises five parts. The first part is Luma SRAM, which is implemented as an SRAM. Luma SRAM stores the Y component of one macro-block and Fig. 24 shows the organization of Luma SRAM.

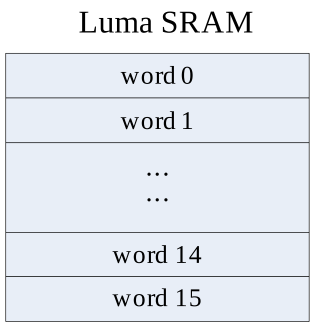


Fig. 24 Organization of Luma SRAM

As mentioned in section 2.1, a macro-block comprises sixteen Y sub-blocks and one word stores one sub-block in Luma SRAM. Fig. 25 represents the mapping relation between the sub-blocks and the words.

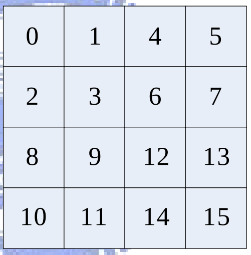


Fig. 25 Y sub-blocks of a macro-block

Fig. 26 shows the organization of a word in Luma SRAM, where we use 14 bits to represent a coefficient. Fig. 27 shows the mapping relation between the numerical labels and the coefficients of a LUMA sub-block.

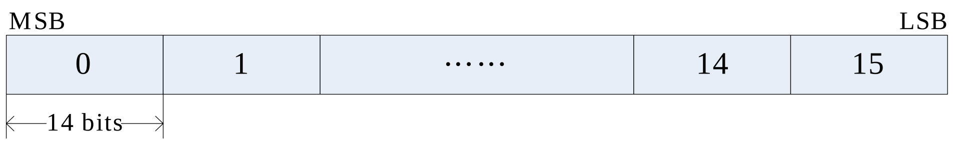


Fig. 26 Organization of a word in Luma SRAM

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Fig. 27 The coefficients of a LUMA sub-block

Fig. 28 shows the mapping relation between the numerical labels of Fig. 26 and the coefficients of a LUMA_AC sub-block. As for LUM_AC, the position “0” in Fig. 26 is useless because the DC is separated. In brief, Luma SRAM comprises 16x16x14 bits.

X	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Fig. 28 The coefficients of a LUMA_AC sub-block

The second part is Chroma SRAM, which is implemented as an SRAM and is used to store CHROMA_AC sub-blocks. Fig. 29 shows the organization of Chroma SRAM.

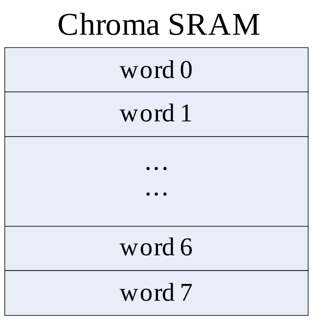


Fig. 29 Organization of Chroma SRAM

A macro-block comprises eight CHROMA_AC sub-blocks and one word stores one sub-block in Chroma SRAM, so there are totally eight words in Chroma SRAM. Fig. 30 shows the mapping relation between the words of Chroma SRAM and the CHROMA_AC sub-blocks of a macro-block.

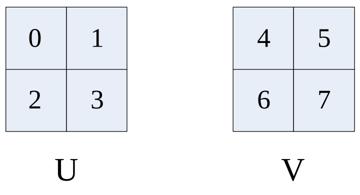


Fig. 30 U and V sub-blocks of a macro-block

Fig. 31 shows the organization of one word in Chroma SRAM, where we use 12 bits to represent one coefficient. Fig. 32 shows the mapping relation between the numerical labels of Fig. 31 and the coefficients of a CHROMA_AC sub-block. Because the DC is separated, the position "0" in Fig. 31 is always useless; however, it exists for the regularity of hardware architecture. In brief, Chroma SRAM comprises 8x16x12 bits.

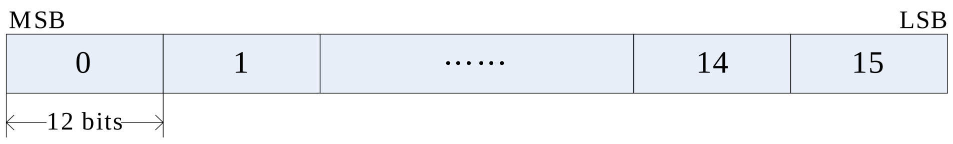


Fig. 31 Organization of one word in Chroma SRAM

X	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Fig. 32 Coefficients of a CHROMA_AC sub-block

The third part is LDC (Luma DC) Register, which is implemented as a register and is used to store one LUMA_DC sub-block. Fig. 33 shows the organization of LDC Register, where we use 14 bits to represent one coefficient. Fig. 34 shows the mapping relation between the numerical labels of Fig. 33 and the coefficients of a LUMA_DC sub-block. In brief, LDC Register comprises 1x16x14 bits because there are at most one LUMA_DC sub-block in one macro-block.

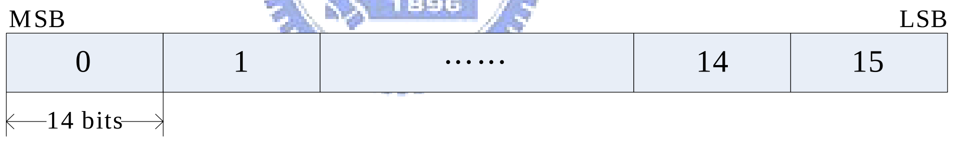


Fig. 33 Organization of LDC Register

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Fig. 34 Coefficients of a LUMA_DC sub-block

The fourth part is CDCU (Chroma DC U) Register, which is implemented as a register and is used to store one CHROMA_DC sub-block of U component. Fig. 35 shows the organization of CDCU Register, where we use 14 bits to store one coefficient. Fig. 36 shows the mapping relation between the numerical labels of Fig. 35 and the coefficients of a CHROMA_DC sub-block. In brief, CDCU Register comprises 1x4x14 bits because U component consists of only one CHROMA_DC sub-block.

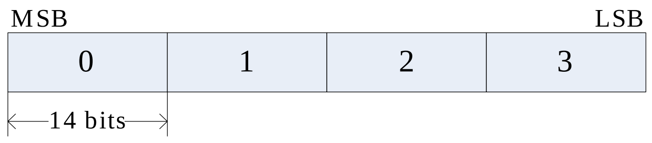


Fig. 35 Organization of CDCU Register

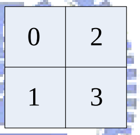


Fig. 36 Coefficients of a CHROMA_DC sub-block

The final part is CDCV (Chroma DC V) Register, which is implemented as a register and is used to store one CHROMA_DC sub-block of V component. The organization of CDCV Register is the same as CDCU Register. The permutation of the coefficients in CDCV Resister is the same as Fig. 36 shows. In brief, CDCV Register comprises 1x4x14 bits because V component consists of only one CHROMA_DC sub-block.

Last but not least, as for Luma SRAM one word comprises 224 bits such that the bus seems a little wider. This is because we can write one sub-block per cycle to reduce cycle counts on macro-block level and to improve the throughput of the macro-block pipeline schedule. On the contrary, if a narrower bus is adopted, we must spend more cycle counts on writing one macro-block into Residual Buffer than a wider bus.

3.2.2 CBP Generator

Fig. 37 shows the architecture of CBP Generator. Prediction Engine writes one sub-block into Residual Buffer per cycle. Global Control Unit provides Global Counter for Prediction Engine to determine which sub-block is written now. In CBP Generator, we use Global Counter to generate Sub-block Index which represents one of the twenty-six sub-block as Fig. 38 shows.

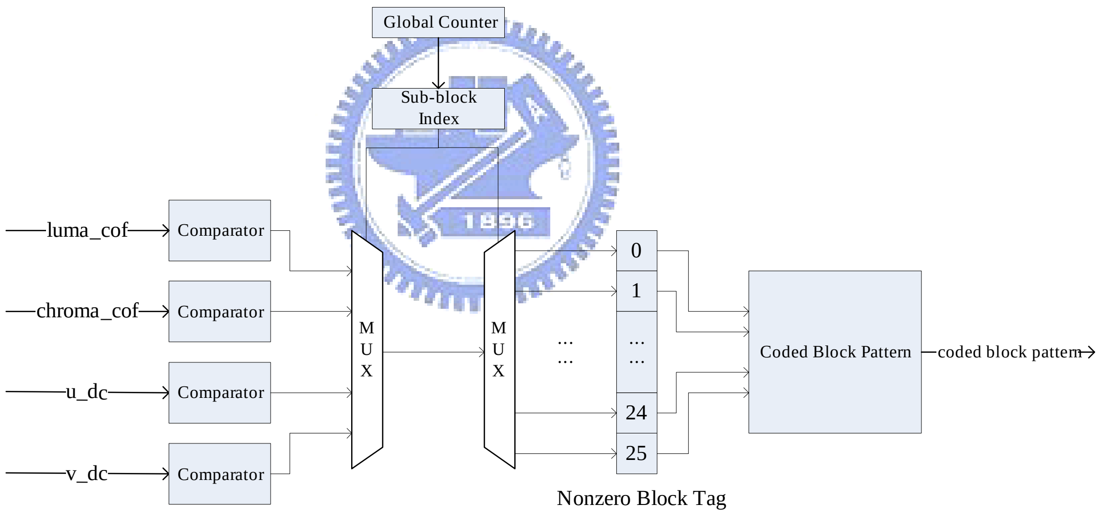


Fig. 37 Architecture of CBP Generator

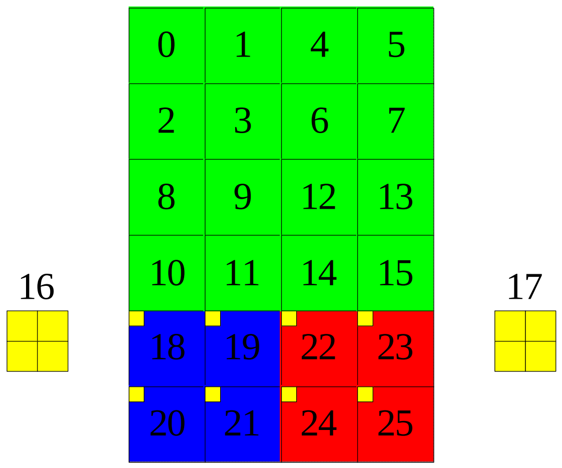


Fig. 38 Sub-block Index of each sub-blocks in a macro-block

Prediction Engine uses luma_cof to send coefficients to Luma SRAM, uses chroma_cof to send coefficients to Chroma SRAM, uses u_dc to send coefficients to CDCU Register, and uses v_dc to send coefficients to CDCV Register. For example, when Sub-block Index is equal to three, luma_cof now comprises coefficients of sub-block “3” in Fig. 38. We use comparator to examine whether it is an all-zero sub-block. If the sub-block is all-zero, we set zero to the 3rd bit in Nonzero Block Tag, which is a 26-bit register. After the 26 sub-blocks are examined through, Nonzero Block Tag is settled and Coded Block Pattern of Fig. 37, a combinational circuit, can generate the coded block pattern for the macro-block.

3.2.3 Entropy SRAM Interface

Fig. 39 shows the architecture of Entropy SRAM Interface. Global Control Unit defines a serial number, syntax_idx_cur, for each syntax element. When syntax_idx_cur is equal to some value, it means encoding of coefficients is started and esi_enable is activated to enable Entropy SRAM Interface.

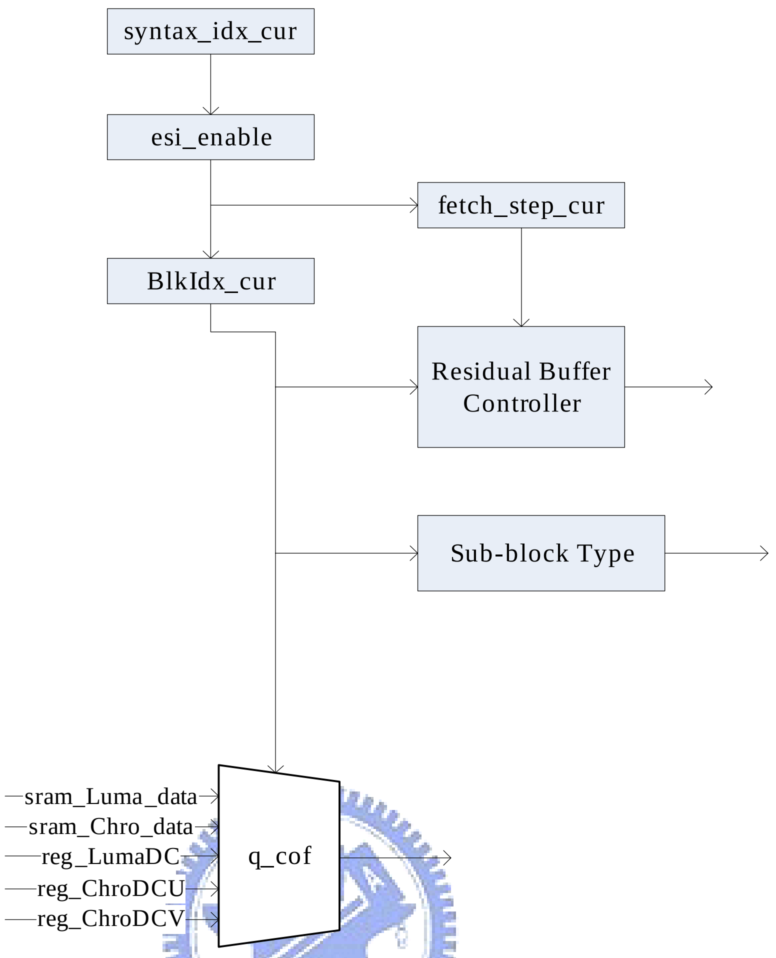


Fig. 39 Architecture of Entropy SRAM Interface

"fetch_step_cur" is initialized as zero and BlkIdx_cur is initialized as zero or one depending on the macro-block type. If the macro-block is an Intra_16x16, BlkIdx_cur is initialized as zero; otherwise, BlkIdx_cur is initialized as one. BlkIdx_cur represents which sub-block Entropy SRAM Interface fetches right now. Fig. 40 shows the mapping relation between BlkIdx_cur and the sub-blocks of an Intra_16x16 macro-block. Fig. 41 shows the mapping relation between BlkIdx_cur and the sub-blocks of a non-Intra_16x16 macro-block.

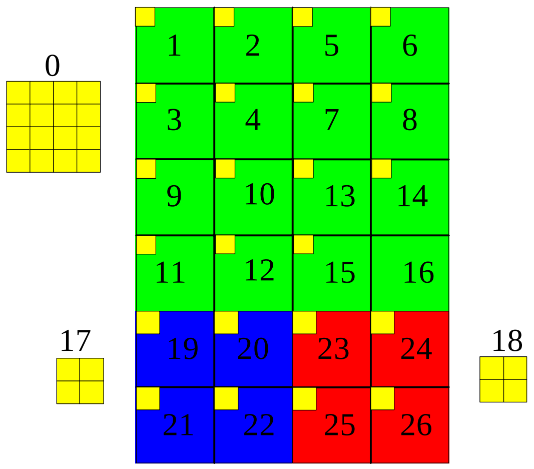


Fig. 40 BlkIdx_cur for each sub-block of an Intra_16x16 macro-block

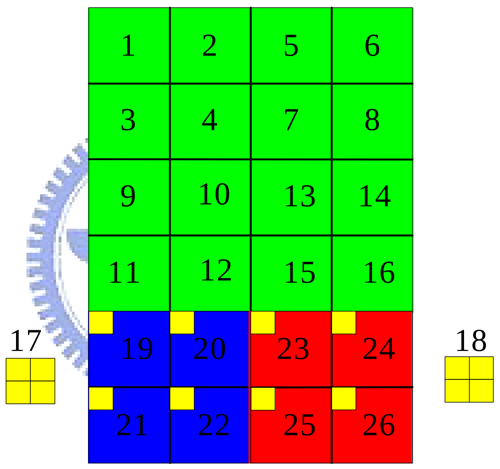


Fig. 41 BlkIdx_cur for each sub-block of a non-Intra_16x16 macro-block

“fetch_step_cur” represents the steps during the interaction between Entropy SRAM Interface and Residual Buffer. Fig. 42 shows the timing schedule of fetch_step_cur.

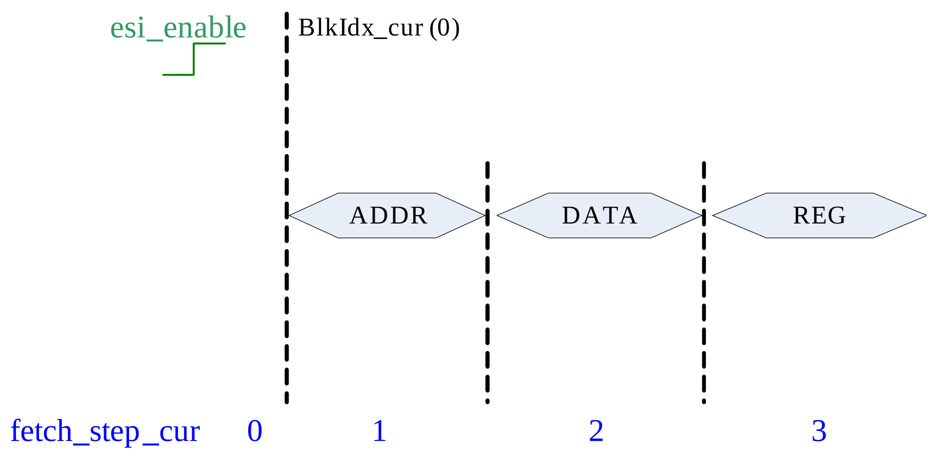


Fig. 42 Timing schedule of fetch_step_cur

When fetch_step_cur is equal to one, Residual Buffer Controller sends read address to Residual Buffer according to BlkIdx_cur. Then, when fetch_step_cur is equal to two, the read data is ready at the output of Residual Buffer. The output of Residual Buffer refers to sram_Luma_data, sram_Chroma_data, reg_LumaDC, reg_ChroDCU, and reg_ChroDCV in Fig. 39. "q_cof" selects one of the five output signals according to BlkIdx_cur. Finally, when fetch_step_cur is equal to three, the coefficients of the sub-block are stored into CAVLC Encoder. Moreover, Sub-block Type calculates the type of sub-block according to the current BlkIdx_cur.

3.2.4 Exp-Golomb Coding Unit

Fig. 43 shows the architecture of Exp-Golomb Coding Unit. "data" is the syntax elements from PR1 (see Fig. 23). As for Exp-Golomb Coding Unit, the syntax elements are divided into three categories as follows: UE (Unsigned Exp-Golomb), SE (Signed Exp-Golomb), and CBP [1]. Global Control Unit sends "mode" to indicate which category according to the current syntax element. We must calculate codeNum [1] of the current syntax element to generate the codeword by Exp-Golomb Code Table.

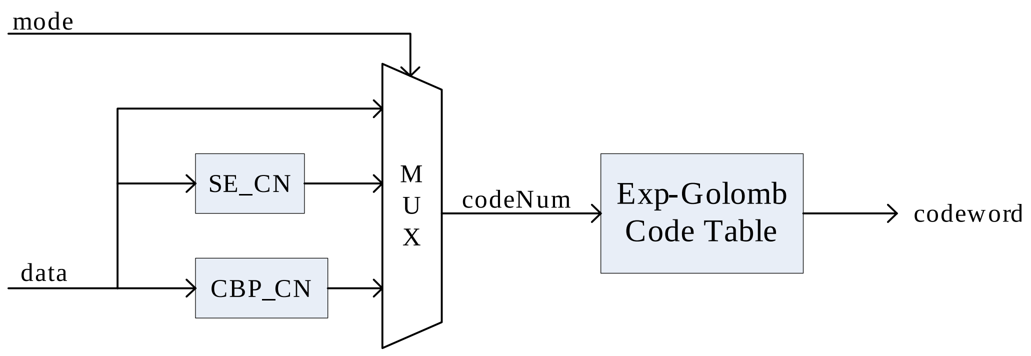


Fig. 43 Architecture of Exp-Golomb Coding Unit

As for UE, the syntax element is equal to codeNum such that "data" can be directly sent to Exp-Golomb Code Table via MUX. As for SE and CBP, we use SE_CN and CBP_CN to calculate the codeNumb respectively.

TABLE 7 shows the structure of Exp-Golomb Code Table.

TABLE 7 Structure of Exp-Golomb Code Table [1]

Codeword	**Range of codeNum**
1	0
0 1 x_0	1-2
0 0 1 x_1 x_0	3-6
0 0 0 1 x_2 x_1 x_0	7-14
0 0 0 0 1 x_3 x_2 x_1 x_0	15-30
0 0 0 0 0 1 x_4 x_3 x_2 x_1 x_0	31-62
…	…

Fig. 44 shows the structure of codeword in TABLE 7. "M" represents the number of prefix zero bits and INFO is the value of suffix bits [11]. "codeNum" is expressed as Eq. 5 shows.

$$\underbrace{0 \quad 0 \quad 0}_{M} \quad 1 \quad \underbrace{x_2 \quad x_1 \quad x_0}_{INFO}$$

Fig. 44 Structure of Exp-Golomb codeword

$$codeNum = 2^M - 1 + INFO$$

Eq. 5

TABLE 8 shows Exp-Golomb Code Table in explicit form. SE_CN and CBP_CN are implementations of TABLE 9 and TABLE 10 respectively. As for TABLE 10, note that the mapping is different between Intra and Inter.

TABLE 8 Exp-Golomb Code Table in explicit form [1]

Bit string	**codeNum**
1	0
0 1 0	1
0 1 1	2
0 0 1 0 0	3
0 0 1 0 1	4
0 0 1 1 0	5
0 0 1 1 1	6
0 0 0 1 0 0 0	7
0 0 0 1 0 0 1	8
0 0 0 1 0 1 0	9
…	…

TABLE 9 Assignment of syntax element to codeNum for signed Exp-Golomb [1]

codeNum	**syntax element value**
0	0
1	1
2	–1
3	2
4	–2

5	3
6	–3
k	$(-1)^{k+1}$ Ceil(k÷2)

TABLE 10 Assignment of codeNum to values of CBP [1]

codeNum	**CBP**	
	Intra	**Inter**
0	47	0
1	31	16
2	15	1
3	0	2
4	23	4
5	27	8
6	29	32
7	30	3
8	7	5
9	11	10
10	13	12
11	14	15
12	39	47
13	43	7
14	45	11
15	46	13
16	16	14
17	3	6
18	5	9
19	10	31
20	12	35
21	19	37
22	21	42
23	26	44
24	28	33
25	35	34
26	37	36

27	42	40
28	44	39
29	1	43
30	2	45
31	4	46
32	8	17
33	17	18
34	18	20
35	20	24
36	24	19
37	6	21
38	9	26
39	22	28
40	25	23
41	32	27
42	33	29
43	34	30
44	36	22
45	40	25
46	38	38
47	41	41

3.2.5 Bit-stream Packer

Fig. 45 shows the input and output ports of Bit-stream Packer. Fig. 46 shows the architecture of Bit-stream Packer.

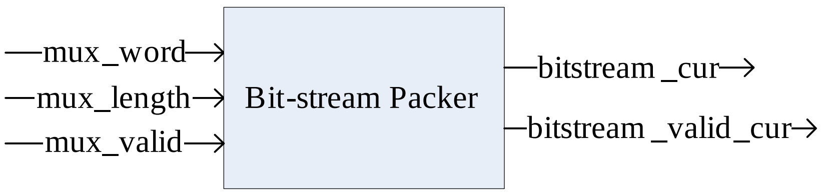


Fig. 45 Input and Output ports of Bit-stream Packer

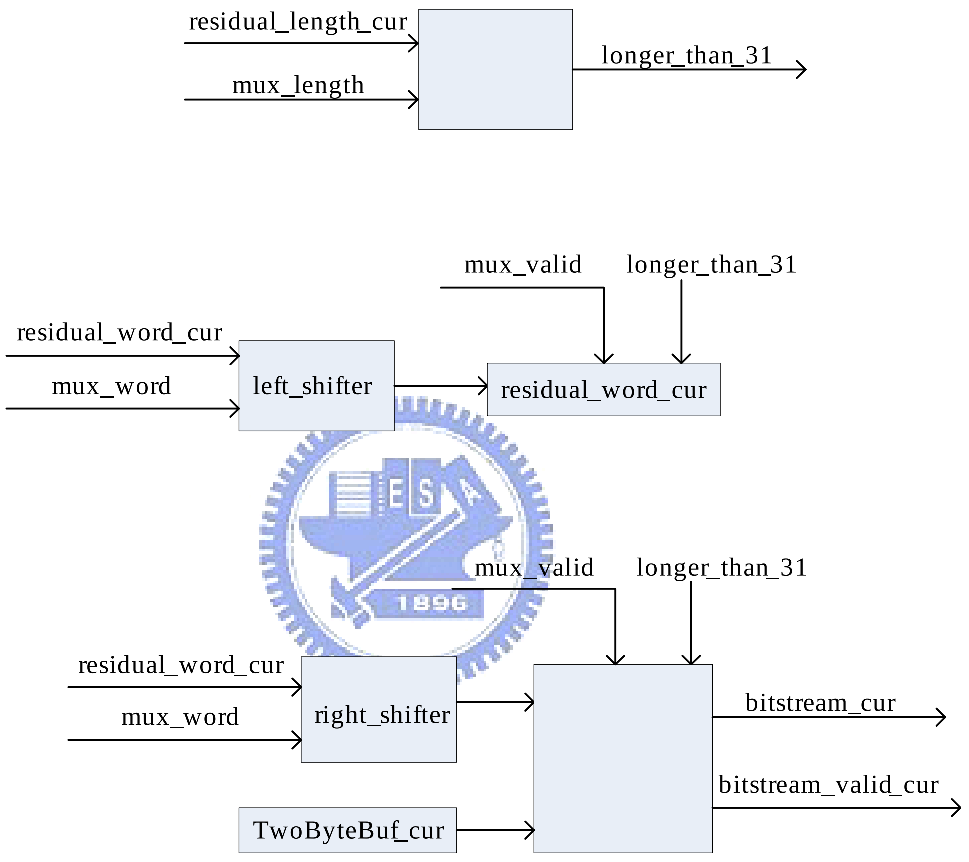


Fig. 46 Architecture of Bit-stream Packer

“mux_word”, “mux_length”, and “mux_valid” are output signals of MUX 1 (See Fig. 23). “mux_word” carries the codeword of some syntax element and mux_length represents the length of the codeword. Fig. 47 shows the structure of mux_word, where the codeword is “00101” and mux_length is equal to five. Note that mux_word is left-alignment.

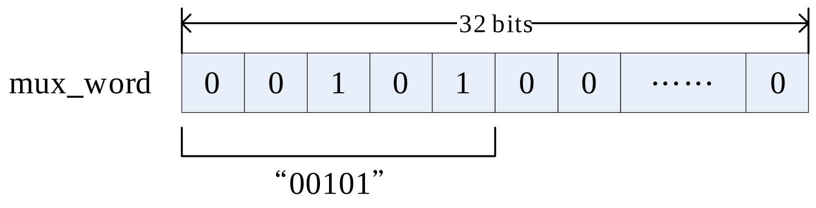


Fig. 47 Structure of mux_word

When mux_valid is high, it means that mux_word is valid and that we must concatenate mux_word into residual_word_cur, a 32-bit register. The length of the codeword in residual_word_cur is represented by residual_length_cur. Fig. 48 shows the structure of residual_word_cur, where the codeword is "00101" and residual_length_cur is equal to five. Note that residual_word_cur is right-alignment.

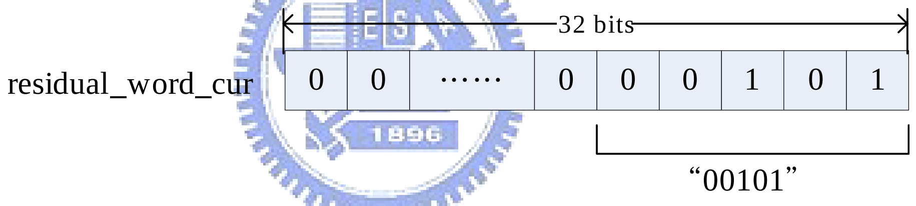


Fig. 48 Structure of residual_word_cur

When mux_valid is high, if the sum of mux_length and residual_length_cur is smaller than 32 (longer_than_31 is low), residual_word_cur is updated by left_shifter as Fig. 49 shows.

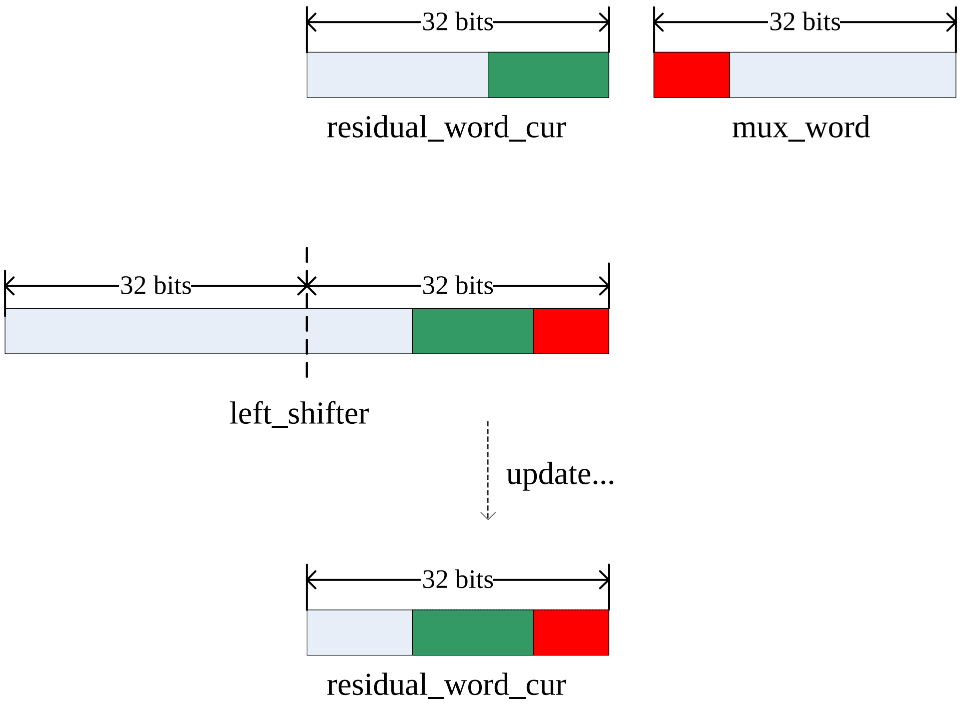


Fig. 49 Updating of residual_word_cur by left_shifter when longer_than_31 is low

"left_shifter" is a right-alignment signal of 64 bits and consists of the concatenation of residual_word_cur and mux_word. On the other hand, if the sum of residual_length_cur and mux_length is larger than 31 (longer_than_31 is high), bitstream_cur is updated as shown in Fig. 50 and bitstream_valid_cur is high such that Bus Interface (see Fig. 23) can fetch the updated bitstream_cur.

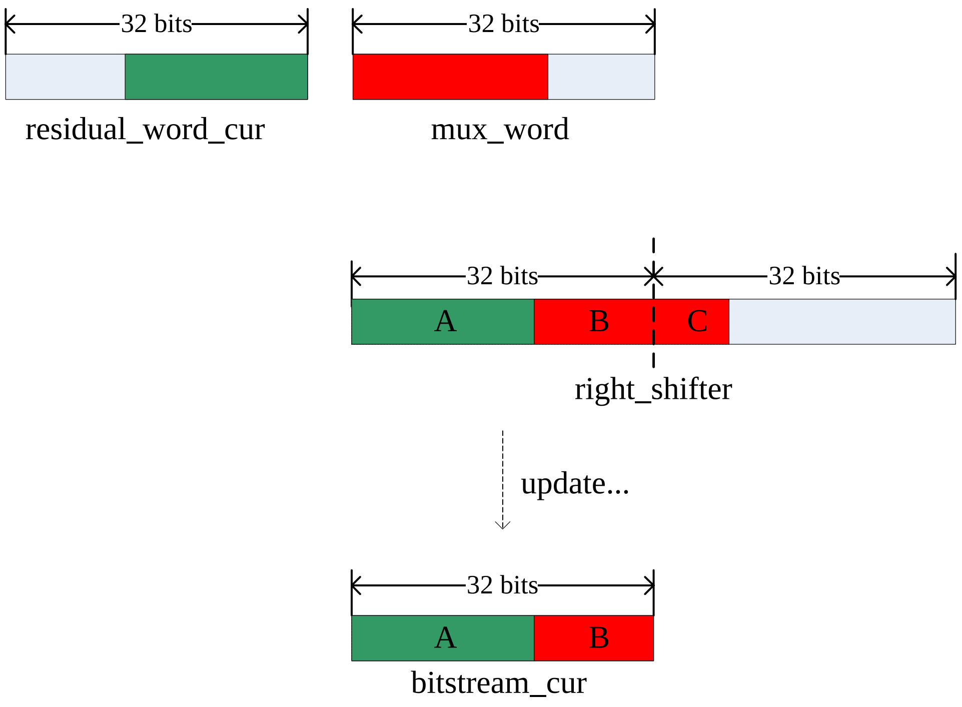


Fig. 50 Updating of bitstream_cur by right_shifter

“bitstream_cur” is a register of 32 bits; in other words, we send 32 bits of bit-stream to Bus Interface in one cycle. “right_shifter” is a left-alignment signal of 64 bits and consists of the concatenation of residual_word_cur and mux_word. The remaining part, “C”, is stored into residual_word_cur.

Moreover, TwoByteBuf_cur is a register of 16 bits and comprises the last two bytes of H.264/AVC bit-stream stored in External Memory (see Fig. 23). We use TwoByteBuf_cur and right_shiter to determine whether emulation prevention byte [1], a byte equal to 0x03, is necessary to be inserted into bitstream_cur. The main purpose of emulation prevention bytes is to ensure that start code prefix [1], a sequence of three bytes equal to 0x000001, occurs only at the beginning of the H.264/AVC bit-stream. When the following successive three bytes are found in the raw byte sequence, emulation prevention byte is inserted:

1. 0x000000 → 0x00000300

2. 0x000001 → 0x00000301

3. 0x000002 → 0x00000302

4. 0x000003 → 0x00000303

The usage of TwoByteBuf_cur is illustrated in Fig. 51. “right_shifter[63:32]” is divided into four bytes and the arrowhead is the insert position of emulation prevention byte. There are only seven cases according to the analysis [12]. For example, case 1 is illustrated in Fig. 52, where the number is hexadecimal. After insertion, bitstream_cur becomes 0x03000003 and 0x0002 is stored into residual_word_cur.

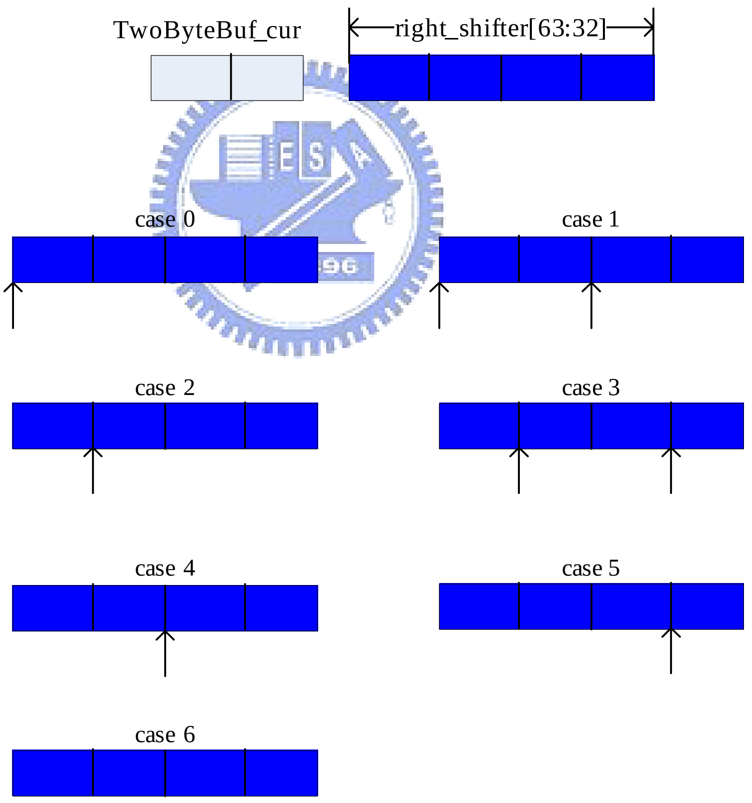


Fig. 51 Usage of TwoByteBuf_cur

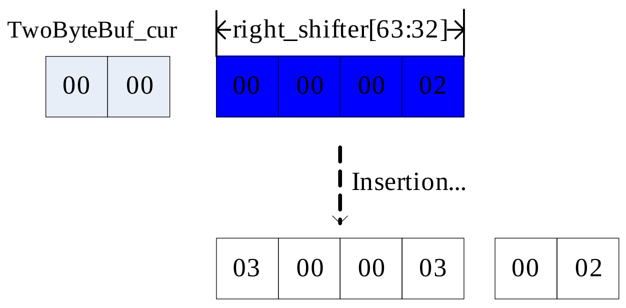


Fig. 52 Example for case 1

3.3 Encoding flow of proposed CAVLC encoder

Fig. 53 represents encoding flow of the proposed CAVLC encoder. Entropy SRAM Interface provides the proposed CAVLC encoder with the coded block pattern and coefficients of a sub-block.

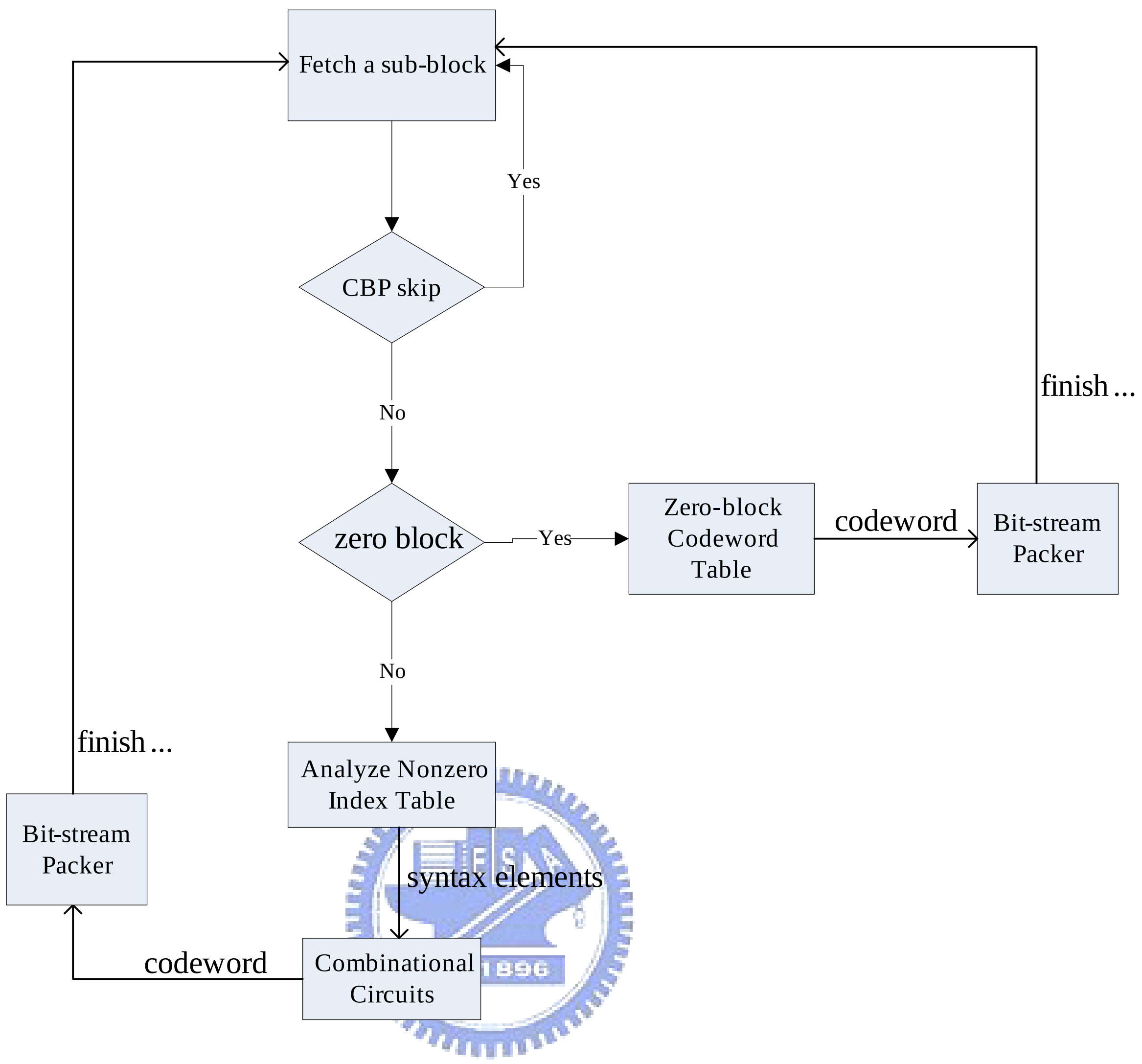


Fig. 53 Encoding flow of the proposed CAVLC encoder

The proposed CAVLC encoder uses the coded block pattern to determine whether the sub-block can be skipped. If the sub-block can be skipped, there are no further steps and we fetch the next sub-block from Entropy SRAM Interface. Otherwise, if the sub-block cannot be skipped, the coefficients are loaded into Input Buffer, as Fig. 54 shows. Nonzero Index Table concurrently records which coefficients are nonzero by setting bit “1”. Both Input Buffer and Nonzero Index Table are registers constructed in the proposed CAVLC encoder.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Buffer	0	0	0	0	0	0	0	0	1	0	-1	-1	1	0	3	0
Nonzero Index Table	0	0	0	0	0	0	0	0	1	0	1	1	1	0	1	0

Fig. 54 Input Buffer and Nonzero Index Table

If all bits are zero in Nonzero Index Table, as Fig. 55 shows, it means that the sub-block is NSZB. Zero-block Codeword Table generates the codeword of the NSZB and Bit-stream Packer concatenates the codeword in parallel. Then the encoding flow is terminated earlier such that we save cycle counts for collecting syntax elements of the NSZB.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Buffer	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Nonzero Index Table	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 55 Nonzero Index Table for an all-zero sub-block

Otherwise, if Nonzero Index Table consists of nonzero bits (see Fig. 54), it means that the sub-block consists of nonzero coefficients. This thesis refers to a sub-block comprising nonzero coefficients as NAZ (Not All Zero). We analyze Nonzero Index Table to obtain syntax elements whose codeword are generated by combinational circuits and are concatenated by Bit-stream Packer. After all syntax elements of the sub-block are encoded into H.264/AVC bit-stream, the encoding of the NAZ sub-block is finished and we can fetch the next sub-block.

Nonzero Index Table makes it possible to directly encode significant coefficients such that we can save cycle counts on zero coefficients in NAZ. We can encode one nonzero coefficient every cycle by updating Nonzero Index Table. Fig. 56 and Fig. 57 show the updating. At some cycle, as shown in Fig. 56, we aim at the coefficient at index “7” and the codeword of the coefficient is concurrently generated by combinational circuits. At the next cycle (see Fig. 57), Nonzero Index Table is updated and we aim at the coefficient at index “4” such that the zeros between index “4” and “7” are ignored. Hence, Nonzero Index Table saves cycle counts on zero coefficients in NAZ.

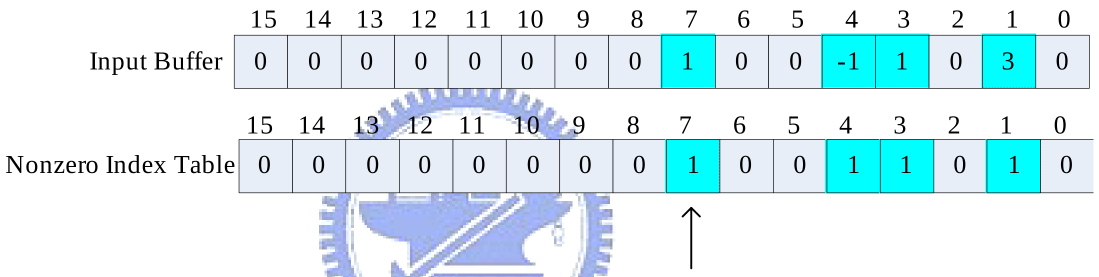


Fig. 56 Coefficient at index 7 is being encoded

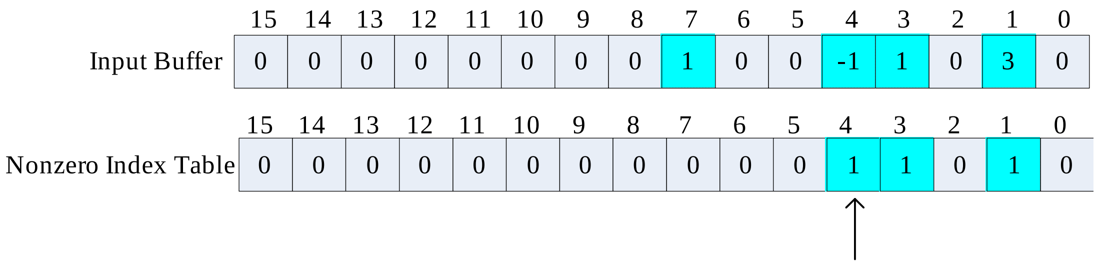


Fig. 57 Coefficient at index 4 is being encoded

3.4 Architecture of proposed CAVLC encoder

According to section 3.3, sub-blocks can be divided into three categories in terms of encoding flow:

1. NAZ: as mentioned in section 3.3.
2. NSZB: as mentioned in section 3.1.2.
3. CS (CBP Skip): a sub-block which can be skipped by the coded block pattern.

In this section, we first describe the architecture of the proposed CAVLC encoder. Then, in the following three sections, we cycle-wise explain the hardware operation for NAZ, NSZB, and CS to manifest the advantages of our design.

Fig. 58 shows the architecture of the proposed CAVLC encoder. "cavlc_data_ready" fetches sixteen coefficients in one cycle. If the sub-block comprises less than sixteen coefficients like CHROMA_DC, Entropy SRAM Interface automatically appends zeros. According to the sub-block type and the coded block pattern, we determine whether the sub-block is CS.

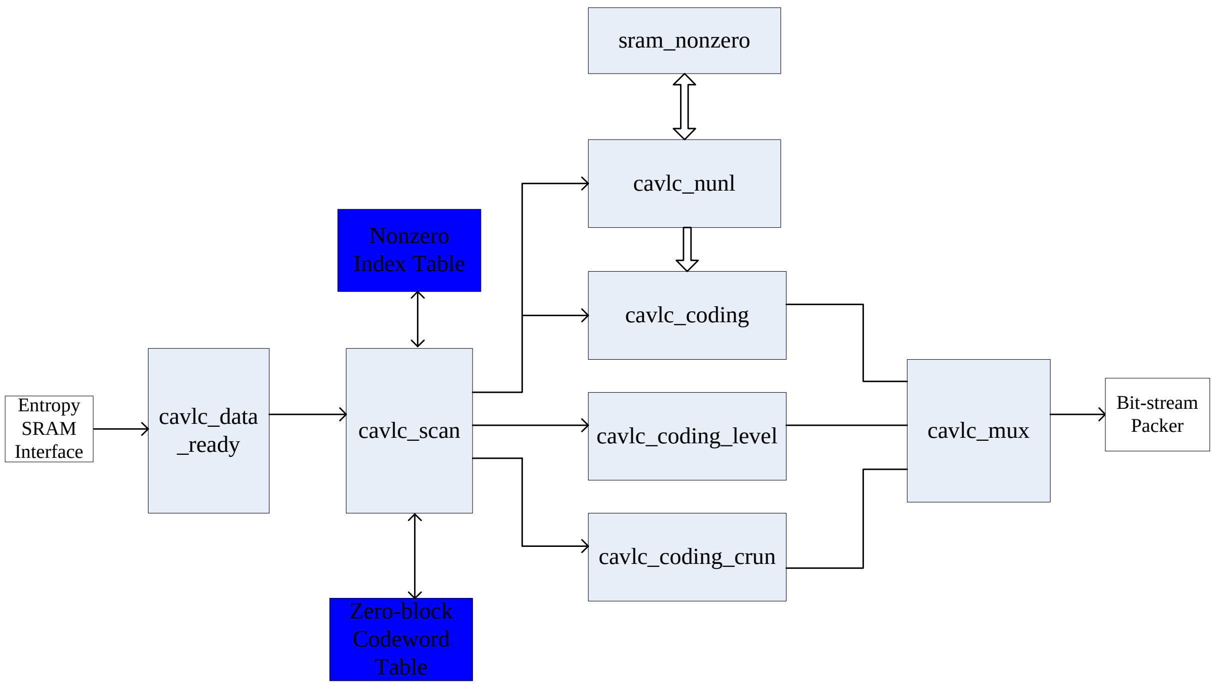


Fig. 58 Architecture of the proposed CAVLC encoder

As for a sub-block of NSZB or NAZ, the sixteen coefficients including the appending zeros are stored into Input Buffer, which comprises sixteen 16-bit registers, as Fig. 59 shows. Input Buffer is constructed in cavlc_data_ready. The coefficients are put into Input Buffer in zigzag scan order. We consider Input Buffer as a 1-D array and index “0” represents the DC. The greater the index is, the higher the frequency is. Nonzero Index Table comprises sixteen 1-bit registers and indicates which coefficients are nonzero. When the coefficients are stored, we concurrently check which coefficients are zeros. If a coefficient is nonzero, we set one to the corresponding bit in Nonzero Index Table. The index of Nonzero Index Table is the same as Input Buffer.

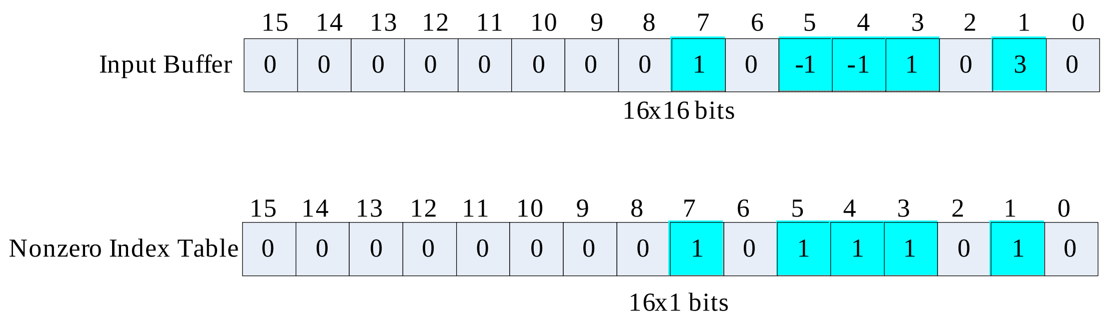


Fig. 59 Structure of Input Buffer and Nonzero Index Table

"sram_nonzero" is an SRAM which records the TotalCoeff of already encoded sub-blocks. "cavlc_nunl" reads neighbor TotalCoeff from sram_nonzero and calculates nC. Then cavlc_nunl writes the current TotalCoeff into sram_nonzero.

"cavlc_scan" extracts syntax elements from Nonzero Index Table. We explain the operation of cavlc_scan cycle-wise below.

At the first cycle, cavlc_scan uses a simple adder to calculate the number of bit "1" in Nonzero Index Table, which is TotalCoeff. Then, cavlc_scan uses start index and stop index to scan Nonzero Index Table, as Fig. 60 shows. Start index is a register and we initialize it as fifteen. This is because the coefficients of higher frequency are encoded earlier [9]. Stop index indicates the leading bit "1" in Nonzero Index Table from index 15 to 0. Stop index is the output of FindLeadingOne, a combinational circuit constructed in cavlc_scan. After stop index and TotalCoeff are known, total_zeros can be defined as Eq. 6.

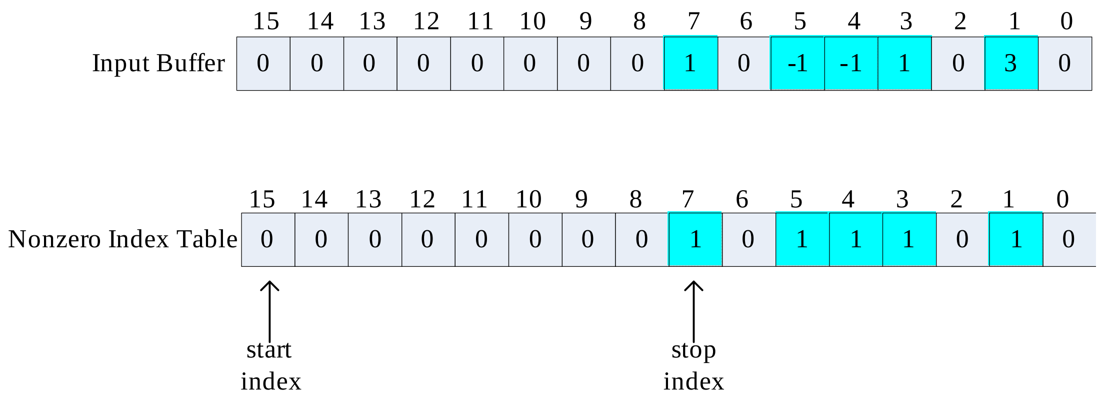


Fig. 60 Start index and stop index with Nonzero Index Table

$$\text{stop index} + 1 - \text{TotalCoeff}$$

Eq. 6

On the other hand, if FindLeadingOne cannot find any one bit in Nonzero Index Table, the sub-block is NSZB. We encode the NSZB by Zero-block Codeword Table (see Fig. 61) and the encoding of the NSZB is finished in advance. In fact, Zero-block Codeword Table is the first row of TABLE 2 because “all-zero” means both TotalCoeff and TrailingOnes are equal to zero.

0<=nC<2	2<=nC<4	4<=nC<8	8<=nC	nC==-1
1	11	1111	000011	01

Fig. 61 Zero-block Codeword Table

At the second cycle, Nonzero Index Table is updated as shown in Fig. 62, where the bit at the position of stop index is set zero and start index moves to the new zero bit. Now we can calculate the run_before of the coefficient pointed by stop index at previous cycle, which is expressed as Eq. 7. For example, the run_before is one for the coefficient at index seven.

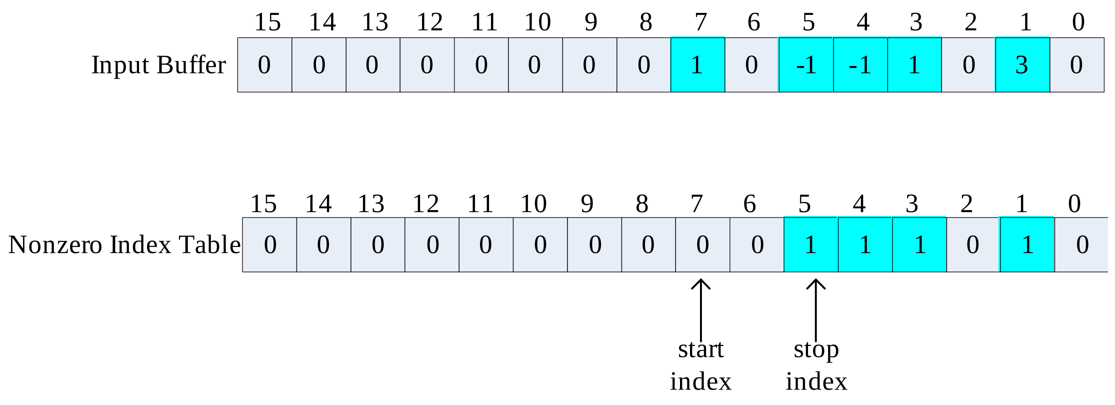


Fig. 62 Updating for Nonzero Index Table

$$\text{start index} - \text{stop index} - 1 \qquad \text{Eq. 7}$$

At the following cycles, we do the same things as the second cycle.

Every cycle we examine whether the coefficient pointed by stop index is a trailing one. If the coefficient is a trailing one, we add one to TrailingOnes and the trailing_one_sign_flag is concurrently determined. As soon as TrailingOnes is settled, cavlc_mux permits the codeword of Coeff_token and trailing_one_sign_flags to be sent to Bit-stream Packer. The codeword of Coeff_token is the concurrent output of cavlc_coding in which TABLE 2 is implemented. Because trailing_one_sign_flags are codeword in itself, no additional circuit is necessary.

After trailing_one_sign_flags are encoded into the bit-stream, the coefficient pointed by stop index is sure to be a level. In such cases, cavlc_mux certainly grants the codeword of the level to be sent to Bit-stream Packer. The codeword of the level is the concurrent output of cavlc_coding_level which is adapted from JM encoder reference software [9]. In JM encoder, levels are encoded by a simple calculation instead of one-to-one mapping tables like TABLE 2.

After each level is encoded into the bit-stream, cavlc_mux grants the codeword of total_zeros to be sent to Bit-stream Packer. The codeword of total_zeros is the output of cavlc_coding in which TABLE 3 and TABLE 5 are implemented.

Although run_befores are the final syntax element in the bit-stream, we can obtain a run_before accompanied with a level or a trailing one in one cycle. The codeword of the run_before is concurrently generated by cavlc_coding_crun in which TABLE 4 is implemented. “cavlc_coding_crun” concatenates each run_before codeword into a 32-bit register before sent to Bit-stream Packer. According to TABLE 4, we can reason that the total length of all run_before codeword for a sub-block is always smaller than 32. After total_zeros is encoded into the bit-stream, we spend only one cycle to send the concatenated codeword to Bit-stream Packer.

3.4.1 Hardware operation for encoding a NAZ

We assume the sub-block is LUMA as Fig. 63 shows. The hardware operation of the proposed CAVLC encoder is described cycle-wise below:

3	-3	0	0
-1	1	2	-1
0	1	-1	-1
0	0	0	0

Fig. 63 Coefficients of a LUMA sub-block

1. 0th cycle: Entropy SRAM Interface updates BlkIdx_cur for the sub-block.

2. 1st cycle: According to BlkIdx_cur, Entropy SRAM Interface does the following two things. First, Entropy SRAM Interface sends read address to Residual Buffer (see Fig. 23). Second, Entropy SRAM Interface sends block_type and block_idx to CAVLC encoder. “block_type” refers to the sub-block type as mentioned in section 2.2.2. “block_idx” has the same definition as the numerical labels of Fig. 87. “block_idx” is used to determine read address of sram_nonzero to fetch neighbor TotalCoeff. “block_idx” is stored into blk_idx_cur and block_type is stored into blk_typ_cur, where both blk_idx_cur and blk_typ_cur are registers constructed in cavlc_data_ready.

   Moreover, CAVLC encoder obtains mb_x and mb_y from PR1. “mb_x” and mb_y are coordinates of the current macro-block. We use mb_x and mb_y to determine whether the sub-block is on border of frame. “mb_x” and mb_y are stored into mb_x_cur and mb_y_cur respectively. Both mb_x_cur and mb_y_cur are registers constructed in cavlc_data_ready.

   Finally, CAVLC encoder obtains the coded block pattern of the current macro-block from PR1. The CBP is stored into cbp_cur, which is a register constructed in cavlc_data_ready.

3. 2nd cycle: All coefficients of the sub-block are ready on the output of Residual Buffer. Moreover, cavlc_nunl sends read address of top neighbor to sram_nonzero.

4. 3rd cycle: “cavlc_nunl” sends read address of left neighbor to sram_nonzero. All coefficients are loaded into Input Buffer as Fig. 64 shows, and Nonzero Index Table is determined at the same time. Start index is initialized as fifteen (see section 3.4).

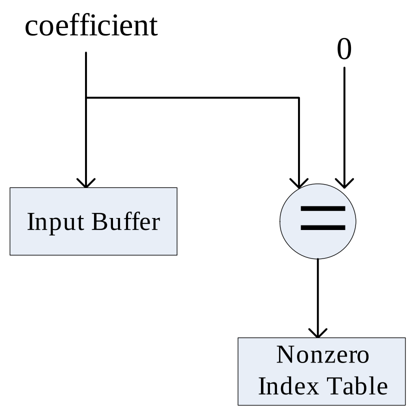


Fig. 64 Loading of Input Buffer and determination of Nonzero Index Table

5. 4th cycle: Start index is equal to 15 and stop index points to "13", as shown in Fig. 65. TotalCoeff is determined by a combinational adder as Fig. 66 shows.

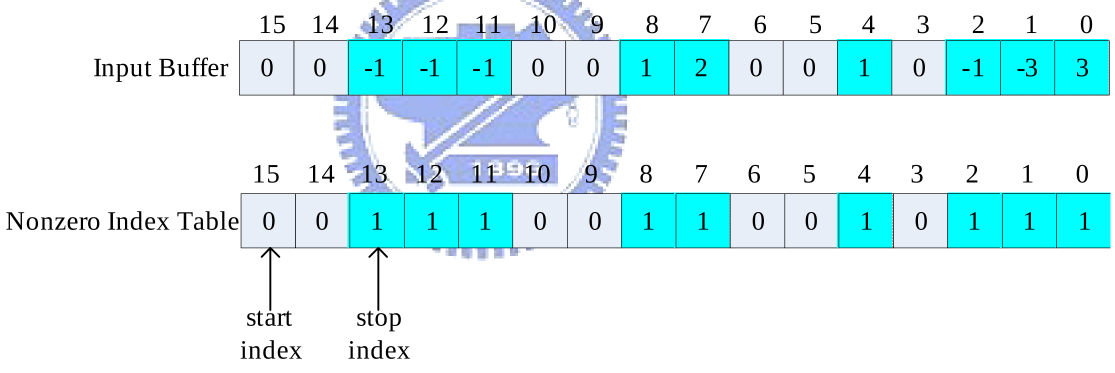


Fig. 65 State of Nonzero Index Table at 4th cycle

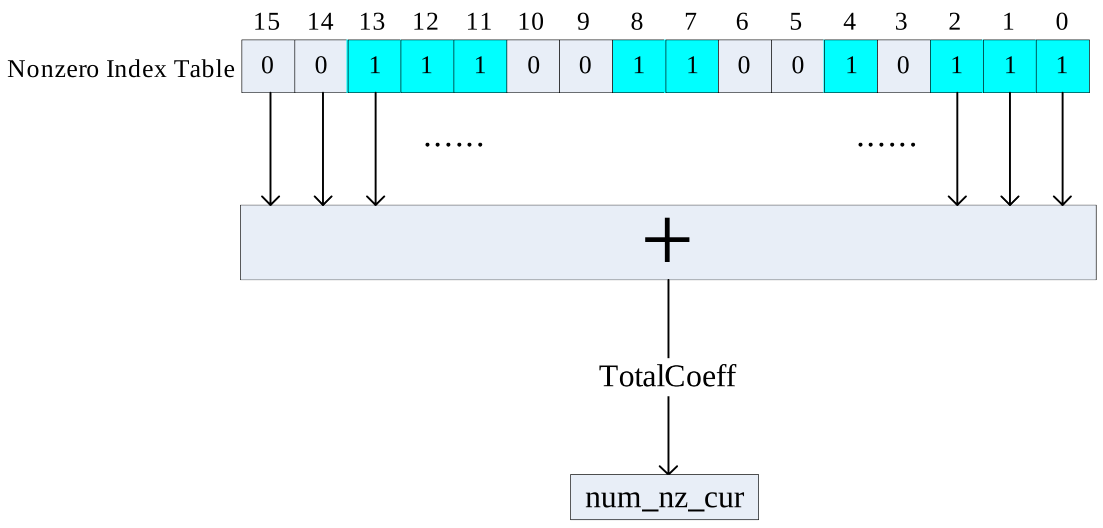


Fig. 66 Determination of TotalCoeff by Nonzero Index Table

TotalCoeff is stored into num_nz_cur which is a register constructed in cavlc_scan. “total_zeros” is determined by Eq. 6 as shown in Fig. 67. “total_zeros” is stored into total_run_cur which is a register constructed in cavlc_scan.

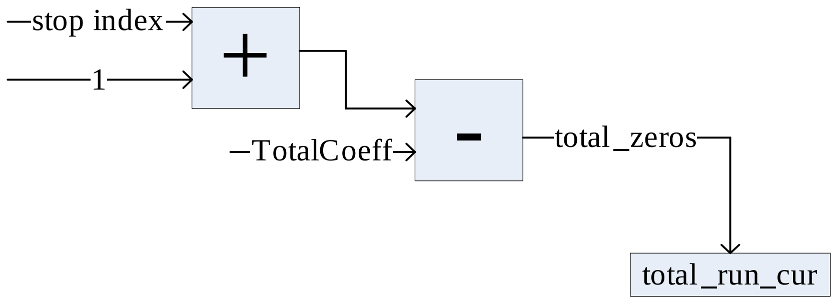


Fig. 67 Determination of total_zeros

The coefficient pointed by stop index is a trailing one whose trailing_one_sign_flag is one, so num_t1_cur, a register constructed in cavlc_scan, is increased as Fig. 68 shows. The meaning of num_t1_cur is the same as TrailingOnes. Trailing_one_sign_flags are stored into sign_t1_cur, a register constructed in cavlc_scan.

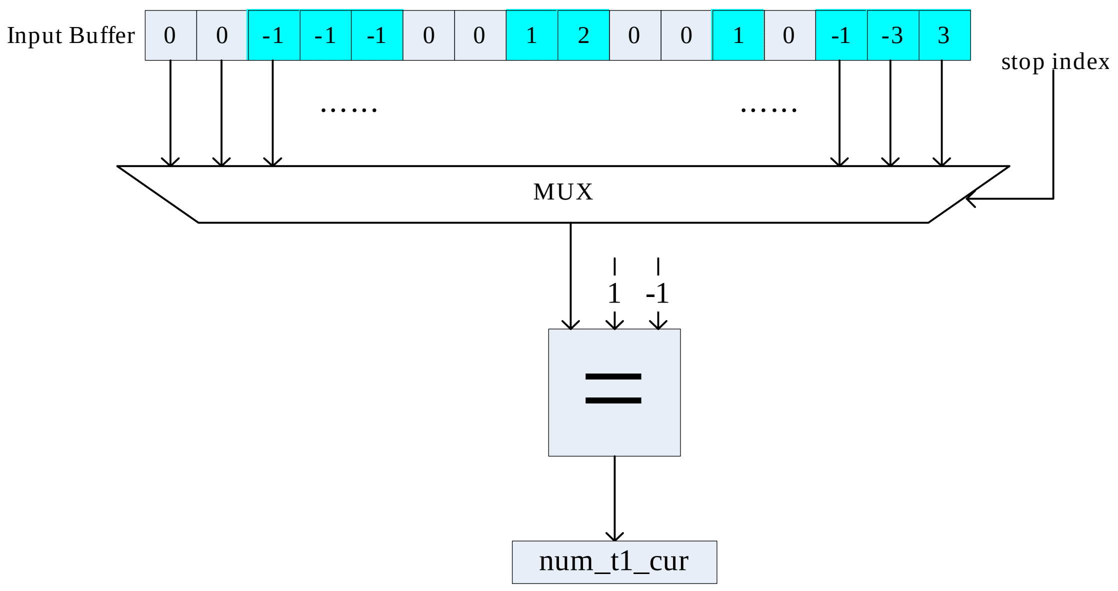


Fig. 68 Determination of TrailingOnes

6. 5th cycle: The TotalCoeff of top neighbor comes from sram_nonzero and is stored into nu_cur, which is a register constructed in cavlc_nunl. Nonzero Index Table is updated as Fig. 69 shows.

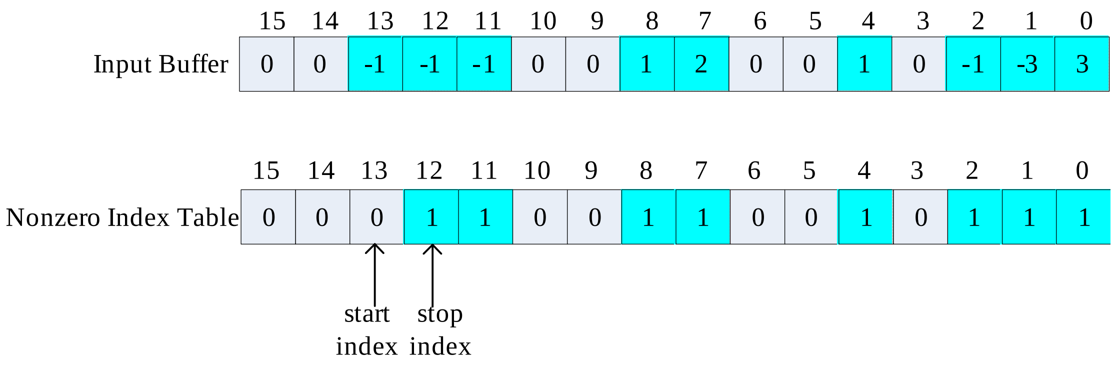


Fig. 69 State of Nonzero Index Table at 5th cycle

The coefficient pointed by stop index, denoted by Input Buffer [12], is a trailing one whose trailing_one_sign_flag is one, so num_t1_cur is increased. The run_before of Input Buffer [13] can be calculated by Eq. 7 as shown in Fig. 70.

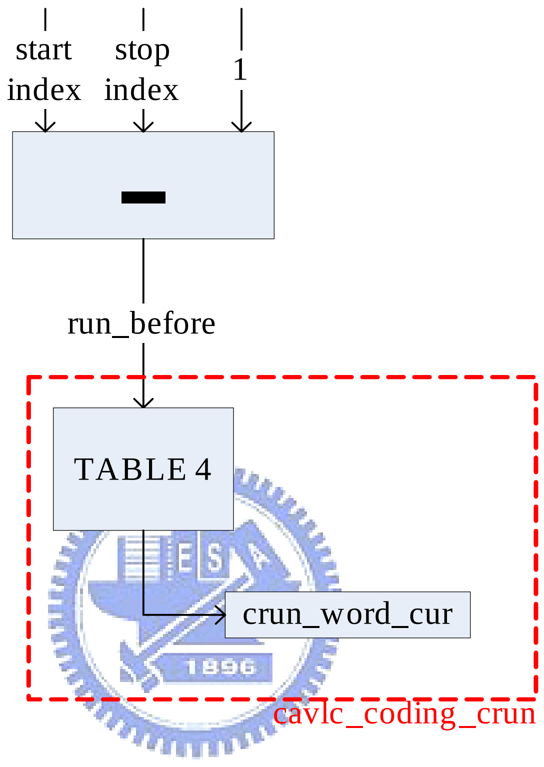


Fig. 70 Determination and concatenation of run_before

The codeword ("11") of the run_before is generated by TABLE 4, implemented as a combinational circuit in cavlc_coding_crun. The codeword is concatenated into crun_word_cur, a 32-bit register constructed in cavlc_coding_crun.

7. 6th cycle: The TotalCoeff of left neighbor comes from sram_nonzero and is stored into nl_cur, which is a register constructed in cavlc_nunl. Now that nu_cur and nl_cur are settled, nC can be determined in the cycle. Nonzero Index Table is updated as Fig. 71 shows.

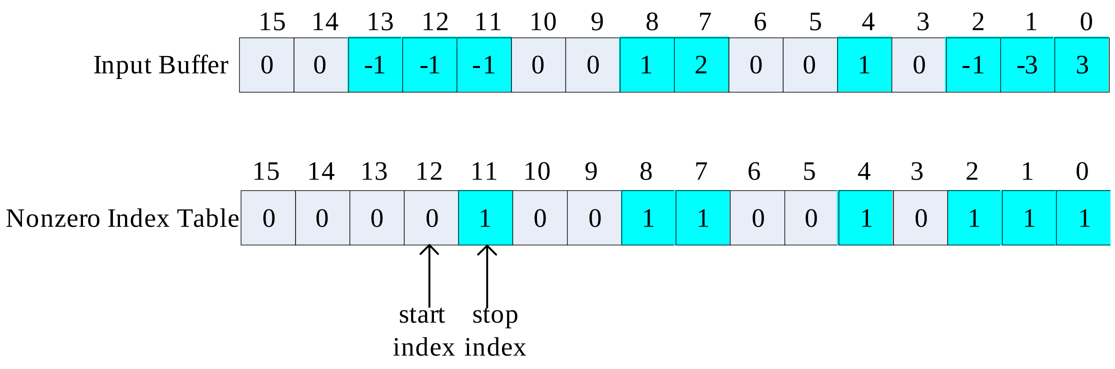


Fig. 71 State of Nonzero Index Table at 6th cycle

The coefficient pointed by stop index is a trailing one whose trailing_one_sign_flag is one, so num_t1_cur is increased. The run_before of Input Buffer [12] is zero and the run_before codeword ("11") is concatenated into crun_word_cur.

8. 7th cycle: Nonzero Index Table is updated as Fig. 72 shows. The value of num_t1_cur is equal to three which is the maximum of TrailingOnes, so the coefficient pointed by stop index is not a trailing one, although its magnitude is equal to one. Because the coefficient pointed by stop index is not a trailing one, trailing ones never occur at the following coefficients such that num_t1_cur and sign_t1_cur are settled.

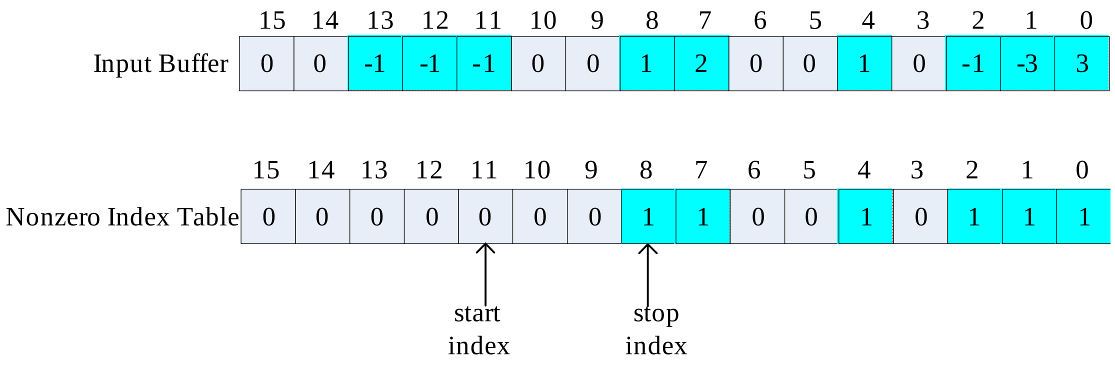


Fig. 72 State of Nonzero Index Table at 7th cycle

Now that nC, TotalCoeff, and TrailingOnes are settled, the coeff_token codeword is valid and is concatenated into Bit-stream Packer. The codeword of coeff_token is generated by TABLE 2, which is implemented as a combinational circuit in cavlc_coding as Fig. 73 shows.

The run_before of Input Buffer [11] is two and the run_before codeword ("011") is concatenated into crun_word_cur.

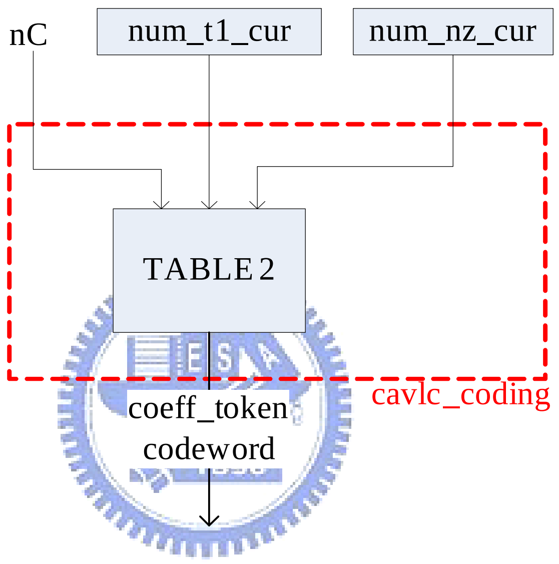


Fig. 73 Generation of coeff_token codeword

9. 8th cycle: "sign_t1_cur" is concatenated into Bit-stream Packer. No updating of Nonzero Index Table occurs.

10. 9th cycle: The state of Nonzero Index Table is the same as 7th cycle. See Fig. 72, Input Buffer [8] is a level, which is the coefficient at stop index. The level codeword, generated by cavlc_coding_level as shown in Fig. 74, is concatenated into Bit-stream Packer.

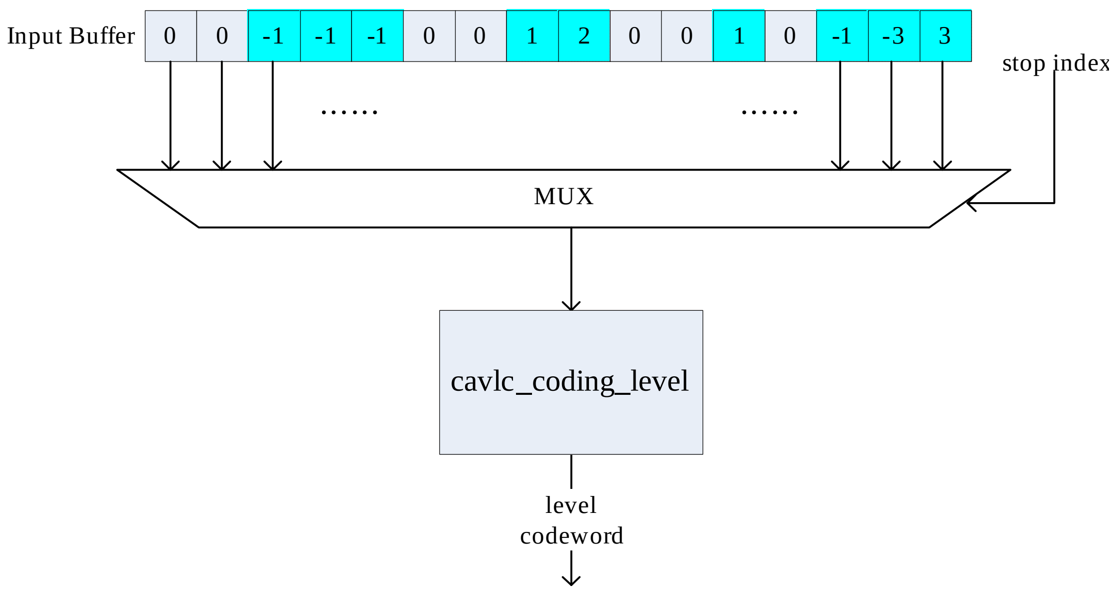


Fig. 74 Generation of level codeword

11. 10th cycle: Nonzero Index Table is updated as Fig. 75 shows. The level codeword of Input Buffer [7], the coefficient as stop index, is concatenated into Bit-stream Packer. The run_before codeword ("11") of Input Buffer [8] is concatenated into crun_word_cur.

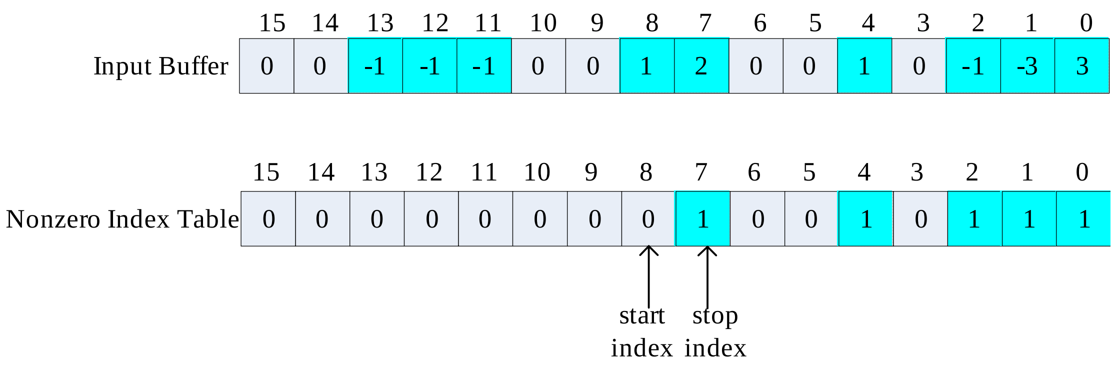
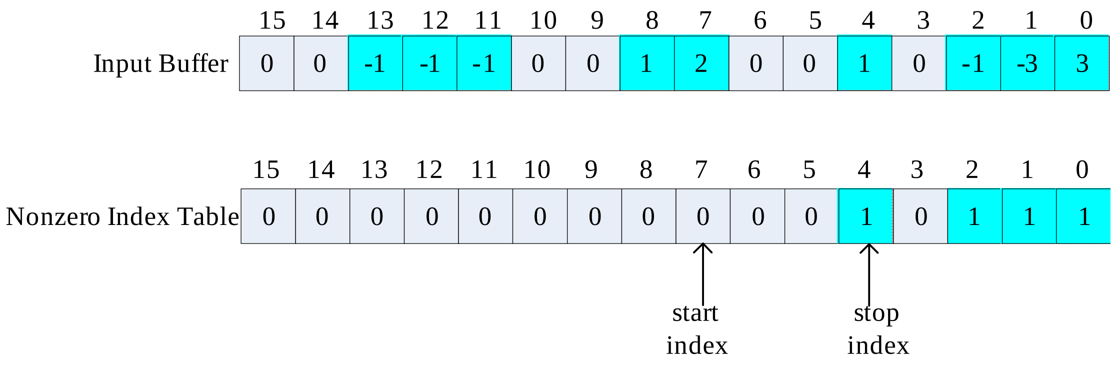


Fig. 75 State of Nonzero Index Table at 10th cycle

12. 11th cycle: Nonzero Index Table is updated as shown in Fig. 76. The level codeword of Input Buffer [4], the coefficient at stop index, is concatenated

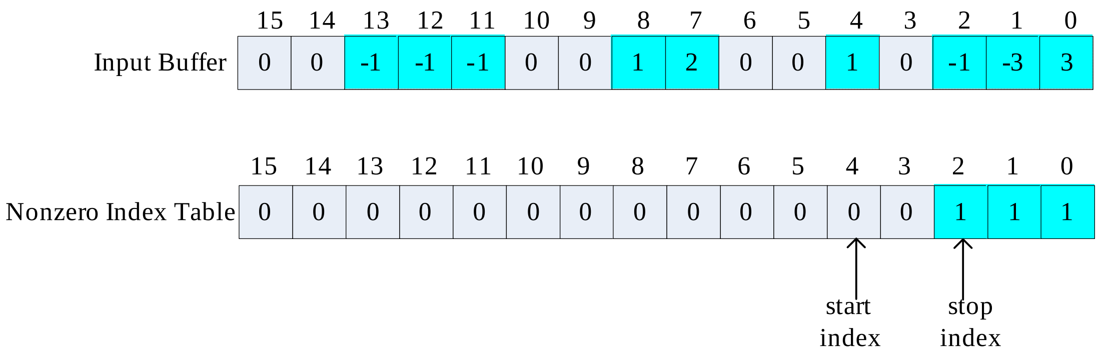
into Bit-stream Packer. The run_before codeword (“01”) of Input Buffer [7] is concatenated into crun_word_cur.



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Buffer	0	0	-1	-1	-1	0	0	1	2	0	0	1	0	-1	-3	3
Nonzero Index Table	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1

Fig. 76 State of Nonzero Index Table at 11th cycle

13. 12th cycle: Nonzero Index Table is updated as shown in Fig. 77. The level codeword of Input Buffer [2], the coefficient at stop index, is concatenated into Bit-stream Packer. The run_before codeword (“0”) of Input Buffer [4] is concatenated into crun_word_cur.



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input Buffer	0	0	-1	-1	-1	0	0	1	2	0	0	1	0	-1	-3	3
Nonzero Index Table	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

Fig. 77 State of Nonzero Index Table at 12th cycle

14. 13th cycle: Nonzero Index Table is updated as shown in Fig. 78. The level codeword of Input Buffer [1], the coefficient at stop index, is concatenated

into Bit-stream Packer. Note that we have encoded all run_befores at 12th cycle such that zerosLeft becomes zero. Therefore, no run_before is encoded after 12th cycle (see section 2.3.1).

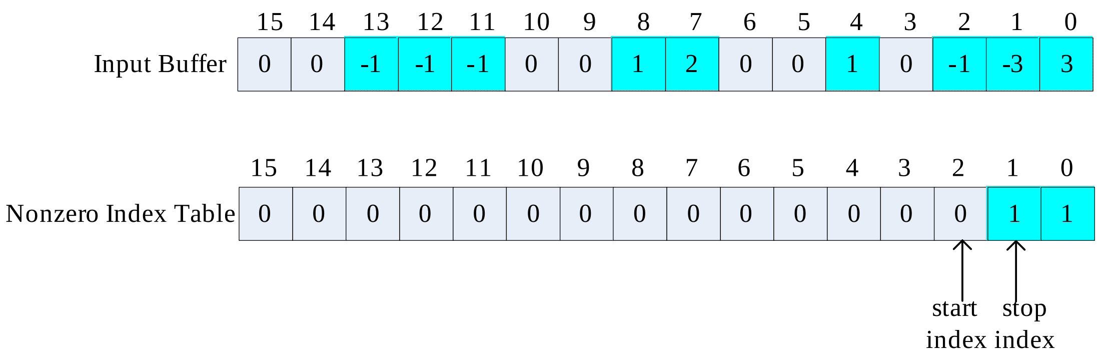


Fig. 78 State of Nonzero Index Table at 13th cycle

15. 14th cycle: Nonzero Index Table is updated as shown in Fig. 79. The level codeword of Input Buffer [0], the coefficient at stop index, is concatenated into Bit-stream Packer.

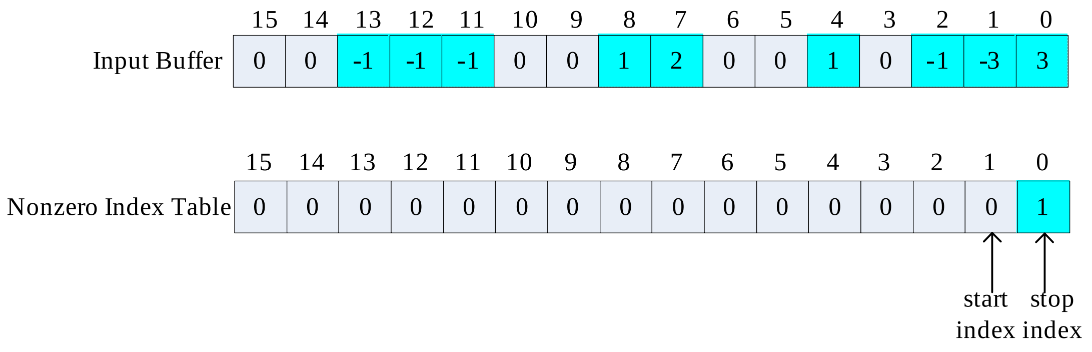


Fig. 79 State of Nonzero Index Table at 14th cycle

16. 15th cycle: Nonzero Index Table is updated as Fig. 80 shows. Note that no one bit exists in Nonzero Index Table. The fact means that encoding of levels is

done.

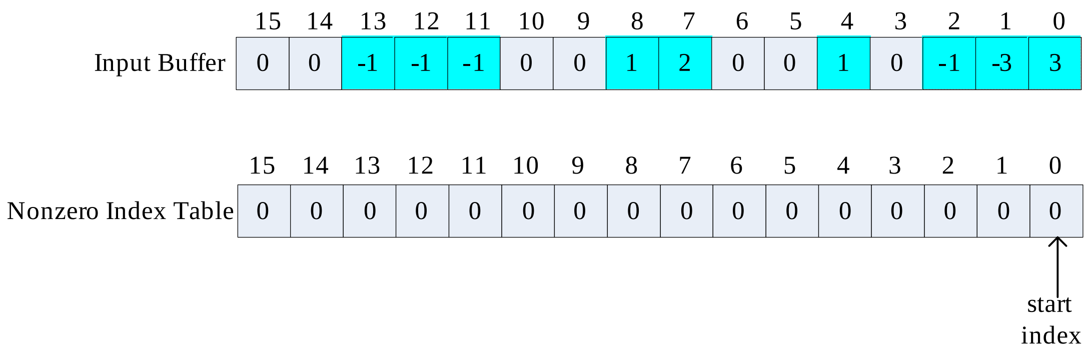


Fig. 80 State of Nonzero Index Table at 15th cycle

17. 16th cycle: The total_zeros codeword, generated by cavlc_coding as shown in Fig. 81, is concatenated into Bit-stream Packer. Because total_run_cur and num_nz_cur have been settled at 4th cycle, the total_zeros codeword is valid right now. Note that TABLE 3 is constructed as a combinational circuit in cavlc_coding.

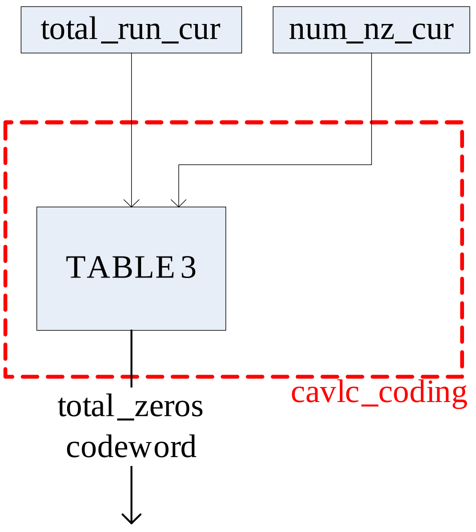


Fig. 81 Generation of total_zeros codeword

18. 17th cycle: "crun_word_cur" is concatenated into Bit-stream Packer, which is the concatenation of each run_before codeword. Then, the encoding of the NAZ is finished.

3.4.2 Hardware operation for encoding a NSZB

We assume the sub-block is LUMA. The hardware operation of the proposed CAVLC encoder is described cycle-wise below:

1. 0th - 2nd cycle: The same as section 3.4.1.
2. 3rd cycle: Input Buffer and Nonzero Index Table are shown in Fig. 82. As shown in Fig. 83, if FindLeadingOne cannot find any nonzero bit in Nonzero Index Table, zero_blk_cur is set one to indicate that the sub-block is all-zero. "zero_blk_cur" is 1-bit register constructed in cavlc_scan. Moreover, "cavlc_nunl" sends read address of left neighbor to sram_nonzero.

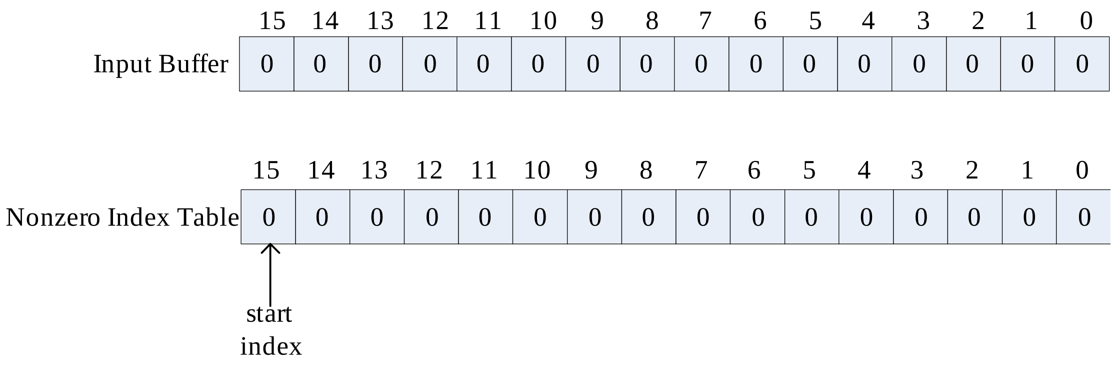


Fig. 82 State of Nonzero Index Table at 3rd cycle for NSZB

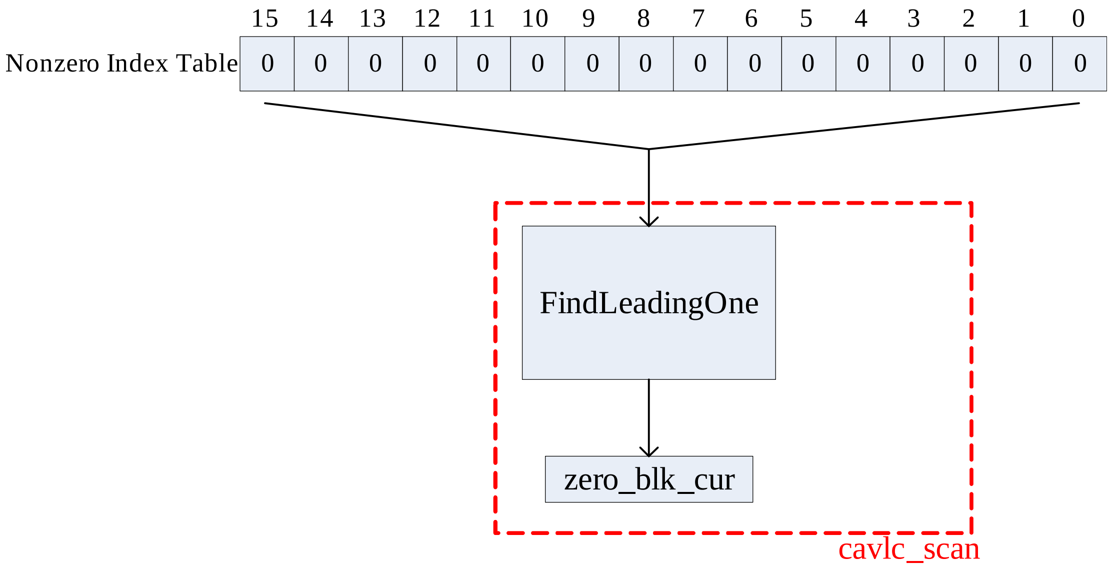


Fig. 83 Determination of an all-zero sub-block

3. 4th cycle: The TotalCoeff of top neighbor comes from sram_nonzero and is stored into nu_cur.

4. 5th cycle: The TotalCoeff of left neighbor comes from sram_nonzero and is stored into nl_cur. Because nu_cur and nl_cur are settled, nC can be determined in the cycle. Therefore, the codeword generated by Zero-block Codeword Table (see Fig. 61) is valid and it can be concatenated into Bit-stream Packer. Then, the encoding of the NSZB is finished.

3.4.3 Hardware operation for encoding a CS

1. 0th - 1st cycle: The same as section 3.4.1.

2. 2nd cycle: According to blk_idx_cur, blk_typ_cur, and cbp_cur, we can confirm that the sub-block is a CS. No codeword is generated and the encoding of the CS is finished.

3.5 Memory system

In this section, we present the size requirement and organization of SRAM (sram_nonzero) constructed in the proposed CAVLC encoder. Section 2.2.4 shows that a macro-block has 24 sub-blocks which may be neighbors of other sub-blocks. For calculating nC, we must record TotalCoeff of the 24 sub-blocks. The neighbors may come from other macro-blocks as shown in Fig. 84. Fig. 84 shows a frame of M by N macro-blocks. The macro-blocks of Row 1 need information from those of Row 0 to calculate nC. Thus we must record the information of macro-blocks in one row of a frame such that the size of SRAM depends on the frame's width.

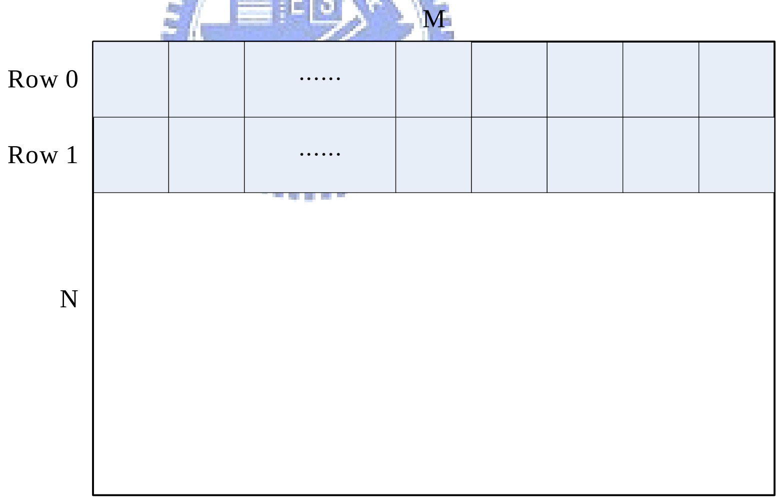


Fig. 84 Two adjacent rows in a frame

We take QCIF as an example and Fig. 85 shows the organization of sram_nonzero for QCIF. Because QCIF's width is equal to 11 macro-blocks, four bits are necessary to represent the macro-blocks. We use five bits to represent the 24 sub-blocks in one macro-block. Therefore the address of SRAM comprises nine bits and the structure of the address is shown in Fig. 86

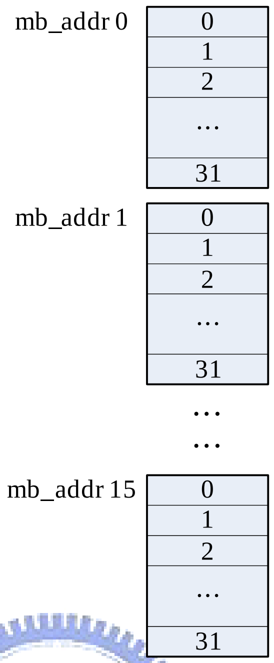


Fig. 85 Organization of sram_nonzero for QCIF

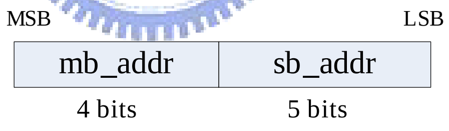


Fig. 86 Structure of sram_nonzero address

The former four bits are denoted as mb_addr which indicate one of the 11 macro-blocks; the remaining five bits are denoted as sb_addr which indicate one of the 24 sub-blocks in a macro-block. One word comprises five bits because the maximum of TotalCoeff is equal to 16. Therefore the size of SRAM for QCIF is equal to 2560 bits, which is expressed as Eq. 8.

$$2^9(word) \times 5(bits/word)$$

Eq. 8

Fig. 87 shows the mapping relation between the numbers in Fig. 85 and the 24 sub-blocks of a macro-block. The smaller the number of a sub-block is, the earlier the sub-block is encoded (see section 2.2.1). For the current sub-block, we read the TotalCoeff of the top and left neighbors and then write the current TotalCoeff into the SRAM, as Fig. 88 shows.

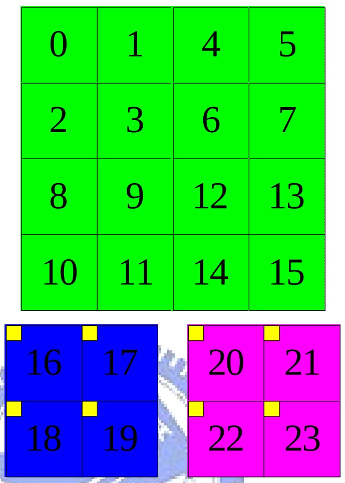


Fig. 87 Mapping relation between sub-blocks and sram_nonzero for a macro-block

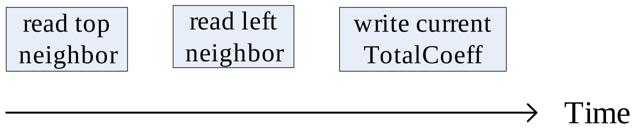


Fig. 88 Timing schedule of sram_nonzero

For each sub-block in Fig. 87, TABLE 11 represents the read addresses to access the neighbors and the write address to record the current TotalCoeff. “X” denotes the current macro-block’s mb_addr, X-1 denotes the left macro-block’s mb_addr, and an address is denoted as {mb_addr, sb_addr}.

TABLE 11 Read and write address of sram_nonzero for a sub-block

Sub-block	top read address	left read address	current write address

0	{X, 10}	{X-1, 5}	{X, 0}
1	{X, 11}	{X, 0}	{X, 1}
2	{X, 0}	{X-1, 7}	{X, 2}
3	{X, 1}	{X, 2}	{X, 3}
4	{X, 14}	{X, 1}	{X, 4}
5	{X, 15}	{X, 4}	{X, 5}
6	{X, 4}	{X, 3}	{X, 6}
7	{X, 5}	{X, 6}	{X, 7}
8	{X, 2}	{X-1, 13}	{X, 8}
9	{X, 3}	{X, 8}	{X, 9}
10	{X, 8}	{X-1, 15}	{X, 10}
11	{X, 9}	{X, 10}	{X, 11}
12	{X, 6}	{X, 9}	{X, 12}
13	{X, 7}	{X, 12}	{X, 13}
14	{X, 12}	{X, 11}	{X, 14}
15	{X, 13}	{X, 14}	{X, 15}
16	{X, 18}	{X-1, 17}	{X, 16}
17	{X, 19}	{X, 16}	{X, 17}
18	{X, 16}	{X-1, 19}	{X, 18}
19	{X, 17}	{X, 18}	{X, 19}
20	{X, 22}	{X-1, 21}	{X, 20}
21	{X, 23}	{X, 20}	{X, 21}
22	{X, 20}	{X-1, 23}	{X, 22}
23	{X, 21}	{X, 22}	{X, 23}

For example, Fig. 89 shows a QCIF frame, where the red square is the current macro-block and the green squares are already encoded macro-blocks. The information of the green macro-block is stored in the SRAM and the number on each green macro-block is the corresponding mb_addr, so X is equal to four. The encoding order of sub-blocks can guarantees that a word in the SRAM is never updated before the word is read by other sub-blocks.

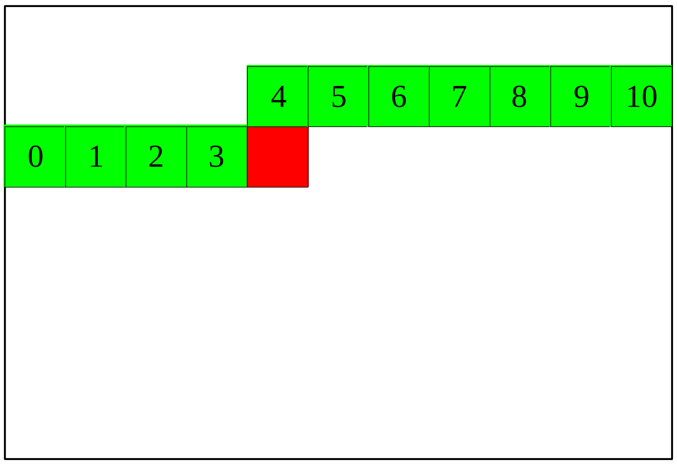


Fig. 89 Current macro-block in a QCIF frame

3.6 Summary

In this chapter, we propose two methods to save cycle counts on zeros which the CBP cannot skip. First, we use Zero-block Codeword Table to encode a NSZB such that the encoding flow can be terminated earlier. Second, Nonzero Index Table with stop index directly jumps to a nonzero coefficient in Input Buffer. Thus zeros in NAZ sub-blocks are skipped. Moreover, we present the size requirement and organization of SRAM in which already encoded TotalCoeff is stored.

Chapter 4. Results and Comparison

4.1 Cycle count analysis

As mentioned in section 3.4, sub-blocks can be divided into the following three categories: NAZ, NSZB, and CS. Below, we show the cycle count analysis for each category.

Fig. 90 shows the timing schedule for encoding a NAZ. “x” is defined as TABLE 12 shows and “y” is defined as Eq. 9. The following things are done during the first 5+x cycles:

1. Fetch coefficients from Entropy SRAM Interface.
2. “cavlc_nunl” interacts with sram_nonzero to calculate nC.
3. Coeff_token, trailing_one_sign_flags, and total_zeros are settled by analyzing Nonzero Index Table.
4. The trailing ones’ run_before codeword is concatenated into cavlc_coding_crun

Afterwards, each syntax element codeword is sent to Bit-stream Packer.

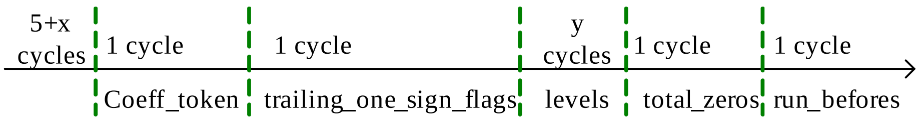


Fig. 90 Timing schedule for encoding a NAZ sub-block

TABLE 12 Definition of variable “x”

TrailingOnes	x
0	0
1	1

2	2
3	2

$$\text{TotalCoeff} - \text{TrailingOnes} + 1 \qquad \text{Eq. 9}$$

Fig. 91 shows the timing schedule for encoding of an NSZB. The following things are done during the first five cycles:

1. Fetch coefficients from Entropy SRAM Interface.
2. "cavlc_nunl" interacts with sram_nonzero to calculate nC.
3. The fact that it is an NSZB is confirmed by analyzing Nonzero Index Table.
4. Zero-block Codeword Table generates Coeff_token codeword.

Afterwards, Coeff_token codeword is sent to Bit-stream Packer and encoding of the NSZB is done.

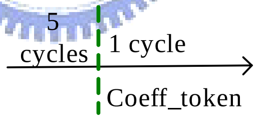


Fig. 91 Timing schedule for encoding of an NSZB sub-block

Fig. 92 shows the timing schedule for encoding of a CS. The following things are done during the three cycles:

1. Fetch coefficients from Entropy SRAM Interface
2. Entropy SRAM provides the coded block pattern and the sub-block type.
3. The fact that it is a CS is confirmed by the coded block pattern and the sub-block type.

Afterwards, no codeword is sent to Bit-stream Packer and encoding of the CS is done.

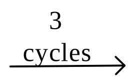


Fig. 92 Timing schedule for encoding of a CS sub-block

In conclusion, TABLE 13 shows the total cycle counts for encoding a NAZ, NSZB, and CS respectively.

TABLE 13 Total cycle counts for CS, NSZB and NAZ

	Total cycle counts per sub-block
CS	3
NSZB	6
NAZ	9+x+y

4.2 Simulation result

Simulation setting is the same as section 3.1. The results are calculated according to the analysis in section 4.1. Fig. 93 shows the average cycle counts to encode a sub-block, including CS, NSZB, and NAZ. Fig. 94 shows the average cycle counts we need to encode a macro-block. We choose 145 MHz as the working frequency of the proposed CAVLC encoder. Fig. 95 presents how many macro-blocks on average are encoded in one second.

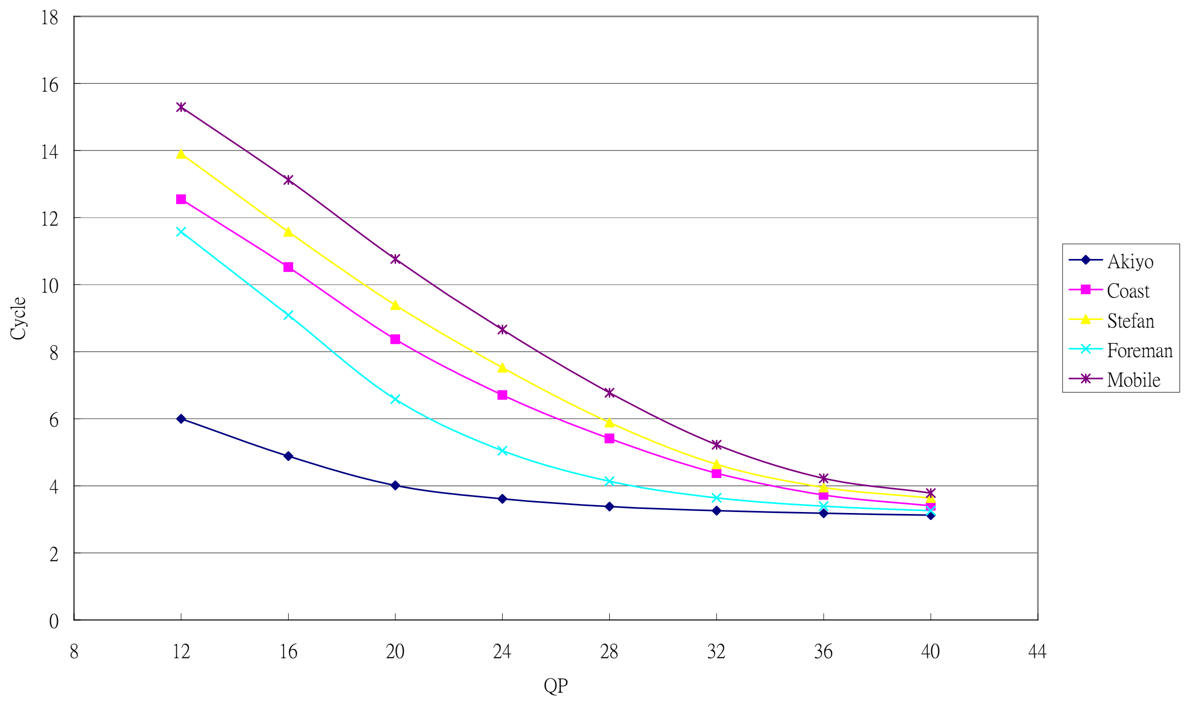


Fig. 93 Average encoding cycles per sub-block

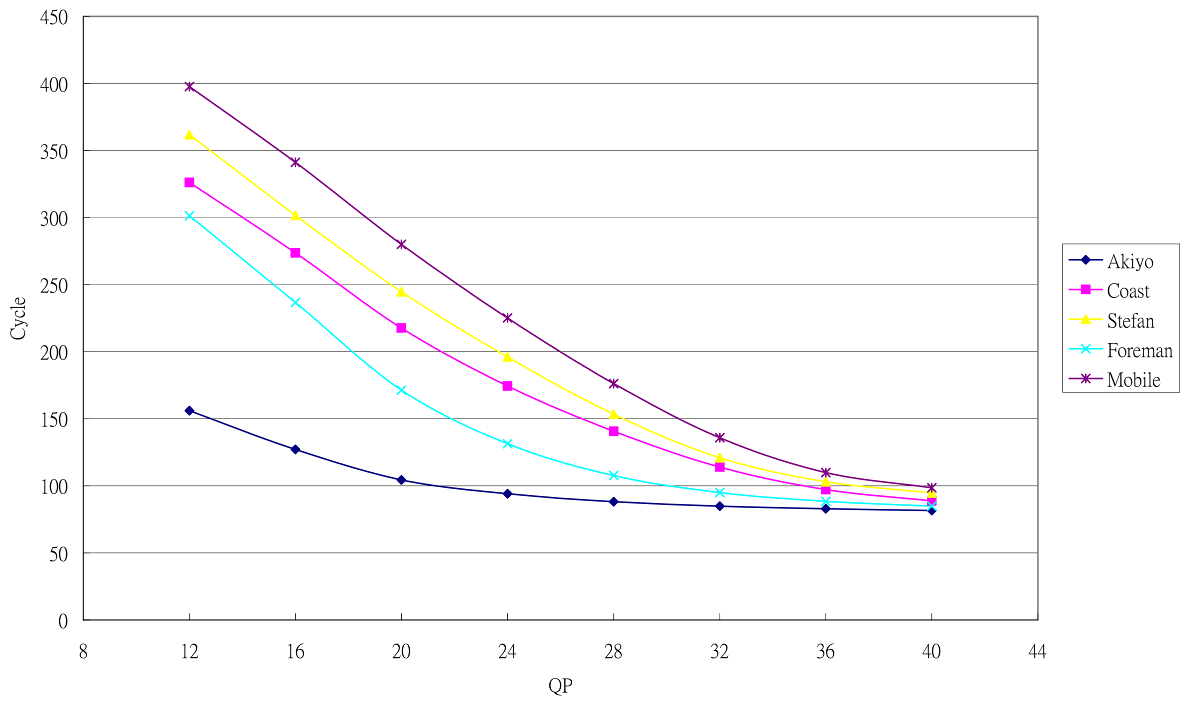


Fig. 94 Average encoding cycles per macro-block

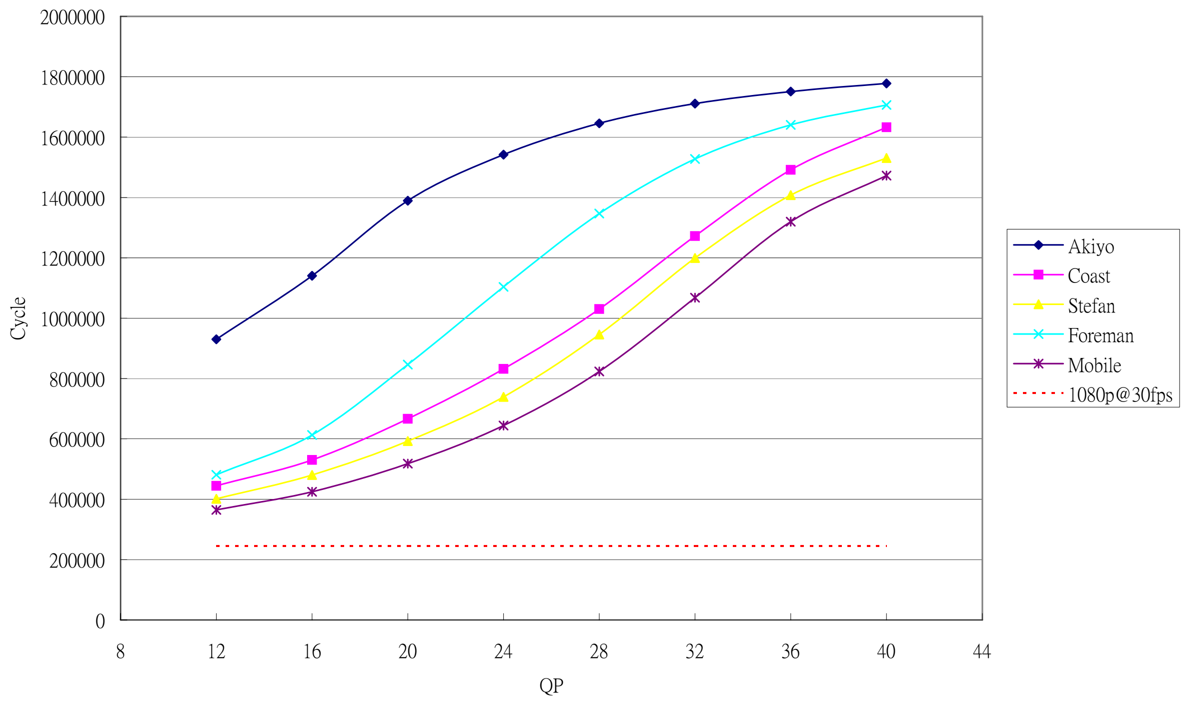


Fig. 95 Average number of macro-blocks encoded per cycle

The red line in Fig. 95 denotes "244800" that is the throughput requirement of 1080p at 30 fps, as TABLE 14 shows [10]. Each case in Fig. 95 is above the requirement and we prove that the proposed CAVLC encoder can support 1080p at 30 fps.

TABLE 14 Requirement of 1080 HD

Format	Width	Height	MBs Total	MBs/second
1080p	1920	1088	8160	244800

4.3 Implementation results and comparison

We used Verilog and UMC 0.13um cell library to implement the proposed CAVLC encoder, which consumes 9.03 K logic gates at 145 MHz. TABLE 15 shows the gate count profile (see Fig. 58) and TABLE 16 shows on-chip memory requirement. TABLE 17 shows the comparison with other designs, where the serial numbers for zero-skipping methods are defined in TABLE 18. In TABLE 17, note that the gate count of Lai [14] includes the bit-stream packer. TABLE 19 shows the gate count profile comparison, where "statistic buffer" only refers to the storage element storing the syntax elements of a sub-block.

TABLE 15 Gate count profile of the propose CAVLC encoder

Item	Gate Count
cavlc_data_ready	3074
cavlc_scan	1252
cavlc_coding	1302
cavlc_coding_level	1176
cavlc_coding_crun	1410
cavlc_nunl	570
cavlc_mux	244
Total	9028

TABLE 16 On-chip memory requirement of the proposed CAVLC encoder

	Number of word	Bits per word	Total bits
QCIF	512	5	2560
1080 HD	4096	5	20480

TABLE 17 Comparison with other CAVLC encoder designs

		Proposed	[3]	[15]	[4]	[13]	[14]
Technology		UMC 0.13um	UMC 0.18um	TSMC 0.18um	FPGA	0.18um CMOS	TSMC 0.35um
Frequency		145MHz	100MHz	27MHz	100MHz	125MHz	28MHz
Gate Count	CAVLC Core	9028	17635	23281	22128	9724	9171
Cycle Count per MB (QP=12)	Foreman	301	350	225	na	na	na
	Mobile	397	430	360			
	Stefan	361	400	na			
	Akiyo	155	na	na	na	na	na

	Coast	326					
	Average	308	393	293	432	402	
Zero-skipping method		1, 2, 3	1	1, 2	none	1	none

TABLE 18 Definition of serial numbers for zero-skipping methods

Serial number	Definition
1	CBP Look-Ahead
2	Nonzero Index Table
3	Zero-block Codeword Table

TABLE 19 Gate count profile comparison

	Proposed	[3]	[15]	[4]	[13]	[14]
coeff_token	1302	864	3758	298	554	na
total_zeros		646		564	420	
levels	1176	1012		2249	1208	
run_befores	1410	263		1361	432	
statistic buffer	0	12283	0	17656	5325	
control	5140	2567	19523		1785	
Total	9028	17635	23281	22128	9724	9171

Chen [3] is an improved version of the CAVLC encoder in Huang [6]. The architecture of Huang [6] is shown in Fig. 96.

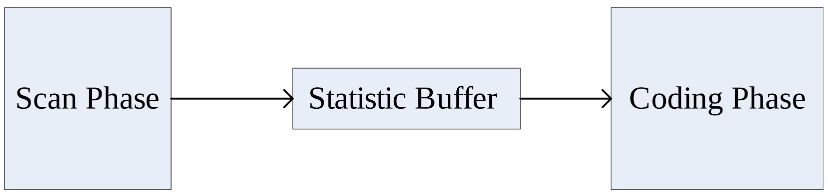


Fig. 96 Architecture of Huang [6]

In Scan Phase, Huang [6] reads one coefficient per cycle and concurrently examine whether the coefficient is nonzero to update TotalCoeff. TotalCoeff is not settled until all coefficients of a sub-block are read. For example, as for a LUMA sub-block, Huang [6] spends at least sixteen cycles to make TotalCoeff settled even if the sub-block consists of many zero coefficients. As a result, many cycle counts are wasted on zero coefficients. Because TotalCoeff is the first syntax element to be encoded into H.264/AVC bit-stream, Huang [6] must use Statistic Buffer to store the syntax elements obtained before TotalCoeff during the sixteen cycles, including levels and run_befores. Otherwise, the syntax elements except TotalCoeff will evaporate after the sixteen cycles. After all syntax elements are ready in Statistic Buffer, Coding Phase generates the codeword of the syntax elements from Statistic Buffer. While Coding Phase is generating the codeword of the current sub-block, Scan Phase cannot write the syntax elements of the next sub-block into Statistic Buffer. Otherwise, Scan Phase will destroy the data of the current sub-block in Statistic Buffer. Therefore, Scan Phase and Coding Phase cannot work in parallel such that the utilization of hardware is halved. To double the utilization, Chen [3] use two statistic buffers switched in ping-pong manner, as shown in Fig. 97.

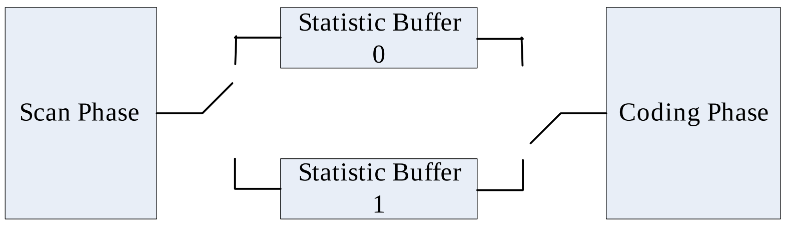


Fig. 97 Architecture of Chen [3]

While Coding Phase is generating the codeword of the current sub-block, whose syntax elements are stored in Statistic Buffer 1, Scan Phase writes the syntax elements of the next sub-block into Statistic Buffer 0, so Scan Phase never destroys the data of the current sub-block in Statistic Buffer 1. Therefore, Scan Phase and Coding Phase can work in parallel such that the utilization is doubled.

On the contrary, the proposed design reads sixteen coefficients in one cycle and TotalCoeff is concurrently settled by Nonzero Index Table. After a few cycles when stop index meets a level for the first time, Coeff_token and trailing_one_sign_flags have been encoded into H.264/AVC bit-stream. Therefore, we can immediately send the level codeword, output of combinational circuits, to the Bit-stream Packer, so the additional redundant buffers are not necessary. As for the run_before, we also use a buffer to store its codeword before sent to Bit-steam Packer. However, Chen [3] and Huang [6] store the un-encoded value of the run_before but we store the encoded codeword of the run_before. Therefore, our buffer gate count is sure to be smaller than Chen [3]. In summary, the proposed design does not need Statistic Buffers to save syntax elements, so the proposed design consumes less logic gates than Chen [3]. Moreover, Chen [3] spends at least sixteen cycles for a sub-block even if the sub-block consists of many zero coefficients. As a result, Chen [3] wastes many cycle counts on zero coefficients. That is the reason why the proposed design can spend less cycle counts per macro-block than Chen [3].

The architecture of Kim [4] is similar to Huang [6], as Fig. 98 shows. Compared with Fig. 96, Kim [4] replaces Statistic Buffer with FIFO. Although Kim [4] has only one storage element, Kim [4] has the same pipelining schedule as Chen [3]. In other words, Kim [4] can process two sub-blocks at the same time.

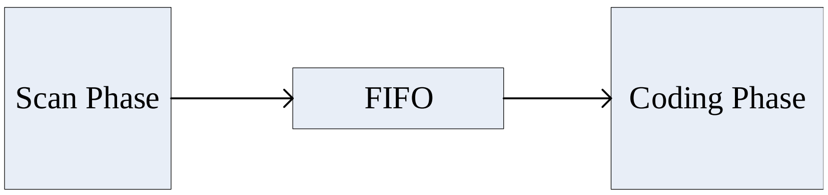


Fig. 98 Architecture of Kim [4]

This is because the FIFO is an IP with a handshaking protocol [8] which grants the new data coming in without destructing the old data. However, Kim [4] spends more cycle counts per macro-block than Chen [3] because Chen [3] takes advantage of CBP Look-Ahead [3]. The FIFO is the reason why Kim [4] consumes more logic gates than the proposed design.

The architecture of Chien [13] is similar to Huang [6], as Fig. 99 shows. Compared with Fig. 96, Chien [13] replaces Statistic Buffer with SIPO (Serial Input Parallel Output).

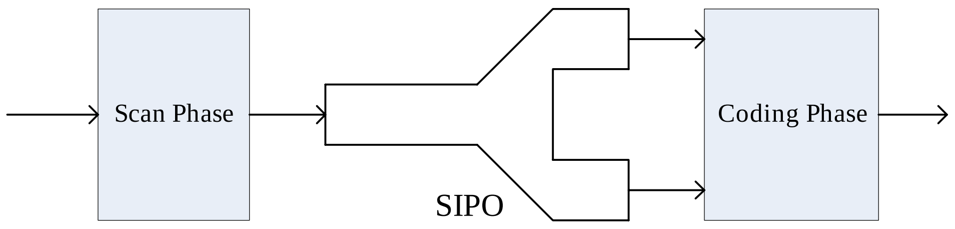


Fig. 99 Architecture of Chien [13]

Although Chien [13] has only one storage element, Chien [13] can concurrently process two sub-blocks as Chen [3]. That is because SIPO accepts one syntax element per cycle from Scan Phase and SIPO sends two syntax elements per cycle to Coding Phase. Therefore the old data are fetched by Coding Phase before they are destroyed by the new data from Scan Phase. However, SIPO is the reason why Chien [13] consumes more logic gates than the proposed design. On the other hand, Chien [13] spends more cycle counts than the proposed design because CBP Look-Ahead is the only zero-skipping method.

The architecture of Lai [14] is similar to Huang [6], as Fig. 100 shows. Compared with Fig. 96, Lai [14] replaces Statistic Buffer with Stack. The Stack is also a storage element to store syntax elements generated by Scan Phase.

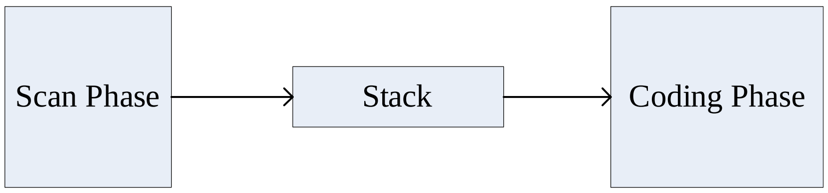


Fig. 100 Architecture of Lai [14]

Like Huang [6], while Coding Phase fetches syntax elements from the Stack, Scan Phase cannot write the syntax elements of the next sub-block into the Stack. Moreover, Scan Phase also spends at least sixteen cycle counts to obtain all syntax elements of a sub-block. Thus, Lai [14] may spend as many cycle counts in encoding a sub-block as Huang [6]. So Lai [14] should spend more cycle counts per macro-block than the proposed design. On the other hand, Lai [14] should consume more logic gates than the proposed design because of the Stack. Note that the total gate count of Lai [14] is close to that of the proposed design. This is because Lai [14] chooses a much lower working frequency than the proposed design does.

Fig. 101 shows the architecture of Tsai [15]. TQ (Transform Quantize) Stage [15] writes coefficients of a macro-block into Residual SRAM [15]. At the same time, TQ Stage records which coefficients are nonzero in Non-zero Flag Reg [15] and TQ Stage records which coefficients are 1 or -1 in Abs-one Flag Reg [15]. A macro-block consists of about 384 coefficients, including luminance and chrominance, so both Non-zero Flag Reg and Abs-one Flag Reg comprise 384 bits.

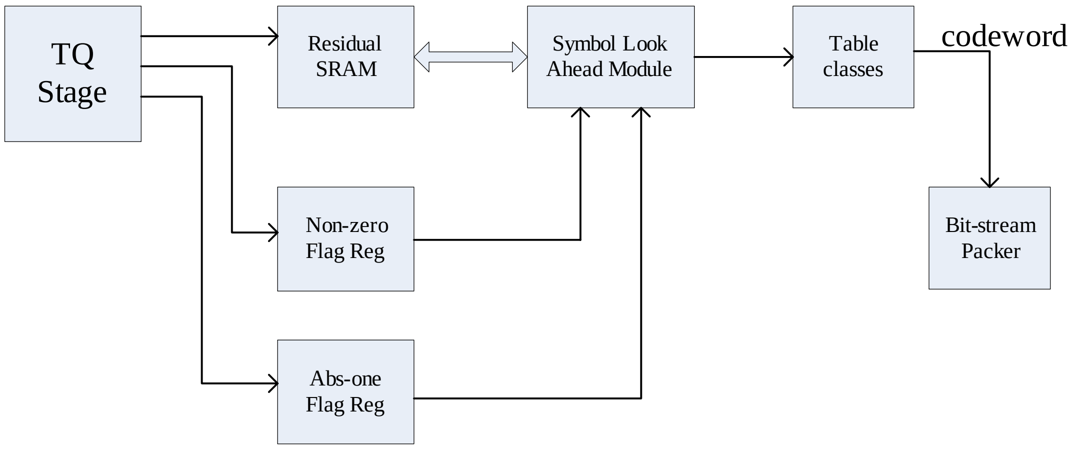


Fig. 101 Architecture of Tsai [15]

First, Symbol Look Ahead Module [15] uses Non-zero Flag Reg and Abs-one Flag Reg to generate TotalCoeff and TrailingOnes in one cycle, whose codeword is concurrently calculated by Table classes [3] and concatenated by Bit-stream Packer. Then Symbol Look Ahead Module directly accesses one nonzero coefficient per cycle according to Non-zero Flag Reg, whose codeword is calculated by Table classes at the same time. Like the proposed design, because TotalCoeff and TrailingOnes have been encoded, Tsai [15] does not need any Statistic Buffer to store the coefficient and the codeword can be immediately concatenated into Bit-stream Packer. However, Tsai [15] consumes more logic gates than the proposed design. The large gate count of Tsai [15] results from the MB-sized Non-zero Flag Reg and Abs-one Flag Reg. On the other hand, note that Tsai [15] spends fewer cycle counts than the proposed design. The reasons may be the following:

1. The operation of Non-zero Flag Reg is similar to that of Nonzero Index Table, because both can skip zero coefficients in NAZ.
2. Abs-one Flag Reg makes Tsai [15] spend fewer cycle counts on calculating TrailingOnes than the proposed design.
3. In TABLE 17, the cycle count of the proposed design includes not only the time for the

CAVLC core process but also the time for interacting with Entropy SRAM Interface, as mentioned in section 3.4.1 and section 4.1. However, Tsai [15] does not disclose the method of calculating cycle count.

Last but not least, we consume less logic gates under higher frequency. In other words, although we have no such statistic buffers to pipeline, the proposed design has a shorter critical path than other designs. Moreover, the demand for the statistic buffer originates from data evaporation. Therefore, critical path is not a point of controversy.

In summary, the designs which need statistic buffers often consume large logic gate count, as TABLE 19 shows. TABLE 17 shows the proposed design can achieve up to 61% decrease in logic gate count. On the other hand, Nonzero Index Table and Zero-block Codeword Table can skip zero coefficients which CBP Look-Ahead [3] cannot detect. In other words, the proposed design can skip every zero coefficient in a macro-block. Therefore, the proposed design achieves higher throughput than other designs. TABLE 17 shows the proposed design can achieve up to 29% decrease in cycle count.

Chapter 5. Conclusions

The main contribution of this thesis is to propose two new hardware architectures which permit direct significance encoding. The proposed architectures compensate for drawbacks of conventional CBP Look-Ahead such that CAVLC encoder for H.264/AVC can achieve a higher throughput. CBP Look-Ahead uses the coded block pattern to detect occurrence of an all-zero sub-block in advance such that encoding time for the all-zero sub-block can be saved. However, our statistics in section 3.1.2 shows that many all-zero sub-blocks are not covered by the coded block pattern, especially at middle QP and low QP. Moreover, a NAZ sub-block is sure to comprise some zero coefficients, which the coded block pattern cannot detect by definition. Our statistics in section 3.1.1 shows that a NAZ sub-block comprises up to 60% zero coefficients on average. As a result, CBP Look-Ahead is not enough to significantly improve throughput of a CAVLC encoder. Thus we propose two new architectures to save encoding cycle counts of the zero coefficients which the coded block pattern cannot take into account. First, Zero-block Codeword Table permits encoding flow to be terminated in advance for an NSZB sub-block. Second, Nonzero Index Table directly skips several zero coefficients in one cycle for a NAZ sub-block. The proposed CAVLC encoder adopts Zero-block Codeword, Nonzero Index Table, as well as CBP Look-Ahead, such that we can save encoding time of all zero coefficients in a macro-block. Comparison in section 4.3 shows that the proposed design achieves a higher throughput than other designs. Simulation results in section 4.2 shows that the proposed design can support 1080p at 30fps.

Moreover, the proposed design has the advantage of fewer gate counts even at a higher working frequency. This is because our design does not need statistic buffers to save intermediate syntax elements, as mentioned in section 4.3.

Eventually, the proposed design can support 1080p at 30 fps with 9.03 K gate count at 145 MHz. We save 61% logic gate count and 29% cycle count.

Bibliography

[1] *Advanced video coding for generic audiovisual services. ITU-T Recommendation H.264, March 2005.*

[2] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, T. Wedi, "Video coding with H.264/AVC: tools, performance, and complexity", in IEEE Circuits and Systems Magazine, vol. 4, no. 1, 2004, pp. 7-28

[3] T.-C. Chen, Y.-W. Huang, C.-Y. Tsai, B.-Y. Hsieh, L.-G. Chen, "Architecture Design of Context-Based Adaptive Variable-Length Coding for H.264/AVC", IEEE Transactions on Circuits and Systems Part II: Express Briefs, vol. 53, no. 9, pp. 832-836, September 2006.

[4] Daeok Kim, Eungu Jung, Hyunho Park, Hosoon Shin, Dongsoo Har, "Implementation of High Performance CAVLC for H.264/AVC Video Codec", The 6th International Workshop on System-on-Chip for Real-Time Applications, pp. 20-23, December 2006.

[5] Min-Chi Tsai, Tian-Sheuan Chang, "High Performance Context Adaptive Variable Length Coding Encoder for MPEG-4 AVC/H.264 Video Coding", IEEE Asia Pacific Conference on Circuits and Systems, pp. 586-589, December 2006.

[6] Yu-Wen Huang, Bing-Yu Hsieh, Tung-Chien Chen, Liang-Gee Chen, "Hardware architecture design for H.264/AVC intra frame coder", Proceedings of the 2004 International Symposium on Circuits and Systems, vol. 2, pp. 269-272, May 2004.

[7] Tung-Chien Chen, Yu-Wen Huang, Chuan-Yong Tsai, Bing-Yu Hsieh, Liang-Gee Chen, "Dual-block-pipelined VLSI architecture of entropy coding for H.264/AVC baseline profile", IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test, pp. 271-274, April 2005.

[8] Synchronous FIFO 5.0, Product Specification. Available: http://www.xillinx.com.

[9] Joint Video Team Reference Software JM9.0. Available: http://iphome.hhi.de/suehring/tml/download/old_jm/

[10] Yi-Hong Huang, "Context Adaptive Binary Arithmetic Decoder of H.264/AVC for Digital TV Application," M.S. thesis, Dept. Elect. Eng., Inst. Elect. Eng., N.C.T.U., Taiwan, 2006.

[11] Iain Richardson. (2002, October 17). H.264 / MPEG-4 Part 10 White Paper. Available: http://www.vcodex.com

[12] Ribin Zan, Asral Bahari, A.T. Erdogan, T. Arslan, "Low power CAVLC architecture for MPEG-4 Advanced Video Coding," unpublished.

[13] Chih-Da Chien, Keng-Po Lu, Yi-Hung Shih, Jiun-In Guo, "A high performance CAVLC encoder design for MPEG-4 AVC/H.264 video coding applications", IEEE

International Symposium on Circuits and Systems, pp. 3838-3841, May 2006.

[14] Yeong-Kang Lai, Chih-Chung Chou, Yu-Chieh Chung, “A simple and cost effective video encoder with memory-reducing CAVLC”, IEEE International Symposium on Circuits and Systems, vol. 1, pp. 432-435, May 2005.

[15] Chuan-Yung Tsai, Tung-Chien Chen, Liang-Gee Chen, “Low Power Entropy Coding Hardware Design for H.264/AVC Baseline Profile Encoder”, IEEE International Conference on Multimedia and Expo, pp. 1941-1944, July 2006.

[16] Iain E. G. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia*. Wiley, 2003, pp. 160 - 161.

作者簡歷

姓名: 吳秈璟

籍貫: 嘉義市

學歷:	雲林縣斗六市私立正心高級中學		(1998 年 9 月-2001 年 6 月)
	國立交通大學電子工程學系	學士	(2001 年 9 月-2005 年 6 月)
	國立交通大學電子研究所	碩士	(2005 年 9 月-2007 年 6 月)